Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Extending the DAPHNE Runtime: Lustre file system integration

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΠΟΣΤΟΛΟΣ ΣΤΑΜΑΤΗΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Extending the DAPHNE Runtime: Lustre file system integration

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΠΟΣΤΟΛΟΣ ΣΤΑΜΑΤΗΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Φεβρουαρίου 2025.

.....	.....	.....
Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2025

Απόστολος Σταματής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Πρόσφατα υπάρχει μια τάση προς τις Ολοκληρωμένες Ροές Ανάλυσης Δεδομένων (Integrated Data Analysis pipelines), οι οποίες ενσωματώνουν διάφορες εργασίες υπολογισμού και επεξεργασίας δεδομένων σε ένα ενιαίο πλαίσιο. Το DAPHNE είναι μια ανοιχτή και επεκτάσιμη υποδομή συστήματος για τέτοιου είδους ροές. Η εργασία αυτή εστιάζει στην ενσωμάτωση του χρόνου εκτέλεσης του DAPHNE με το σύστημα αρχείων Lustre. Το Lustre είναι ένα κατανεμημένο σύστημα αρχείων, συμβατό με το POSIX, το οποίο υιοθετείται ευρέως στον χώρο των High Performance Computing (HPC) συστημάτων, λόγω της δυνατότητάς του να χειρίζεται αποτελεσματικά παράλληλες λειτουργίες I/O. Η διασύνδεση των δύο συστημάτων επιτυγχάνεται μέσω της ανάπτυξη πυρήνων C++ που υποστηρίζουν τις λειτουργίες ανάγνωσης και εγγραφής για CSV και DAPHNE Binary Data Format (dbdf) αρχεία. Η μέθοδος χρήσης ενός ενιαίου αρχείου για όλες τις διεργασίες έχει επιλεγεί ώστε να μειωθεί ο φόρτος σχετικός με τα metadata και να βελτιωθεί η επεκτασιμότητα.

Πραγματοποιήθηκαν πειράματα σε ένα υπολογιστικό σύμπλεγμα στο AWS, ώστε να αναλυθεί η βελτίωση της απόδοσης σε λειτουργίες εγγραφής και ανάγνωσης, η επεκτασιμότητα καθώς αυξάνεται ο αριθμός των κόμβων, και ο αντίκτυπος των αλλαγών σε παραμέτρους του Lustre. Τα αποτελέσματα δείχνουν ότι η ενσωμάτωση με το Lustre βελτιώνει σημαντικά την απόδοση του κατανεμημένου χρόνου εκτέλεσης του DAPHNE και επιτρέπει την καλύτερη επεκτασιμότητά του.

Λέξεις κλειδιά

Κατανεμημένα συστήματα αρχείων, κατανεμημένα συστήματα, σύστημα αρχείων Lustre, DAPHNE

Abstract

Recently, there has been a trend toward Integrated Data Analysis (IDA) pipelines that integrate various computational and data processing tasks within a unified framework. DAPHNE is an open and extensible system infrastructure for such IDA pipelines. This study focuses on the integration of the DAPHNE runtime with the Lustre file system. Lustre is a POSIX-compliant, object-based distributed file system, which is widely adopted in High-Performance Computing (HPC) due to its ability to handle parallel I/O operations efficiently. This integration is achieved via the development of specialized C++ kernels that support read and write operations for CSV and DAPHNE Binary Data Format (dbdf) files. The Single-File approach is selected to reduce metadata overhead and improve scalability.

Experiments were conducted in an AWS-based cluster to analyze performance improvements in read/write operations, scalability with increasing worker nodes, and the impact of various optimization techniques such as stripe size adjustments, file preallocation, and stripe alignment. Results indicate that Lustre integration significantly enhances the performance of DAPHNE's distributed runtime and enables better scalability for large datasets.

Key words

Distributed file systems, distributed systems, Lustre file system, DAPHNE

Ευχαριστίες

Η εκπόνηση αυτής της διπλωματικής εργασίας σηματοδοτεί το τέλος μιας μακράς πορείας. Αναλογιζόμενος τα τελευταία χρόνια, θα ήθελα να ευχαριστήσω τους ανθρώπους που με βοήθησαν και μοιράστηκαν μαζί μου τις δυσκολίες και τις χαρές.

Θα ήθελα να ευχαριστήσω τον καθηγητή μου, κύριο Δημήτρη Τσουμάκο, για την επίβλεψη και καθοδήγηση της παρούσας εργασίας, καθώς επίσης και όλους τους καθηγητές που κατά την διάρκεια της φοίτησής μου με ενέπνευσαν και μου μετέδωσαν το πάθος τους για την Μηχανική Υπολογιστών.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους και την οικογένειά μου, για την στήριξη, την συμπαράσταση, και κυρίως την αγάπη τους καθ' όλη την διάρκεια αυτού του ταξιδιού.

Απόστολος Σταματής,

Αθήνα, 21η Φεβρουαρίου 2025

Περιεχόμενα

Κατάλογος πινάκων

Πίνακες στο αγγλικό κείμενο

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κατάλογος συντομεύσεων

Abbreviation	Meaning
CSV	Comma Seperated Values
dbdf	DAPHNE binary data format
DFS	Distributed File System
DSL	Domain Specific Language
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
I/O	Input/Output
IDA	Integrated Data Analysis
Lnet	Lustre Networking
MDS	Metadata Server
MDT	Metadata Target
MGS	Management Server
MGT	Management Target
OSS	Object Storage Server
OST	Object Storage Target

Κεφάλαιο 0

Εκτενής Περίληψη

0.1 Κατανεμημένα Συστήματα Αρχείων

Τα Κατανεμημένα Συστήματα Αρχείων (Distributed File Systems, DFSs) αποτελούν θεμελιώδες συστατικό των σύγχρονων υπολογιστικών συστημάτων, που επιτρέπουν την αποτελεσματική αποθήκευση, ανάκτηση και διαχείριση δεδομένων σε ένα δίκτυο διασυνδεδεμένων υπολογιστών. Στην πιο βασική τους μορφή, μιμούνται τη λειτουργικότητα ενός μη κατανεμημένου συστήματος αρχείων για προγράμματα-πελάτες που εκτελούνται σε απομακρυσμένους υπολογιστές. Επιπλέον, προσφέρουν δυνατότητες όπως αντιγραφή (replication) αρχείων και αυξημένη επίδοση. Από την πλευρά των προγραμματιστών, η μεγάλη διευκόλυνση είναι ότι δεν απαιτείται αλλαγή στις εφαρμογές ώστε να χρησιμοποιούν κατανεμημένο χώρο αποθήκευσης.

Κατά τον σχεδιασμό ενός Κατανεμημένου Συστήματος Αρχείου, υπάρχουν διάφορες προκλήσεις που πρέπει να αντιμετωπιστούν. Κάποιες από αυτές είναι:

1. Ονομασία και Διαφάνεια
2. Σημασιολογία Κοινής χρήσης
3. Μέθοδοι Απομακρυσμένης Πρόσβασης
4. Ανοχή Σφαλμάτων
5. Επεκτασιμότητα

0.2 Ολοκληρωμένες Ροές Ανάλυσης Δεδομένων (IDA Pipelines)

Οι Ολοκληρωμένες Ροές Ανάλυσης Δεδομένων είναι σύνθετες διαδικασίες συσχετιζόμενες με την διαχείριση και την ανάλυση δεδομένων. Συνήθως αποτελούνται από πολλαπλά διαφορετικά στάδια, μερικά από τα οποία είναι:

1. Εξαγωγή, Μετατροπή, Εισαγωγή
2. Μηχανική Μάθηση
3. Αριθμητικοί Υπολογισμοί και Προσομοιώσεις
4. Ανάλυση Δεδομένων

Τα κύρια χαρακτηριστικά αυτών των ροών είναι ο συντονισμός των διεργασιών, η διαχείριση μεγάλων και ποικιλόμορφων συλλογών δεδομένων και η διασύνδεση διαφορετικών τεχνολογιών (όπως βάσεις δεδομένων και εργαλείων μηχανικής μάθησης).

Παραδείγματαα τέτοιων ροών είναι:

1. Earth Observation (DLR): Ένα project που εστιάζει στην αναγνώριση Τοπικών Κλιματικών Ζωνών (Local Climate Zones, LCZs) χρησιμοποιώντας δεδομένα δορυφόρων. Επεξεργάζεται ετησίως δεδομένα της τάξης των petabytes για την εκπαίδευση μοντέλων μηχανικής μάθησης.

2. Semiconductor Manufacturing (Infineon): Project που στοχεύει στην πρόβλεψη της επιτυχίας ρύθμισης της δέσμης ιόντων κατά την κατασκευή ημιαγωγών. Περιέχει την εξαγωγή στοιχείων, την προεπεξεργασία και την εκπαίδευση μοντέλων μηχανικής μάθησης για να ελαχιστοποιηθούν δαπανηρές παύσεις παραγωγής.

3. Material Degradation (KAI): Το project αυτό αξιολογεί την υποβάθμιση ημιαγωγικών υλικών μέσω επιταχυμένων δοκιμών αντοχής. Τα δεδομένα επεξεργάζονται σε σύστημα HPC για τη μοντελοποίηση της φυσικής υποβάθμισης με την πάροδο του χρόνου.

4. Vehicle Development (AVL): Η ροή αυτή στοχεύει στην βελτιστοποίηση της γεωμετρίας εκτοξευτήρων καυσίμου κυψελών μέσω διαδοχικών προσομοιώσεων δυναμική ρευστών. Ακόμα, προβλέπονται κρίσιμοι δείκτες απόδοσης (KPIs) οχημάτων χρησιμοποιώντας μοντέλα Γκαουσιανής παλινδρόμησης. Βασισμένα σε δεδομένα δοκιμών και προσομοιώσεων.

0.3 Lustre

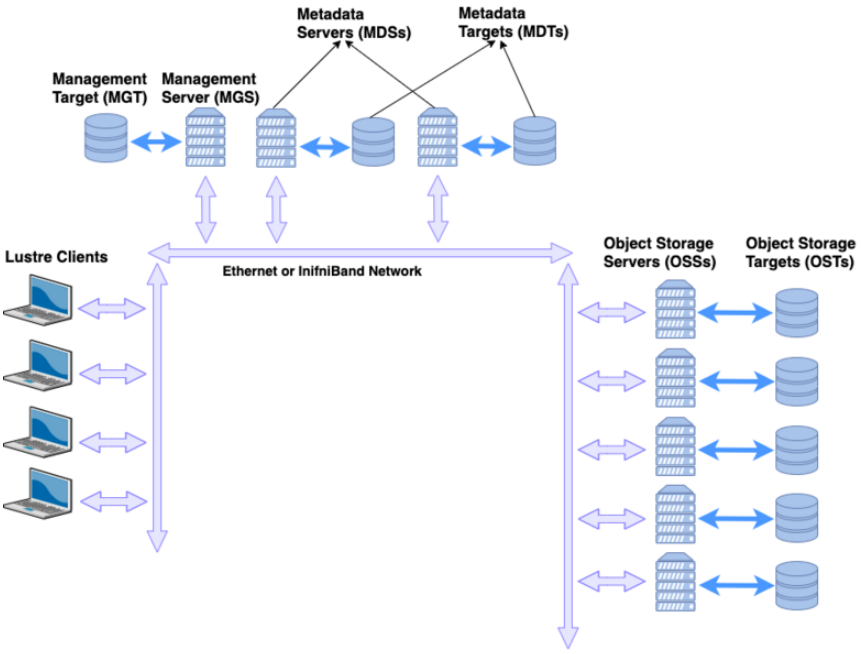
Το Lustre είναι ένα Κατανεμημένο Σύστημα Αρχείων σχεδιασμένο για συστήματα μεγάλης κλίμακας. Είναι υλοποιημένο εξ ολοκλήρου στον πυρήνα και η αρχιτεκτονική του βασίζεται στην κατανεμημένη αποθήκευση αντικειμένων. Είναι ευρέως χρησιμοποιούμενο σε τομείς που απαιτούν εκτενή επεξεργασία δεδομένων, όπως επιστημονική έρευνα, τεχνητή νοημοσύνη, ανάλυση μεγάλου όγκου δεδομένων. Είναι γνωστό για την επεκτασιμότητά του και την δυνατότητα του να εξυπηρετεί χιλιάδες πελάτες και petabytes δεδομένων. Για τον λόγο αυτό χρησιμοποιείται από πληθώρα υπολογιστικών συστημάτων στην λίστα top 500.

Κάποια από τα χαρακτηριστικά που συμβάλουν στην δημοτικότητά του είναι τα εξής:

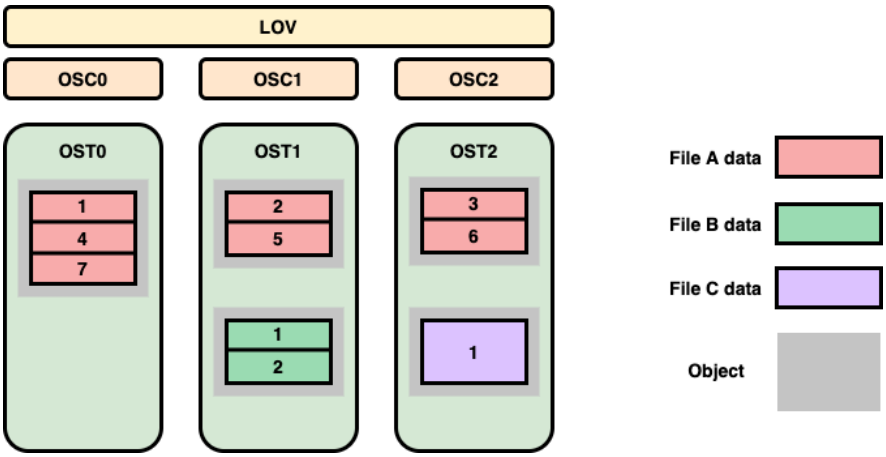
1. Συμβατότητα με το σύστημα POSIX
2. Online έλεγχος αρχείων
3. Ελεγχόμενη διάταξη (layout) αρχείων
4. Υποστήριξη διαφορετικών backend συστημάτων αρχείων, όπως το ldiskfs και το ZFS
5. Υποστήριξη διαφορετικών δικτύων υψηλής επίδοσης
6. Υψηλή διαθεσιμότητα
7. Παροχές Ασφάλειας
8. Αύξηση χωρητικότητας

Τα στοιχεία από τα οποία αποτελείται το Lustre απεικονίζονται στην εικόνα 0.1 και είναι τα εξής:

1. Management Server (MGS)
2. Management Target (MGT)
3. Metadata Server (MDS)
4. Metadata Target (MDT)
5. Object Storage Server (OSS)
6. Object Storage Target (OST)
7. Clients
8. Lustre Networking (LNet)



Σχήμα 0.1: Αρχιτεκτονική του Lustre [16].



Σχήμα 0.2: Κατάτμηση αρχείου στο Lustre [16].

Το Lustre ακολουθεί την λογική της κατάτμησης για να αποθηκεύσει τα αρχεία. Σύμφωνα με αυτήν, το αρχείο σπάει σε κομμάτια, μεγέθους ίσο με την παράμετρο stripe size. Τα κομμάτια αυτά στην συνέχεια αποθηκεύονται στους server με κυκλικό τρόπο, μέχρι να αποθηκευτεί ολόκληρο το αρχείο. Παραδείγματα αυτής της τεχνικής φαίνονται στην εικόνα 0.2.

0.4 DAPHNE

Το DAPHNE είναι ένα project που αναπτύχθηκε για να διευκολύνει την εργασία με Ολοκληρωμένες Ροές Ανάλυσης Δεδομένων. Έχει αναπτυχθεί σε C++. Ο χρήστης γράφει κώδικα σε μια γλώσσα ειδικού τομέα, την DaphneDSL. Στην συνέχεια, ο κώδικας μεταγλωττίζεται με μια αλυσίδα πολλαπλών επιπέδων. Ο σχεδιασμός αυτός παρέχει ευελιξία και επεκτασιμότητα. Το DAPHNE χρησιμοποιεί πυρήνες, οι οποίοι είναι υλοποίηση λειτουργικότητας σε C++, για να εκτελέσει μια συγκεκριμένη λειτουργία. Μέσω τέτοιων πυρήνων είναι δυνατή η διαλειτουργικότητα του DAPHNE με το Lustre. Το DAPHNE μπορεί να εκτελεστεί τόσο τοπικά όσο και κατανεμημένα. Κατά την κατανεμημένη εκτέλεση, ο μεταγλωττιστής αποφασίζει ποιος κώδικα θα εκτελεστεί από τον κάθε κόμβο. Ο

κόμβος-συντονιστής αναλαμβάνει να διαμοιράσει τα δεδομένα καθώς και το κομμάτι κώδικα στους κόμβους.

0.5 Υλοποίηση

Η διασύνδεση του DAPHNE με το Lustre γίνεται μέσω πυρήνων υλοποιημένων σε C++. Οι πυρήνες που αναπτύξαμε είναι οι εξής:

1. Πυρήνας για διάβασμα CSV αρχείων από το Lustre
2. Πυρήνας για γράψιμο CSV αρχείων στο Lustre
3. Πυρήνας για διάβασμα dbdf αρχείων από το Lustre
4. Πυρήνας για γράψιμο dbdf αρχείων στο Lustre

Επειδή το Lustre είναι συμβατό με το POSIX, οι κλήσεις συστήματος `read`, `write`, και `open` μπορούν να χρησιμοποιηθούν για τις περισσότερες λειτουργίες. Παρόλα αυτά χρειαζόμαστε κάποια βιβλιοθήκη του Lustre για να προσδιορίσουμε το layout του αρχείου κατά την δημιουργία του. Για τον λόγο αυτό χρησιμοποιήθηκε η βιβλιοθήκη που περιέχεται στον πηγαίο κώδικα του Lustre, `liblustreapi`.

Στο πλαίσιο της εργασίας αυτής επεκτείναμε docker image του DAPHNE, προσθέτοντας τις απαραίτητες βιβλιοθήκες ώστε να υποστηρίζει την σύνδεση με το Lustre. Έτσι, ένα docker container μπορεί να εκτελεστεί σε κόμβο που ήδη έχει κάνει mount το σύστημα αρχείων Lustre και να έχει πρόσβαση στο εκάστοτε σύστημα αρχείων.

Η παρούσα υλοποίηση χρησιμοποιεί ένα κοινό αρχείο για όλους τους κόμβους του κατανεμημένου χρόνου εκτέλεσης. Για να είναι αυτό εφικτό, πρέπει να διασφαλιστεί ότι ο κάθε κόμβος διαχειρίζεται διαφορετικό κομμάτι του αρχείου. Αυτό σημαίνει πως χρειάζεται κάποιου είδους αντιστοίχιση μεταξύ του index κάποια γραμμής του πίνακα και της θέσης στο αρχείο που της αντιστοιχεί. Στην περίπτωση των CSV αρχείων, η κάθε τιμή καταλαμβάνει ένα μη προκαθορισμένο εύρος χαρακτήρων. Για την επίλυση του προβλήματος αυτού, προσθέτουμε κενούς χαρακτήρες ώστε να επιτευχθεί ένα προκαθορισμένο εύρος χαρακτήρων. Στην περίπτωση των dbdf αρχείων κάτι τέτοιο δεν είναι αναγκαίο, καθώς αυτή η μορφή αρχείων χρησιμοποιεί την διαδική αναπαράσταση των δεδομένων η οποία καταλαμβάνει προκαθορισμένο εύρος bytes ανάλογα των τύπο δεδομένων.

0.6 Πειράματα

Πραγματοποιήσαμε τα εξής πειράματα για να εξετάσουμε κατά πόσο η διασύνδεση του DAPHNE με το Lustre είναι επικερδής, αλλά και για να εξετάσουμε την επιρροή διαφορετικών παραμέτρων του Lustre στην επίδοση:

1. Σύγκριση υπάρχουσας υλοποίησης με την υλοποίηση διασύνδεσης με το Lustre
2. Σύγκριση διαφορετικών τιμών της παραμέτρου stripe size
3. Σύγκριση υλοποίησης που χρησιμοποιεί την εντολή `ftruncate` για να αλλάξει το μέγεθος του αρχείου στο επιθυμητό, με υλοποίηση που δεν την χρησιμοποιεί
4. Σύγκριση υλοποίησης που ευθυγραμμίζει τις κλήσεις συστήματος με υλοποίηση που δεν τις ευθυγραμμίζει

Τα πειράματα πραγματοποιήθηκαν σε cluster στο AWS. Χρησιμοποιήθηκαν συνολικά 9 κόμβοι, εκ των οποίων ο ένας ήταν συντονιστής και οι υπόλοιποι 8 workers.

0.7 Αποτελέσματα

Τα πειράματα που προαναφέρθηκαν δείχνουν ότι η διασύνδεση με το Lustre είναι επικερδής, καθώς ο χρόνος εκτέλεσης είναι συστηματικά μικρότερος σε σύγκριση με την υπάρχουσα υλοποίηση που δεν χρησιμοποιεί κάποιο κατανεμημένο σύστημα αρχείων. Ακόμα, είναι εμφανής η επεκτασιμότητα (scalability) του συστήματος καθώς η ύπαρξη περισσότερων κόμβων οδηγεί σε αύξηση της απόδοσης. Τέλος, φαίνεται πως η διαφοροποίηση των παραμέτρων σχετικών με το Lustre, όπως το stripe size, η αλλαγή του μεγέθους του αρχείου πριν τις εγγραφές και η ευθυγράμμιση των κλήσεων συστήματος, δεν επηρεάζουν τα αποτελέσματα. Αυτό μπορεί να οφείλεται σε διάφορες παραμέτρους, όπως το buffering σε επίπεδο λειτουργικού συστήματος ή την page cache. Είναι επίσης πιθανό ότι υπάρχει κάποιο διαφορετικό σημείο καμπής, όπως η περιορισμένες δυνατότητες των κόμβων ή του δικτύου.

Κείμενο στα αγγλικά

Chapter 1

Introduction

In this chapter, we present some foundational knowledge about Distributed File Systems (DFSs) and Integrated Data Analytics (IDA) pipelines. A more detailed overview of the specific technologies used in this work (namely Lustre File System and DAPHNE) can be found in Chapter 2.

1.1 Distributed File Systems

1.1.1 Overview

DFSs are a foundational component of modern computing environments, enabling efficient storage, retrieval, and management of data across a network of interconnected computers. In their most basic form, they emulate the functionality of a non-distributed file system for client programs running on multiple remote computers [6]. Apart from that, they can provide advanced features, such as file replication, and also increased performance, such as increased bandwidth.

From a developer's perspective, the great power of the file system interface lies in the fact that applications do not need to be modified in order to use distributed storage [3]. This means that no additional care needs to be taken in order for the nodes of a system to share file data. In comparison, without a DFS, the developer of a distributed application needs to implement a way to transfer file data between nodes. This can be both cumbersome and less performant.

1.1.2 Challenges

When designing distributed systems in general, multiple challenges need to be addressed. In particular, additional caution must be taken when dealing with DFSs. Some of these challenges are the following [27, 6, 41, 34]:

1. Naming and Transparency

   Naming is essentially a mapping between logical (usually represented by file names) and physical objects. In a traditional file system, a file is mapped to a range in the disk. In a DFS, an extra abstraction level is introduced, as the client needs not be aware of where the actual file resides. This allows file replication, where the file name is mapped to a list of the locations of replicas.

2. Semantics of Sharing

   The Semantics of Sharing characterize the behavior of the DFS when multiple clients access the same shared file simultaneously. These semantics are responsible for specifying when the modifications of data by a client are observable by other clients, if at all. In practice, there are different approaches to tackling this issue. For example, in UNIX semantics, writes to an open file are immediately visible by all clients that have this file opened. It follows that every read operation sees the effect of all previous writes performed on that file. Some other common strategies are Session Semantics, Transaction-Like Semantics, and Immutable Shared File Semantics.

3. Remote Access Methods

   A decision needs to be made between accessing the remote server for each operation or having the client cache a file. There are advantages in both approaches. When using client caches, operations can be performed faster, as no network traffic is required in the majority of cases. However, the problem of keeping caches up to date arises, as well as the loss of data when a client crashes. On the other hand, requiring the client to communicate with the server for each operation is simpler but slower as it creates network traffic for each operation. Designing a caching scheme requires additional decisions, like the cache unit size, cache location, modification policy, and cache validation.

4. Fault Tolerance

   Due to the multiple components of DFSs, they are more prone to errors than traditional file systems. Deciding whether connections between servers and clients are stateless or stateful impacts fault tolerance. With stateless connections, servers can seamlessly be replaced, as each request is self-contained. On the other hand, when the connections are stateful, in case of failure all volatile data is lost. This comes at a performance cost though, since stateless request messages are longer and processing them is generally slower, since no prior session information (for example file descriptors) can be used to speed it up.

   Another important aspect is file availability. Increased availability can often be achieved by file replication, where different replicas of the same file reside on failure-independent machines. DFSs choosing this approach need to implement mechanisms to decide the replication degree and the placement of replicas, as well as maintain the consistency of replicas.

5. Scalability

   An essential need of DFSs is scalability. DFSs should be scalable, since they are often used with very large amounts of data. They should be able to scale horizontally or vertically as data volume and user demands increase.

   Horizontal scaling is achieved by adding more nodes. It is a relatively low-cost solution, which can increase fault tolerance due to replication and redundancy between nodes. The performance is increased linearly with the number of nodes, allowing the system to gradually scale. For those reasons, it is best suited for scenarios requiring flexible capacity and performance, such as cloud storage services.

   Vertical scaling is achieved by enhancing the resources of a single node (e.g., CPU, memory). It is a more expensive solution, with scalability limited by the physical architecture and performance of individual components. The performance impact is generally lower compared to horizontal scaling, especially when servers are operating close to their limits. Also, it is not helpful regarding fault tolerance. For those reasons, this solution is mostly used together with horizontal scaling,

1.1.3 Taxonomy

There are multiple ways to categorize DFS, depending on the criterion used. We briefly present some of those criteria, as described in [42] and [34]:

1. Architecture

   The oldest architecture is the Client-Server Architecture, which allows clients to access the files stored on a server. More modern approaches use a Cluster-Based Architecture, where a single master controls multiple chunk servers. Another way to differentiate the implementations is whether they use Symmetric or Asymmetric Architectures. The former is based on peer-to-peer technology, whilst the latter utilizes one or more dedicated metadata managers. Finally, in

the Parallel Architecture data blocks are striped in parallel across multiple storage devices on multiple storage servers. This allows all the workers of parallel applications to access the same file simultaneously.

2. Processes

   The main consideration regarding processes is whether they are stateful or stateless. There are advantages to each approach, as described earlier.

3. Communication

   Mostly, the Remote Procedure Call (RPC) method- possibly with some enhancements to support special cases- is used. This allows the DFS to be independent of the underlying operating system and network. One alternative is to provide a Network Abstraction Layer, which supports heterogeneous networks.

4. Naming

   As discussed earlier, the DFS must provide a mapping of the file system abstraction on to physical storage media. Here we also consider whether there is a central metadata server, or metadata is distributed across all nodes.

5. Synchronization

   This includes the Semantics of File Sharing, which we already discussed, and the File Locking System. The two topics are coupled together, since different file sharing semantics need different locking schemes.

6. Consistency and Replication

   When sending data through a communication network, validation can be achieved via checksums. This provides consistency. Replication is linked with caching, which was discussed earlier. It is worth pointing out that both metadata and data objects can be replicated.

7. Fault Tolerance

   Apart from the fault tolerance considerations we already examined, a distinction is made on how the DFS handles inevitable faults. The two paradigms are ”failure as exception” and ”failure as norm”. When choosing the ”failure as exception” approach, the system isolates the failure node or recovers from the last normal running state. This is the case with Lustre, since it is assumed data is available as long as the physical device is healthy, thus treating the physical failure of the device as an exception. On the other hand, ”failure as norm” systems utilize replication, which is triggered whenever the replication ratio becomes unsafe.

8. Security

   Most DFSs employ security with authentication, authorization and privacy by leveraging existing security systems. However, it is also possible to assume trust between all nodes and clients and not deploy any dedicated security mechanism (GFS and Hadoop being two examples).

We have included the table 1.1 from [42]. It is interesting to see how different implementations tackle each challenge. For example, since we are mostly interested in the Lustre file system, we observe that it is the only one that provides a network abstraction layer, making it network-independent.

1.1.4 Components

As we have seen, there are many different implementations of DFS. This makes it difficult to find a common scheme to describe them. However, among most of the implementations, the following components are present:

File System	GFS	KFS	Hadoop	Lustre	Panasas	PVFS2	RGFS
Architecture	Clustered-based, asymmetric, parallel, object-based	Clustered-based, asymmetric, parallel, object-based	Clustered-based, asymmetric, parallel, object-based	Clustered-based, asymmetric, parallel, object-based	Clustered-based, asymmetric, parallel, object-based	Clustered-based, symmetric, parallel, aggregation-based	Clustered-based, asymmetric, parallel, block-based
Processes	Stateful	Stateful	Stateful	Stateful	Stateful	Stateless	Stateful
Communication	RPC/TCP	RPC/TCP	RPC/TCP	Network Independence	RPC/TCP	RPC/TCP	RPC/TCP
Naming	Central metadata server	Central metadata server	Central metadata server	Central metadata server	Central metadata server	Metadata distributed in all nodes	Metadata distributed in all nodes
Synchronization	Write-once-read-many, give locks on objects to clients, using leases	Write-once-read-many, give locks on objects to clients, using leases	Write-once-read-many, give locks on objects to clients, using leases	Hybrid locking mechanism, using leases	Give locks on objects to clients	No locking method, no leases	Give locks on objects to clients
Consistency and Replication	Server-side replication, asynchronous replication, checksum	Server-side replication, asynchronous replication, checksum	Server-side replication, asynchronous replication, checksum	Server-side replication, only metadata replication, client-side caching, checksum	Server-side replication, only metadata replication	No replication, relaxed semantic for consistency	No replication
Fault Tolerance	Failure as norm	Failure as norm	Failure as norm	Failure as exception	Failure as exception	Failure as exception	Failure as exception
Security	No dedicated security mechanism	No dedicated security mechanism	No dedicated security mechanism	Security in the form of authentication, authorization, and privacy	Security in the form of authentication, authorization, and privacy	Security in the form of authentication, authorization, and privacy	Security in the form of authentication, authorization, and privacy

Table 1.1: Overall Comparison of Different Distributed File Systems [42]

1. NameNode

   The central node, responsible for maintaining namespace and metadata information. It receives file operation requests from clients and forwards them to the appropriate DataNodes.

2. DataNodes

   DataNodes are responsible for storing the actual file data and handling requests from clients.

3. Clients

   Clients offer the user interface of the DFS. They communicate requests to the NameNode, such as file creation, deletion, renaming, and locating. Additionally, they can interact directly with the DataNodes for read and write operations, using the information provided by the NameNode.

4. Metadata Storage

   Metadata storage is responsible for persistent metadata storage, which is essential for recovery in case of failure. It can be either a single server or a cluster. The NameNode must ensure that metadata is consistent between this persistent storage and itself.

1.2 IDA Pipelines

1.2.1 Overview

IDA pipelines are complex workflows that combine various processes related to data management and analysis. These pipelines typically consist of multiple phases including, but not limited to:

1. ETL (Extract, Transform, Load): This phase involves gathering data from different sources, processing it to a suitable format, and loading it into a target system for analysis.

2. Machine Learning (ML) Training and Scoring: In this phase, machine learning models are trained using the prepared data, and the trained models are then used to make predictions or score new data.

3. Numerical Computation and Simulations: IDA pipelines may also involve performing complex mathematical computations or simulations to interpret the data further.

4. Data Analysis: This final phase is where actual data insights are derived, often involving querying database systems or applying statistical methods.

The main characteristics of IDA pipelines are the orchestration of these processes, handling large and heterogeneous data collections, and the integration of various technologies (e.g., database systems, machine learning frameworks, and computational tools) to streamline analysis and enhance results. The goal is to create efficient, scalable, and manageable workflows that facilitate robust data analysis across different domains, such as manufacturing, healthcare, finance, and more.

1.2.2 Challenges

Some of the challenges encountered when designing IDA pipelines are:

1. Integration Complexity: Developing and deploying IDA pipelines is a cumbersome process that involves integrating various systems, programming paradigms, resource managers, and data representations, leading to significant complexity.

2. Compatibility Across Systems: There is a lack of a seamless infrastructure that allows different systems (i.e., data management, high-performance computing, and machine learning) to work together efficiently, causing potential compatibility issues.

3. Resource Management: Static resource allocation can lead to temporal and spatial under-utilization of resources, which harms overall efficiency and effectiveness.

4. Overhead from Boundary Crossing: Utilizing specialized systems for orchestration can introduce overhead due to the need to materialize intermediate results and handle boundary crossings between different system components.

5. Diverse Language Abstractions: The varying programming models and language abstractions used in data management, HPC, and ML systems make it difficult to develop a unified approach for IDA pipelines.

6. Hardware Challenges: The systems involved are affected by hardware limitations such as the end of Dennard scaling and Moore's law, which contribute to dark silicon and the need for increasing specialization at device, storage, and workload levels.

7. Dynamic Workflow Management: The complexity of handling various workflows that include data extraction, pre-processing, ML training, and query processing complicates the design of IDA pipelines.

8. Need for High-level APIs: There is a requirement for seamless high-level APIs and domain-specific languages to effectively operate across data management, HPC, and ML .

1.2.3 Examples

Earth Observation

The Earth Observation project focuses on classifying Local Climate Zones (LCZs) to model climate-relevant surface properties using satellite data from the Sentinel-1 radar and Sentinel-2 optical images, part of the Copernicus initiative. The project generates about 4 petabytes of global data annually. To train LCZ classifiers, the DLR team created the So2Sat LCZ42 dataset, which includes 400,673 pairs of 32x32 image patches and LCZ labels. These labels were hand-annotated by 15 experts and verified, achieving 85% confidence. The dataset for training, testing, and validation totals approximately 55.1 GB in HDF5 format. The training pipeline involves pre-processing Sentinel-2 data and training a ResNet20 classifier. However, the primary challenge is efficiently implementing the scoring pipeline at a petabyte scale, which includes managing complex data storage, preprocessing, quantization, and conducting analyses related to urban change.

Semiconductor Manufacturing

In the Semiconductor Manufacturing process, ion implantation is used to alter the physical, chemical, and electrical properties of silicon wafers by accelerating dopants onto them using specialized equipment. To ensure successful tuning of ion beams after any recipe change, a prediction model is employed to estimate tuning success, minimizing costly timeouts that occur after 15 minutes of unsuccessful attempts. Data preparation involves scanning and parsing raw implant log files, which are then stored in a multi-table database. This data is joined and exported into CSV files containing 79 categorical and 2,468 numerical features. The model training process includes preprocessing steps such as eliminating low-variance and highly-correlated features, splitting data into training and testing sets, imputing missing values, one-hot encoding, and normalization. A random forest classifier is trained with extensive hyper-parameter tuning and cross-validation, evaluated through metrics like F1 measure, AUC, and correlation measures.

Material Degradation

In the Material Degradation study, accelerated stress tests are conducted to assess the degradation of power semiconductors. Multiple devices are tested simultaneously, with the resulting voltage and current measurements recorded as waveforms, and millions of electrical signals are stored in TDMS files. To analyze degradation, physics-based models simulate the microscopic deterioration of thin metal layers and estimate their lifetime under various mechanical stresses, which informs reliability analysis and degradation modeling. The analysis pipeline involves reading TDMS files, extracting relevant data, computing power waveforms, simplifying waveforms for data reduction, and ingesting the results into a database via a REST API. Subsequently, finite element method (FEM) simulations are performed on a separate high-performance computing (HPC) cluster, with the simulation results stored back in the database.

Vehicle Development

In the Vehicle Development project, two primary use cases are examined.
The first involves optimizing ejector geometry for PEM and SOFC fuel cells, where hydrogen flow is supplied from a tank, mixed in a chamber, and adjusted based on geometric variables and operating conditions to achieve a specific entrainment ratio and suction pressure. This optimization process iteratively predicts design variants using a behavioral model and conducts computational fluid dynamics (CFD) simulations, revising the design until the desired properties are achieved while minimizing the costs associated with CFD simulations.

The second use case focuses on predicting key performance indicators (KPIs) such as fuel consumption, vehicle mass, and aerodynamic drag. These predictions are validated through simulations and hardware-in-the-loop tests. The analysis pipeline collects data in JSON format and utilizes Gaussian process regression (GPR) models to create prediction models that account for individual KPIs and their interactions. However, a significant challenge lies in integrating the diverse measurements and simulation data throughout the development process.

Chapter 2

Background

This section examines the Lustre File System and the Daphne Runtime.

2.1 The Lustre File System

2.1.1 Introduction

Lustre[1] is a high-performance, POSIX-compliant, open-source parallel distributed file system designed primarily for large-scale computing environments. It is implemented entirely in the kernel and its architecture is founded upon distributed object-based storage. It is widely used for applications requiring extensive data processing, such as scientific research, AI, and big data analytics. Lustre is renowned for its scalability, with the ability to handle petabytes of data and thousands of clients, making it suitable for high-performance computing (HPC) clusters and supercomputing environments.
It is being used at 7 out of the top 10 fastest computers in the world today, over 70% of the top 100, and also for over 60% of the top 500 [16]. One more testament to Lustre's popularity is that International Data Corporation (IDC) shows it as being the file system with the largest market share in HPC [19]
Table 2.1 demonstrates Lustre's scalability capabilities.

The features that make Lustre such a popular choice are:

1. POSIX Compliance

   With few exceptions, Lustre passes the full POSIX test suite. Most operations are atomic to ensure that clients do not see stale data or metadata. Lustre also supports mmap() file IO.

[1] https://www.lustre.org/

Feature	Current Practical Range	Known Production Usage
Client Performance	100 - 100,000	50,000+ clients, many in the 10,000 to 20,000 range
OSS Scalability	Single client: 90% of network bandwidth Aggregate: 10 TB/s	Single client: 4.5 GB/s (FDR IB, OPA1), 1000 metadata ops/sec Aggregate: 2.5 TB/s
OSS Scalability	Single OSS: 1-32 OSTs per OSS Single OST (ldiskfs): 300M objects, 256TiB per OST Single OST (ZFS): 500M objects, 256TiB per OST OSS count: 1000 OSSs w/ up to 4000 OSTs	Single OSS (ldiskfs): 32x8TiB OSTs per OSS, 8x32TiB OSTs per OSS Single OSS (ZFS): 1x72TiB OST per OSS OSS count: 450 OSSs w/ 1000 4TiB OSTs, 192 OSSs w/ 1344 8TiB OSTs, 768 OSSs w/ 768 72TiB OSTs
OSS Performance	Single OSS: 15 GB/s Aggregate: 10 TB/s	Single OSS: 10 GB/s Aggregate: 2.5 TB/s
MDS Scalability	Single MDS: 1-4 MDTs per MDS Single MDT (ldiskfs): 4 billion files, 8 TiB per MDT Single MDT (ZFS): 64 billion files, 64 TiB per MDT MDS count: 256 MDSs w/ up to 265 MDTs	Single MDS: 3 billion files MDS count: 7 MDSs w/ 7x 2 TiB MDTs in production (256 MDSs w/ 256 64 GiB MDTs in testing)
MDS Performance	50,000 create ops/sec, 200,000 metadata stat ops/sec	15,000 create ops/sec, 50,000 metadata stat ops/sec
File system Scalability	Single File (max size): 32 PiB (ldiskfs) or 2^{63} bytes (ZFS) Aggregate: 512 PiB total capacity, 1 trillion files	Single file (max size): multi-TiB Aggregate: 55 PiB capacity, 8 billion files

Table 2.1: Lustre scalability and performance numbers [16].

2. Online file system checking

   Lustre provides a file system checker (LFSCK) to detect and correct file system inconsistencies. LFSCK can be run while the file system in online and in production, minimizing potential downtime.

3. Controlled file layouts

   The file layouts that determine how data is placed across the Lustre servers can be customized on a per-file basis. This allows users to optimize the layout to best fit their specific use case.

4. Support for multiple backend file systems

   When formatting a Lustre file system, the underlying storage can be formatted as either ldiskfs (a performance-enhanced version of ext4) or ZFS.

5. Support for high-performance and heterogeneous networks

   Lustre can utilize RDMA over low latency networks such as Infiniband or Intel OmniPath in addition to supporting TCP over commodity networks. The Lustre networking layer provides the ability to route traffic between multiple networks making it feasible to run a single site-wide Lustre file system.

6. High-availability

   Lustre supports active/active failover of storage resources and multiple mount protection (MMP) to guard against errors that may result from mounting the storage simultaneously on multiple servers. High-availability software such as Pacemaker/Corosync can be used to provide automatic failover capabilities.

7. Security features

   Lustre follows the normal UNIX file system security model enhanced with POSIX ACLs. The root squash feature limits the ability of Lustre clients to perform privileged operations. Lustre also supports the configuration of Shared-Secret Key (SSK) security.

8. Capacity growth

   File system capacity can be increased by adding additional storage for data and metadata while the file system is online.

2.1.2 Architecture

The Lustre file system is built on a client-server architecture, split into specialized components that manage metadata and storage for high efficiency and scalability. The key components are:

1. Management Server (MGS)

   Provides configuration information for the file system. Clients contact the MGS to retrieve details on how the file system is configured when mounting it. The MGS is shared among all Lustre clients.

2. Management Target (MGT)

   Block device used by the MGS to persistently store Lustre file system configuration information.

3. Metadata Server (MDS)

   Manages the file system namespace and provides metadata operations (e.g., directory structure, permissions). The file system will contain at least one MDS. For small file systems, the MGS and MDS may be combined into a single server and the MGT may coexist on the same block device as the primary MDT.

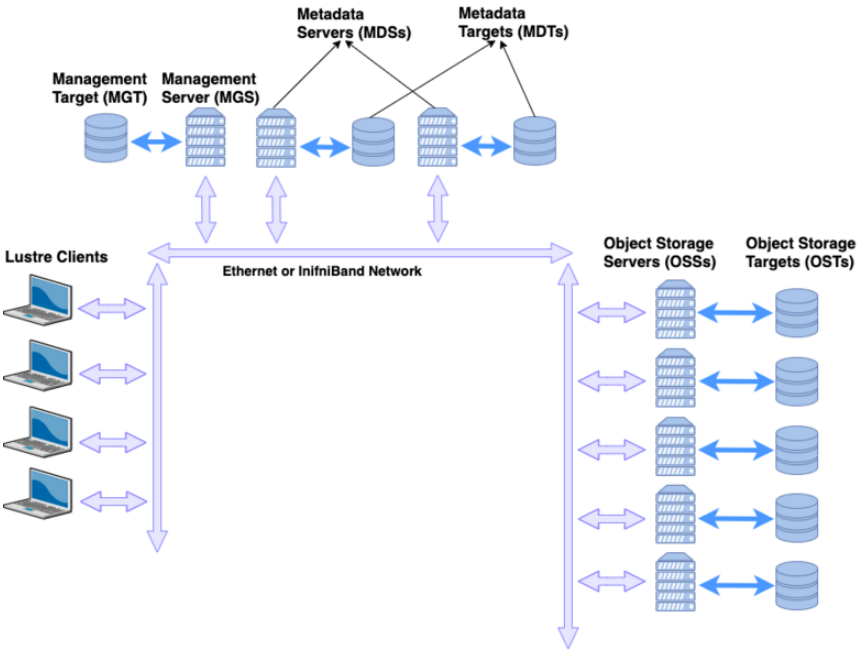


Figure 2.1: Lustre architecture [16].

4. Metadata Target (MDT)

   Block device used by an MDS to store metadata information. A Lustre file system will contain at least one MDT which holds the root of the file system. Common configurations will use one MDT per MDS/

5. Object Storage Server (OSS)

   Handles the actual file data, storing it across one or more Object Storage Targets (OSTs). A file system will typically have many OSSs.

6. Object Storage Target (OST)

   Block device used by an OSS node to store the contents of user files. An OSS node will often host several OSTs. The total capacity of the file system is the sum of all the individual OST capacities.

7. Clients

   Endpoints that mount the Lustre file system, appearing as a POSIX-compliant mount point on the client OS.

8. Lustre Networking (LNet)

   Network protocol used for communication between Lustre clients and servers. Supports RDMA on low-latency networks and routing between heterogeneous networks.

2.1.3 Striping

Lustre stores file data by splitting the file contents into chunks and then storing those chunks across the storage targets. By spreading the file across multiple targets, the file size can exceed the capacity of any one storage target [16]. It also allows clients to access parts of the file from multiple Lustre servers simultaneously, effectively scaling up the bandwidth of the file system. Users have the ability to control many aspects of the file's layout by means of the `lfs setstripe` command, and they can query the layout for an existing file using the `lfs getstripe` command.

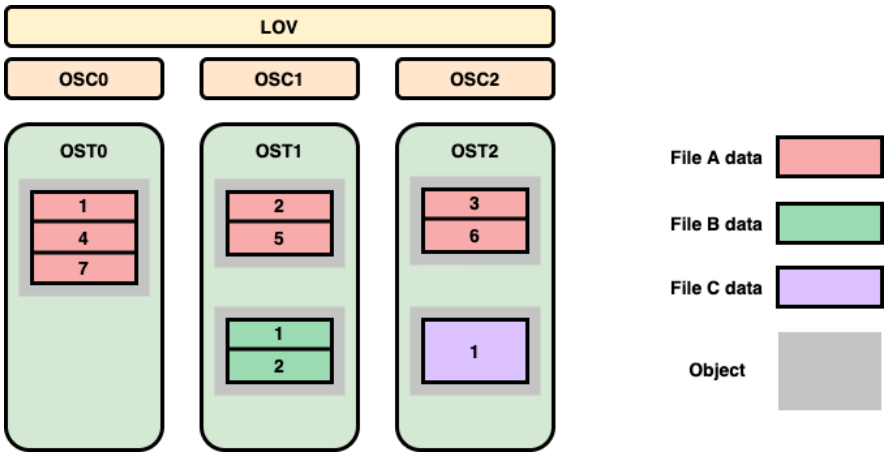


Figure 2.2: Normal RAID0 file striping in Lustre [16].

File layouts fall into one of two categories:

1. Normal / RAID0

   File data is striped across multiple OSTs in a round-robin manner. The stripe count determines how many OSTs will be used to store the file data, while the stripe size determines how much data will be written to an OST before moving to the next OST in the layout. The following example from [16] is depicted on Figure 2.2. File A has a stripe count of three, so it will utilize all OSTs in the file system. Assume that the stripe size is set to the default value of 1 MB. When file A is written, the first 1 MB chunk gets written to OST0. Lustre then writes the second 1 MB chunk of the file to OST1 and the third chunk to OST2. When the file exceeds 3 MB in size, Lustre will round-robin back to the first allocated OST and write the fourth 1MB chunk to OST0, followed by OST1, etc. Files B and C show layouts with the default Lustre stripe count of one, with stripe sizes of 1 MB and 2 MB respectively.

2. Composite

   Complex layouts that involve several components, each with potentially different striping patterns. The most basic version of this is a Progressive File Layout (PFL), which can be viewed as an array of normal layouts, associated with a start and end point. Then, for each of the non-overlapping regions of the file, the corresponding file layout is used. Lustre also offers extensions of this concept, the Data on MDT and Self Extending Layout.

2.2 DAPHNE

In this chapter, we present an overview of the DAPHNE project, based on [9, 43, 21]. We then cover in more detail some key aspects that are relevant to our work.

2.2.1 Architecture

Figure 2.3 depicts the DAPHNE system architecture.
DAPHNE is built from scratch in C++, but utilizes MLIR [2] as a multi-level, LLVM-based intermediate representation (IR) as well as existing runtime libraries such as BLAS, LAPACK, DNN kernels, and collective operations. These libraries are augmented with more specialized, custom kernel implementations. Users specify their IDA pipelines in DaphneDSL (a language similar to Julia, PyTorch, or R) or DaphneLib (a Python API with lazy evaluation that creates DaphneDSL well). These DSL

[2] https://mlir.llvm.org/

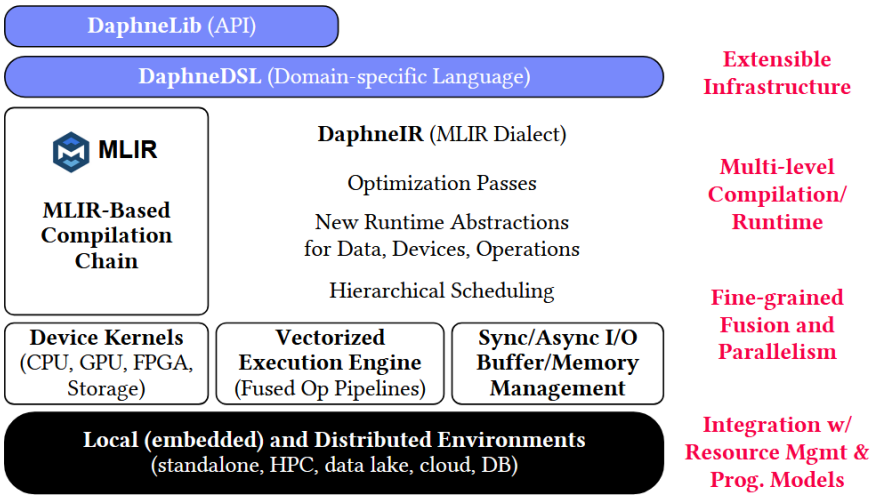


Figure 2.3: DAPHNE architecture [9].

programs are then compiled—via a multi-level compilation chain—into executable runtime plans. An example of such a DaphneDSL program can be seen in Figure A.1.
This design allows for the registration of new data types, kernels, and scheduling algorithms, making the infrastructure highly extensible. Sideway entries into the multi-level compilation chain are also allowed, as a way of enforcing certain physical data types and kernels.
Via the compilation chain, DaphneDSL is converted into DaphneIR (an MLIR-based intermediate representation) by an ANTLR parser. Subsequently, multiple optimization passes are performed.
Finally, during runtime, the kernels are executed sequentially and produce materialized intermediates in memory.

2.2.2 Building and Running DAPHNE

DAPHNE is developed using C++ and the CMAKE [3] build system is used to build the DAPHNE targets. A highly configurable build script, `build.sh` is provided, which is able to download third-party dependencies and build the DAPHNE targets. DAPHNE can be executed both natively and in a containerized environment. For both use cases, users and developers can choose to build the DAPHNE targets/container images from the source code or download the prebuilt packages provided. Detailed instructions on building and running DAPHNE can be found in the project's repository [4].

2.2.3 Runtime Overview

Kernels are the actual code that is executed on a device's hardware, such as a CPU or GPU, and is used to perform a specific operation [43]. It is through such kernels, developed in C++, that the integration of DAPHNE with Lustre is implemented. A full list of supported kernels can be found in the DAPHNE repository [5].

For completeness, we include the definition of a Kernel, as seen in [9]:

Definition 2.2.1 (Kernel)**.** A kernel is an implementation of an IR operation (or registered user-defined kernel) that operates on instantiated and materialized data types. Most kernels are stateless (except memory allocation) and deterministic. Stateful kernels are allowed as well (e.g., implementing configuration management and setup/tear-down of context objects for device/cluster initialization and cleanup).

[3] https://cmake.org/
[4] https://github.com/daphne-eu/daphne/blob/main/doc/GettingStarted.md
[5] https://github.com/daphne-eu/daphne/blob/main/src/runtime/local/kernels/kernels.json

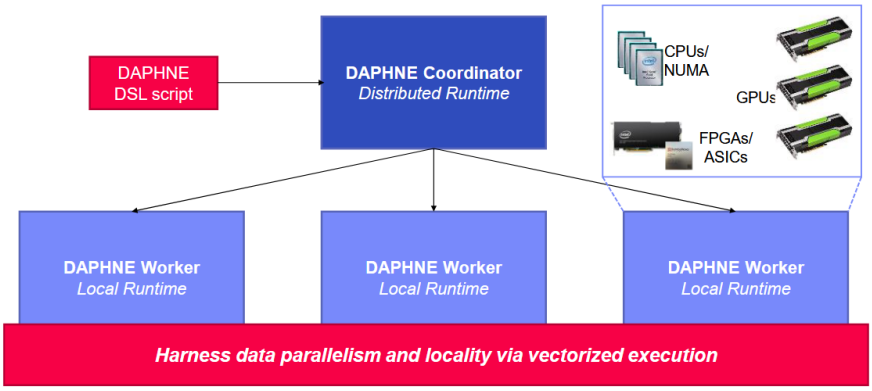


Figure 2.4: DAPHNE Runtime hierarchical approach [43]

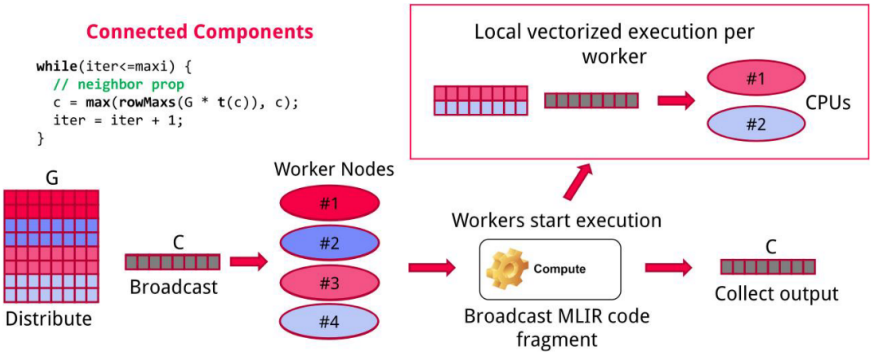


Figure 2.5: Example of hierarchical and vectorized execution for the Connected Components algorithm [43].

Context objects are used to access distributed runtimes and hardware accelerators. Information is encapsulated in such a context object, which in turn is passed to individual kernels. The initializers of specific devices or frameworks are local kernels themselves that add state to the global context. This approach simplifies the integration of new accelerators by registering such kernels in the extension catalog. The compiler produces an execution plan with calls to C++ host kernels for local, distributed, or accelerator operations. Kernels make heavy use of C++ templates for both value types and combinations of dense and sparse inputs. Since hundreds of operations that require specializations are supported, the template instantiations are automatically generated. For n-ary operations with mixed types, the compiler injects casts, some of which (e.g., casting an FP32 frame-column to an FP32 matrix) are no-ops.

Local and Distributed Runtime

The local runtime is responsible for the execution of complex pipelines in a single compute node. This node could consist of multiple heterogeneous resources, e.g., multicore CPUs, GPUs, FPGAs, and computational storage devices.

The distributed runtime system coordinates the distribution of work among worker nodes and collects the results. The compiler decides which code should be executed by each worker, and sends it in the form of an MLIR snippet. The coordinator uses distribution primitives (such as broadcast, all-reduce, ring-reduce, scatter/gather, etc.) to distribute data and the aforementioned MLIR snippet to worker nodes. Finally, each worker locally compiles and executes the generated code through the local runtime system via the vectorized execution engine. An example of this procedure is depicted in Figure 2.5.

Data Representations

DAPHNE's basic data types are frames, matrices, and scalars. Each matrix, scalar or frame-column has a value type. At DSL level, users deal with these abstract data types, and the compiler systematically lowers operations to kernels that produce local or distributed physical data structures that are the inputs/outputs of kernels.

Communication Backends

When running distributed, a communication framework is needed to allow different nodes to exchange information. The distributed runtime implementation is decoupled from the communication framework that is used. This makes it easy to extend DAPHNE with any communication framework. Currently, MPI (Message Passing Interface) and gRPC (synchronous and asynchronous) are supported. The integration with both of these frameworks allows the users to choose which one they prefer, based on their needs.

MPI is the ideal choice for complex, high-performance computing tasks. However, deploying MPI can be more challenging compared to traditional communication mechanisms, such as gRPC, as users may encounter hurdles during the installation and configuration of MPI libraries, which can vary depending on the specific computing environment and system configurations.
Usage of gRPC is more straightforward compared to MPI, both for users and developers. It offers a more intuitive and linear approach to handling requests and responses, and is responsible for handling resources for requests and responses making it behave in a more reliable and performant manner.

Supported I/O formats

DAPHNE supports I/O operations from multiple commonly used data formats. This includes CSV, Arrow, and Matrix market.
Additionally, DAPHNE defines its own binary representation for the serialization of in-memory data objects (matrices/frames). This representation is intended to be used by default whenever we need to transfer or persistently store these in-memory objects, e.g., for:

- The data transfer in the distributed runtime
- A custom binary file format
- The eviction of in-memory data to secondary storage

More details about the DAPHNE binary data format (dbdf), including the specific mappings of bytes, can be found in the project's repository [10].

Chapter 3

Implementation

3.1 Shared file and file per process comparison

In this section, we examine the two common approaches to handling multiple processes that are trying to write in the same file in High-Performance Computing (HPC) systems. This is related to our work since this is a decision we also had to take. The first is the File Per Process technique, where each process writes to a separate, independent file (Figure 3.1). The second one is the Shared File approach, where each process writes to a different section of the same, shared file (Figure 3.2). When reading, the performance should generally be the same. This is because reading allows multiple processes to access the same region of a file if needed. There are arguments to be made for choosing either one, which are briefly discussed below [23].

When each process uses a separate file, locking on file level is unnecessary. In that case, I/O performance is determined by the file system's capabilities, since the shared resource among processes is the bandwidth of the server which owns the parts of the file they are accessing. This means that generally, this is the most performant solution when a relatively small number of processes is considered. However, since every process handles a different file, the number of files required is proportional to the number of processes. For large process numbers, for example, more than 10000, there is significant stress imposed on the distributed file system due to the potentially huge number of files. This is caused by the additional metadata operations and general housekeeping required by the file system.

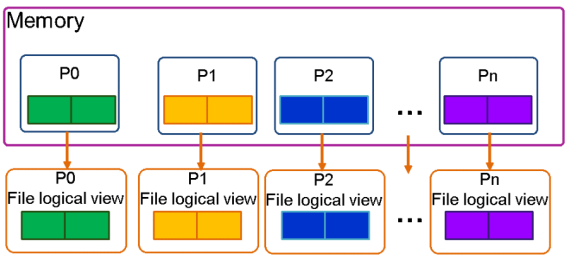


Figure 3.1: File per process approach to parallel I/O [31].

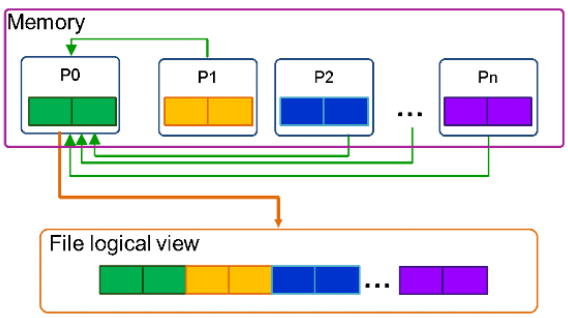


Figure 3.2: Single file, multiple writers approach to parallel I/O [31].

For this reason, this approach is not considered very scalable [31]. Additionally, it is common that the output of the task is then used as input on some external tool, which more often than not accepts a single file as input. This means that often there is the additional overhead of concatenating the partial files for the information to be usable.

The Single File approach imposes less stress on the File System since any given I/O task uses only one file. However, additional care needs to be taken to ensure that operations on the file are performed on non-overlapping regions. This means that lock contention can be a significant performance bottleneck, depending on the implementation of the file system. For example, if a write lock is imposed on file level, then this would mean that no parallelism can be achieved via this method. Suppose on the other hand that locks are imposed on server level, and I/O operations are aligned so that each of them occupies a different server. In that case, higher performance can be achieved as servers are utilized simultaneously. It is worth noting that not all file systems support appending data at a random offset of a file. HDFS is such a file system where this is not possible. In that case, the Single File approach cannot be used, since nodes would not be able to write on demand at a given position.

In this work, we use the Single File approach, due to its reduced metadata overhead and better scalability.

3.2 Developed Kernels and Supported I/O formats

Currently, we support I/O operations from CSV and dbdf files. However, due to the flexibility of the distributed runtime design combined with the POSIX-compliant nature of Lustre, it is very easy to extend this work to other formats. Once a kernel is implemented for a file format, and care is taken to map matrix rows to byte offsets, then a corresponding Lustre kernel can be developed that adds support for Lustre striping. The kernels we have developed for this work are:

1. Read kernel for CSV files
2. Write kernel for CSV files
3. Read kernel for dbdf files
4. Write kernel for dbdf files

The kernels use metadata files, identified by the `.meta` extension, to query information about the target file, such as number of rows, number of columns, data type of values. These metadata files are simple JSON-like files, which are parsed to create the distributed execution plan. An example metadata file can be seen in Figure 3.3

```
1 {"numCols":5000,"numRows":1000,"valueType":"f64"}
```

Figure 3.3: Metadata example for CSV file of a float64 matrix with 1000 rows and 5000 columns.

3.3 Padding

The current implementation of the Daphne partitioner is based on splitting the matrix by rows. This means that I/O kernels accept the starting row and the number of rows as parameters, and are expected to handle the corresponding matrix segment. Since we are proceeding with the shared file approach, it is necessary to ensure that workers operate on non-overlapping regions of the file. It is also important to support configurations with different worker counts. This means that a mapping from row index to byte offset is necessary, as the starting row for each worker depends on the worker count and can be arbitrary.

When dealing with dbdf files, calculating the offset given a row is straightforward. That is because in this format, the values are saved in their binary representation. This means that the size of each value is fixed and equal to the bytes used for its data type. In the case of CSV files however, since the string representation of the values is used, it is not possible to know deterministically its size. We solve this problem by enforcing a fixed string size for each value. We do this by padding the string with empty characters. The target string size should be big enough to ensure no information is lost, meaning we only pad the value up to the desired size and never truncate it. This method might increase the file size significantly in cases where a high percentage of values needs to be padded and the padding is big. Another solution would be to simply read newline characters until we reach the desired row index. This would not increase the size of the file, but it would be significantly slower. Finally, we could also add some information to the metadata files to help with the offset calculation. However it would be impossible to cover every possible starting index without having metadata files linear in size compared to the actual data files. So again, some combination of other methods would have to be used. Essentially, this problem and its solutions boil down to a trade-off between time and space complexity.

Using the padding method described above, it becomes straightforward to calculate the byte offset of any given row as seen in the code snippet at Figure 3.4

```
size_t rowSize = argNumCols * CHARS_PER_CSV_CELL + (argNumCols-1) * sizeof(',')
↪  + sizeof('\n');
size_t offset = startRow * rowSize;
```

Figure 3.4: Calculation of byte offset a worker should start performing I/O operations, given the starting row.

It is worth noting that the problem is not caused by our decision to use the single file approach. The same situation would arise if the file per process implementation was chosen. In that case, suppose that a matrix is written by a DAPHNE script in a configuration of 5 workers. This results in 5 segments. When a DAPHNE script, executed in a distributed environment of 6 workers attempts to read the previously generated matrix, the problem of mapping occurs again.

3.4 Usage of `pread` and `pwrite` for Concurrent File I/O

In the C++ implementation of the kernels, `pwrite` and `pread` system calls are utilized to enhance the efficiency and reliability of file I/O operations.

Unlike standard `write` and `read` functions, which rely on the current file offset, `pwrite` and `pread` explicitly specify the byte offset for each operation, enabling concurrent processes to read from and write to different regions of the same file without interfering with one another. This approach is particularly beneficial in the distributed environment, as it eliminates the need for external synchronization mechanisms, such as file locks, when managing multiple worker nodes. By directly controlling the read/write positions, `pwrite` and `pread` minimize overhead, improve parallel access performance, and ensure that data consistency is maintained when working with large datasets distributed across many OSTs.

3.5 Lustre Library

Since Lustre is a POSIX-compliant file system, the standard system calls (`open`, `read`, `write`, or `pread`, `pwrite` in our case) can be used for most of the kernel's needs. However, an essential functionality we need from the C++ library of Lustre is the way to determine the file layout when creating files. For this need, we use the built-in C API that is provided in the source code of Lustre, and is built when building the Lustre client. We simply link the library, `liblustreapi`, to our project via

CMAKE. The library provides the `llapi_file_open` method. We are using Lustre's normal layouts and the `llapi_file_open` allows us to determine the stripe size, stripe count, and starting OST parameters at file creation, as seen in 3.5.

```
int llapi_file_open(const char *name, int flags, int mode, unsigned long long
↪  stripe_size, int stripe_offset, int stripe_count, int stripe_pattern);
```

Figure 3.5: `llapi_file_open` declaration [30].

3.6 Docker

The most common way users deploy the Daphne project is via docker images. For this reason, the main way of deployment this work focuses on is providing a docker image which includes all the necessary libraries that
In order to provide instructions to build a docker image capable of interacting with the Lustre FS, we have to make the necessary changes on `build.sh`. In short, those changes are:

1. Clone the Lustre source code from the official repository
2. Configure the building process
3. Build the Lustre client modules
4. Preserve only the necessary dependencies for our use case

In order for the Docker container to have access to the Lustre file system, the following procedure is used:

1. The host OS, where the container will be deployed, mounts the file system via the client interface to some mount point, for example `/lustre`.
2. The container mounts the host Lustre directory (`/lustre` in this case) to some directory inside the container.
3. DAPHNE (and any other program) can then run on the container and have access to the Lustre file system.

3.7 Extensibility

As previously discussed, the current prototype is designed with the per-row partitioning in mind. However, the design with a fixed length per CSV value is easily extensible to support any future partitioner. For example, one design could partition elements by columns instead of rows. In that case, and with the padding already applied, it is straightforward to calculate the byte offset of any given value, thus making any partitioner design possible. The dbdf files bypass the need for padding and the same ease of use achieved via padding on the CSV case is inherently built into the format. This applies here as well, where the byte offset needed for any potential distribution can be easily calculated.

It is worth noting that the way DAPHNE partitions matrix data across nodes is independent of the way that data is stored on Lustre OSTs. Even though it makes sense for each node to store its data on the OST residing on the same node if possible, this is not necessarily the best choice. This design allows us to experiment with whatever partitioning we like, both for the DAPHNE data distribution and the Lustre striping.

3.8 Configuration

We allow the user to provide configuration parameters at runtime via either the command line or a JSON file. An example of such a configuration file can be found in the project's repository [1].

Lustre kernels support the following parameters at runtime:

- Stripe size
- Stripe count
- Starting OST

Those parameters are passed to the `llapi_file_open` method.

The parameters are used only for the files storing the matrices. For metadata files, the parameters are fixed. This is because, as seen in figure 3.3, metadata files are very small, thus striping them across multiple OSTs is not advisable as it would potentially deteriorate performance.

3.9 Kernel Execution Example

In order to provide a better understanding of the kernel and information flow when performing Lustre I/O operations via DAPHNE, as well as to provide a concrete example, we present here the procedure followed when reading a matrix from a CSV formatted file stored in the Lustre file system. We examine both the local and distributed executions.

Assume we have the simple DaphneDSL script, `example.daph`, depicted in figure 3.6, and the file `example.csv.lustre` exists along with the corresponding metadata file `example.csv.lustre.meta` under the `/lustre` directory. Also, assume that the CSV file is written using the necessary padding. This can be done either via another DAPHNE script, since the Lustre write kernels are implemented with the padding methodology, or via an external tool which takes care of padding.

```
1 // DAPHNE script that reads the contents of /lustre/example.csv.lustre into the
  ↪  matrix object A
2 A = readMatrix("/lustre/example.csv.lustre");
```

Figure 3.6: DaphneDSL script, `example.daph`, to read a matrix stored in CSV format on the Lustre file system.

Note that the convention used for naming files is a multi-level extension, where the filename is followed by the file format and then the underlying file system. Such an example is `example.csv.lustre`, where a CSV file is stored in the Lustre file system. If no DFS (HDFS or Lustre) is used, only the file format must be present. This means that the command `readMatrix("/lustre/example.csv")` would instead invoke the kernel for the local file system.

3.9.1 Local Execution

Local execution only depends the `daphne` executable, which can be built via the `build.sh` script. No additional configurational steps need to be taken, compared to what we will see in the Distributed Execution section. The `example.daph` script of figure 3.6 can be executed using the daphne executable via the command `./daphne example.daph`.

At first, the Read kernel is executed. This is the entry point for any read operation. It uses the file extension to decide which specialized kernel is to be invoked next. In this example, the `.lustre` extension hints that the Lustre kernels should handle the file. Subsequently, the Lustre kernel parses

[1] https://github.com/daphne-eu/daphne/blob/main/UserConfig.json

the file format extension to determine the way the file should be read. In that case the `.csv` extension suggests that the `readLustreCsv` method is the appropriate one. In case of dbdf files, the `readDaphneLustre` method would have been used instead.

This multi-level specialization allows developers to introduce new methods in a sideways manner. For example, to support additional formats of files stored in Lustre, a contributor would have to implement methods that operate after the Lustre kernel is involved. This eliminates the need to modify existing implementations and provides a nice level of abstraction.

3.9.2 Distributed Execution

As stated already, DAPHNE supports execution in a distributed fashion. Utilizing the DAPHNE distributed runtime does not require any changes to the DaphneDSL script, which is a major benefit of the overall design. The compiler automatically fuses operations and creates pipelines for the distributed runtime, which then uses multiple distributed nodes (workers) that work on their local data, while a main node, the coordinator, is responsible for transferring the data and code to be executed.

At the time of writing, some limitations exist on the distributed runtime. The ones that affect this example are:

1. Lustre kernels are only implemented for the the synchronous gRPC communication backend. For this reason, this section focuses on the procedure followed when the synchronous gRPC framework is used.

2. The only supported types are `DenseMatrix` types and value types `double`, meaning objects of `DenseMatrix<double>`.

The steps that need to be followed for the distributed execution of DaphneDSL scripts are the following:

1. Building the Distributed Worker

   The DAPHNE `DistributedWorker` is an executable which can be build using the build-script and providing the `--target` argument:

   ```
   ./build.sh --target DistributedWorker
   ```

2. Starting the Distributed Workers

   Before executing Daphne on the distributed runtime, worker nodes must first be up and running. The executable built in the previous step can be started via the command:

   ```
   ./bin/DistributedWorker IP:PORT ,
   ```

   where `IP` and `PORT` are the IP and port the worker server will be listening to.

3. Export environment variables at coordinator

   Before running DAPHNE we need to specify which IPs and ports the workers are listening too. At the time of writing, the environmental variable `DISTRIBUTED_WORKERS` is used for this purpose, where we list IPs and ports of the workers separated by a comma. For example, if a setup with two workers on IPs 192.168.1.1 and 192.168.1.2, both listening on port 5000, we would use the following command:

   ```
   export DISTRIBUTED_WORKERS=192.168.1.1:5000,192.168.1.2:5000
   ```

4. Execute DAPHNE at coordinator

   When all workers are up and running and the environmental variable is set, the DAPHNE script can be executed by the coordinator in a distributed manner. The distributed runtime is enabled by specifying the `--distributed` flag. The communication backend can be specified via

the `--dist_backend` flag. In order to execute the `example.daph` script with the synchronous gRPC framework, the command to be used is:

```
./bin/daphne --distributed --dist_backend=sync-gRPC ./example.daph
```

The distributed runtime has some additional utility methods that are the backbone of each communication backend worker implementation. Those are:

1. Store method

   Stores an object in memory and returns an identifier.

2. Compute method

   Receives the IR code fragment along with identifier of inputs, computes the pipeline and returns identifiers of pipeline outputs.

3. Transfer method

   Returns an object using an identifier.

As far as the actual I/O operations on the worker, the same kernels that were examined on the local execution section above can be used. This is another benefit of the design, as kernels need to be implemented only once by contributors (or at least very minor changes are needed) for both the local and distributed runtime.

Chapter 4

Experiments

4.1 Setup

4.1.1 Hardware

The following experiments were conducted on an AWS cluster, utilizing t3.xlarge EC2 instances with the following characteristics:

- 4 vCPUs
- 16 GB RAM
- 45 GB gp3 SSD as general purpose storage device
- 5 GB gp3 SSD as the Lustre block device

4.1.2 Software

This was a bare-metal installation, meaning that fresh Red Hat Enterprise Linux (RHEL) 8.10 images were used and normal AWS[1] EC2 instances and networking, as opposed to existing AWS solutions, for example, Lustre FS by AWS[2]. This was done to keep as much control as possible over the experiments and the configuration, as well as to investigate potential challenges for future installations to other systems.

We have 9 total available EC2 instances. One of them is the DAPHNE coordinator and combined Lustre MGS and MDS node. The other 8 instances are both Lustre clients and DAPHNE workers. DAPHNE is deployed using Docker, as described in an earlier section. The kernel version of the RHEL images (after the Lustre patch) is 4.18.0-513.9.1.el8_lustre.x86_64. The Lustre version used is 2.15.4.

Through our experiments, we investigate the effect of the Lustre striping parameters as well as different techniques aimed at improving I/O performance. We focus on simple I/O workflows:

1. A random matrix is generated with varying dimensions, as seen in Figure A.3.
2. The Lustre kernels are used to read and write the generated file, as seen in Figure A.2.

All the experiments are executed three times and the average time of the executions is used. The generated files is deleted after each experiment, to ensure that the file system is always under no stress other than running experiment.

[1] https://aws.amazon.com/
[2] https://aws.amazon.com/fsx/lustre/

4.2 Comparison of existing kernels with new proposed Lustre kernels

In this set of experiments, the goal is to determine whether the proposed Lustre kernels perform better than the existing ones and to observe their scalability. Figures 4.1 and 4.2 show the comparison between the existing kernels and the newly developed Lustre kernels. In those experiments, the file is striped across all 8 OSTs.

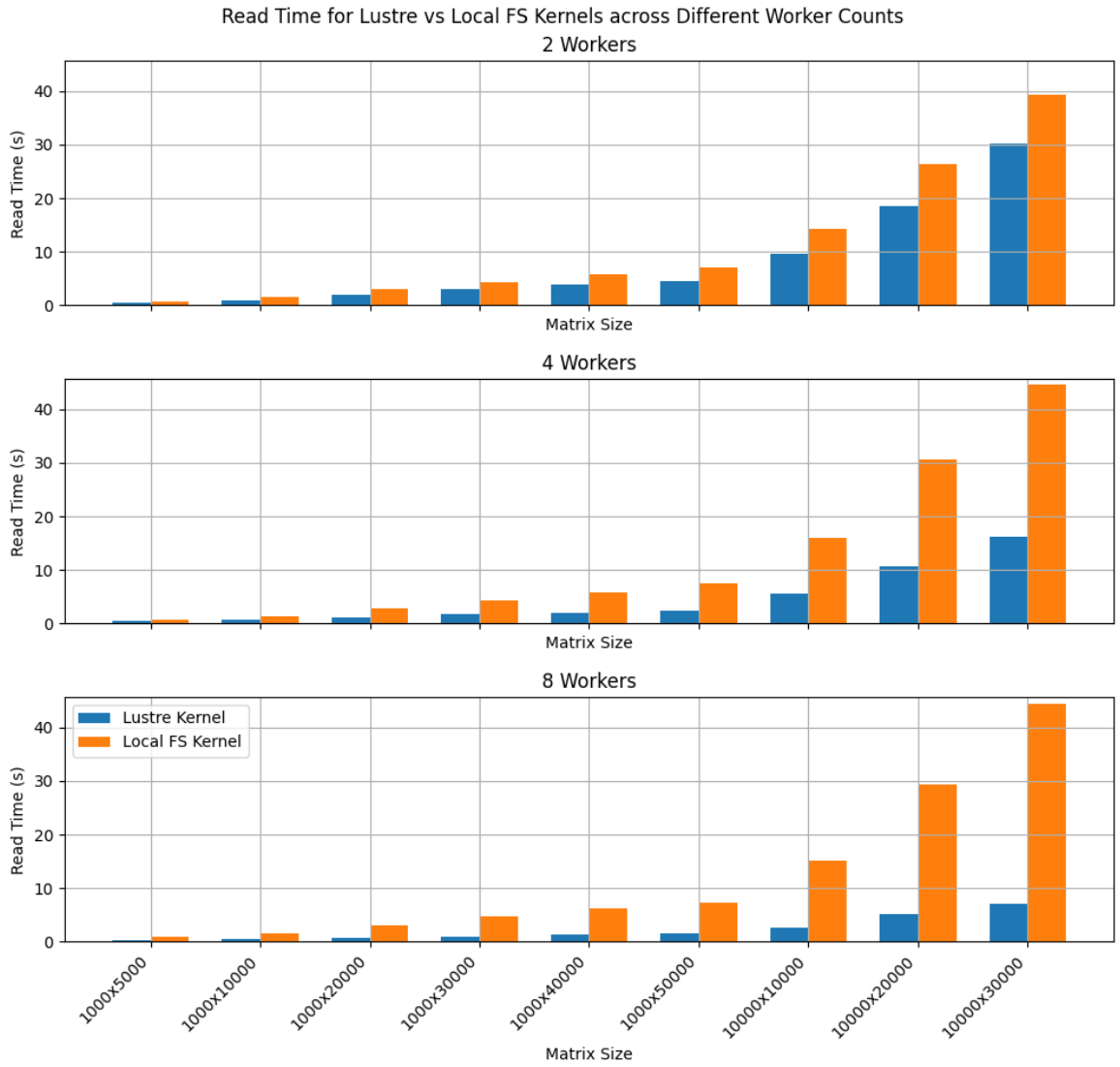


Figure 4.1: Comparison of Lustre and existing kernels. Read time vs. number of worker nodes and matrix size (CSV files).

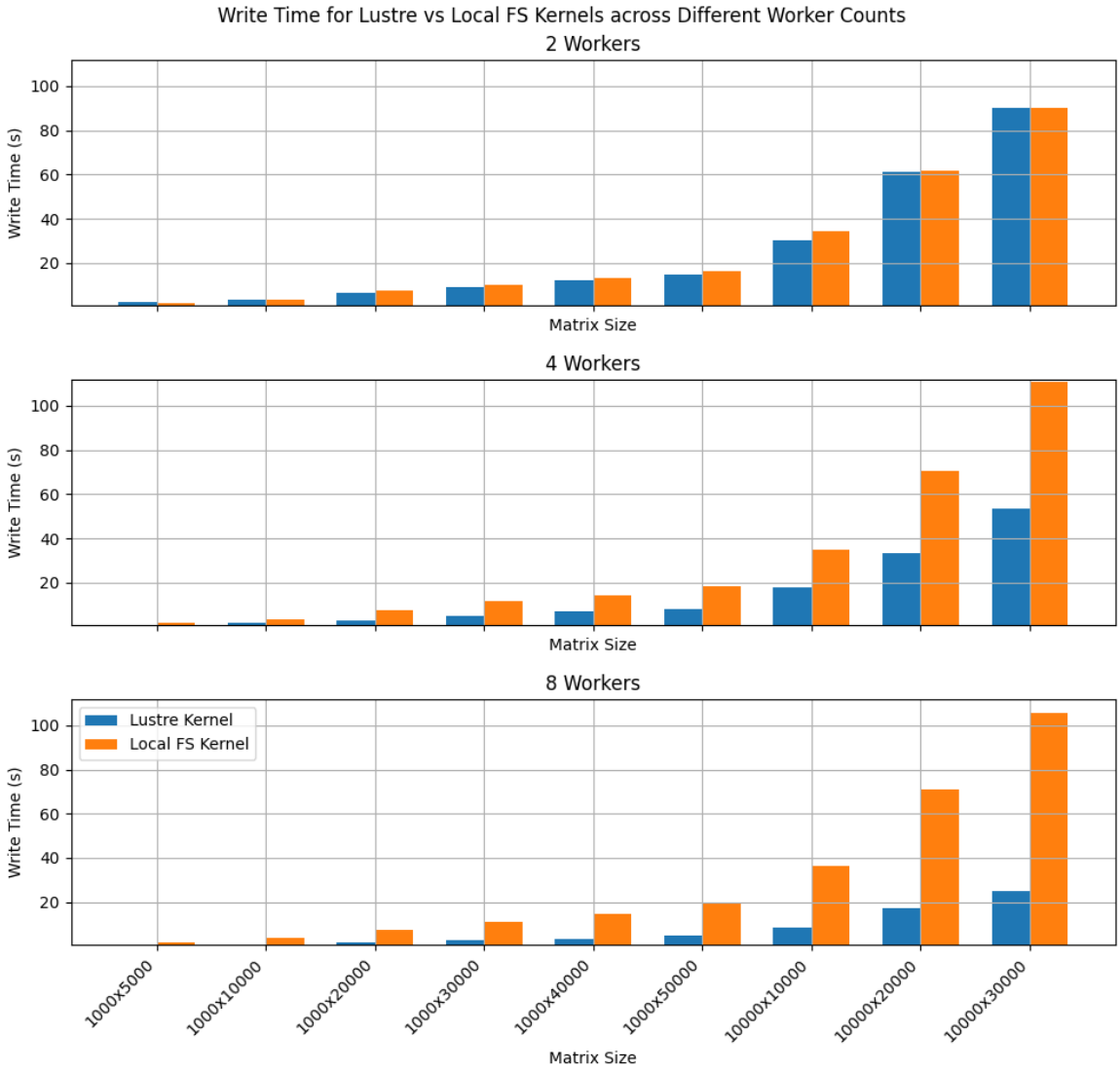


Figure 4.2: Comparison of Lustre and existing kernels. Write time vs. number of worker nodes and matrix size (CSV files).

Figures 4.3 and 4.4 show how the number of stripes (meaning the number of OSTs the file is striped to) affects performance. In the experiments of this section, the stripe size is fixed to 65 KB and the I/O buffer to 1MB. Values of the CSV are padded up to 17 characters. This is quite a large amount of padding, and it was used to showcase that even with this the Lustre kernels perform better than the existing ones. For the experiments following this section, this has been reduced to 8, which is enough for the data types used.

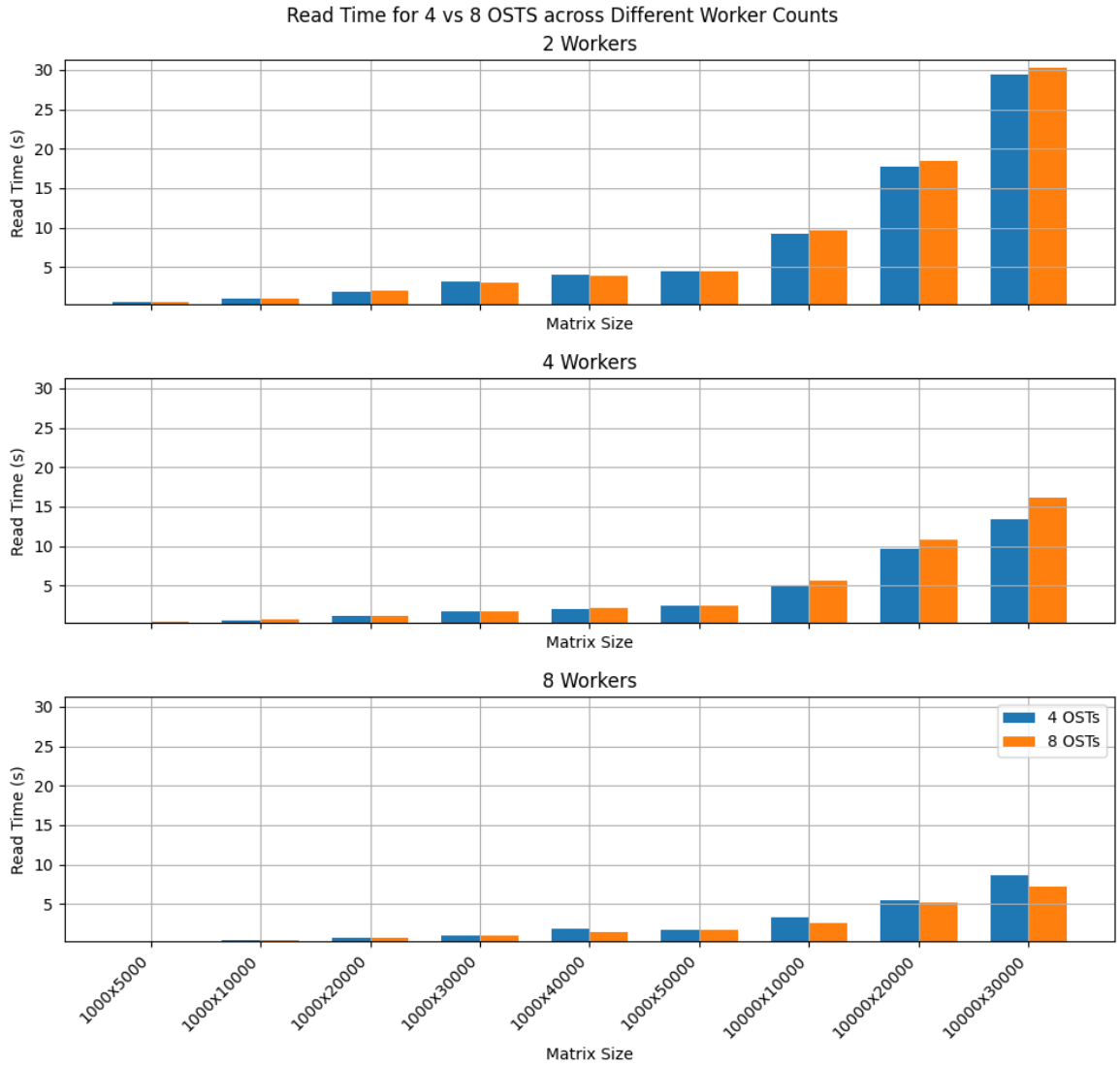


Figure 4.3: Read time vs. number of worker nodes and matrix size for 4/8 OSTs (CSV files).

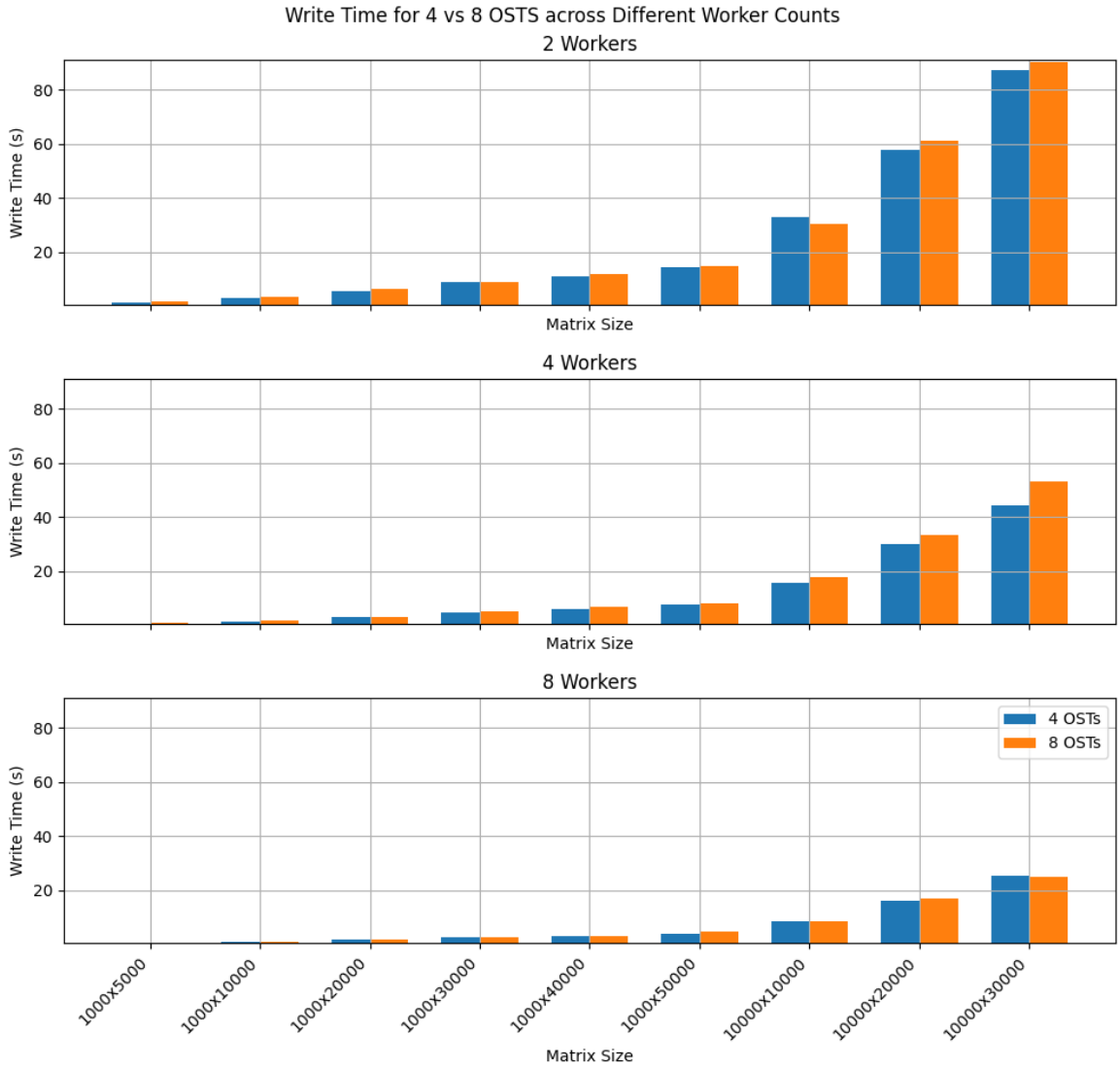


Figure 4.4: Write time vs. number of worker nodes and matrix size for 4/8 OSTs (CSV files).

For all the following experiments, the number of OSTs the file is striped upon is always equal to the number of workers. This is done to closely resemble clusters where each compute node has an associated storage device.

4.3 Stripe size

The default and recommended value for the stripe size is 1 MB [16]. We wanted, however, to determine whether in our application different stripe sizes would make more sense. Figures 4.5 and 4.6 show the comparison for different stripe sizes across 1 and 8 workers. We chose to perform the experiments only with the least and most available workers since these are the most volatile configurations. The values displayed on the legend are in bytes. Note that some matrix sizes are missing. This is because the increase of the matrix size from, e.g., 1000x10000 to 1000x20000 should not produce any different results for different stripe sizes, considering we have already reached a sufficiently large file.

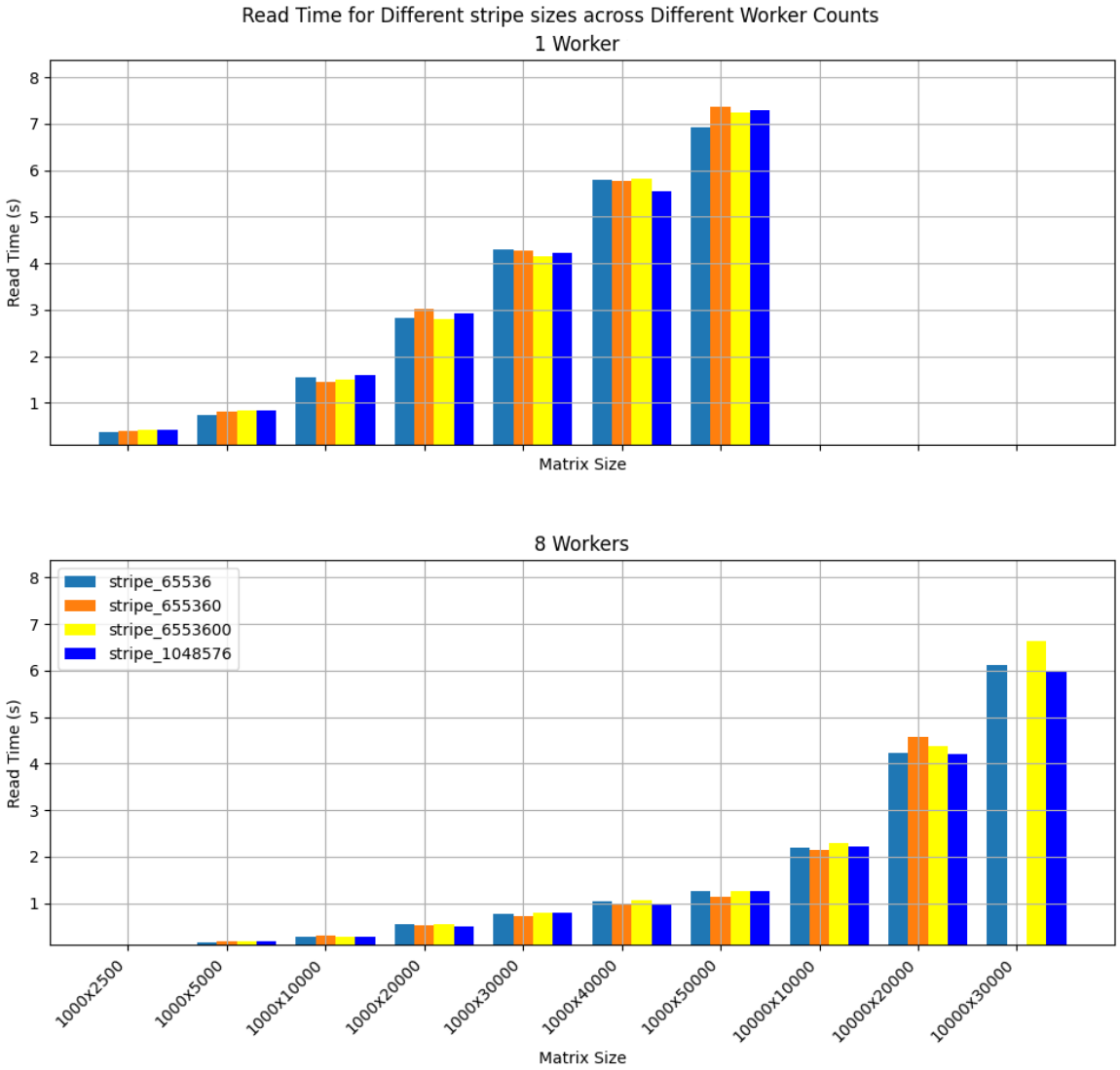


Figure 4.5: Comparison of different stripe size values for 1 MB buffer. Read time vs. number of worker nodes and matrix size (CSV files).

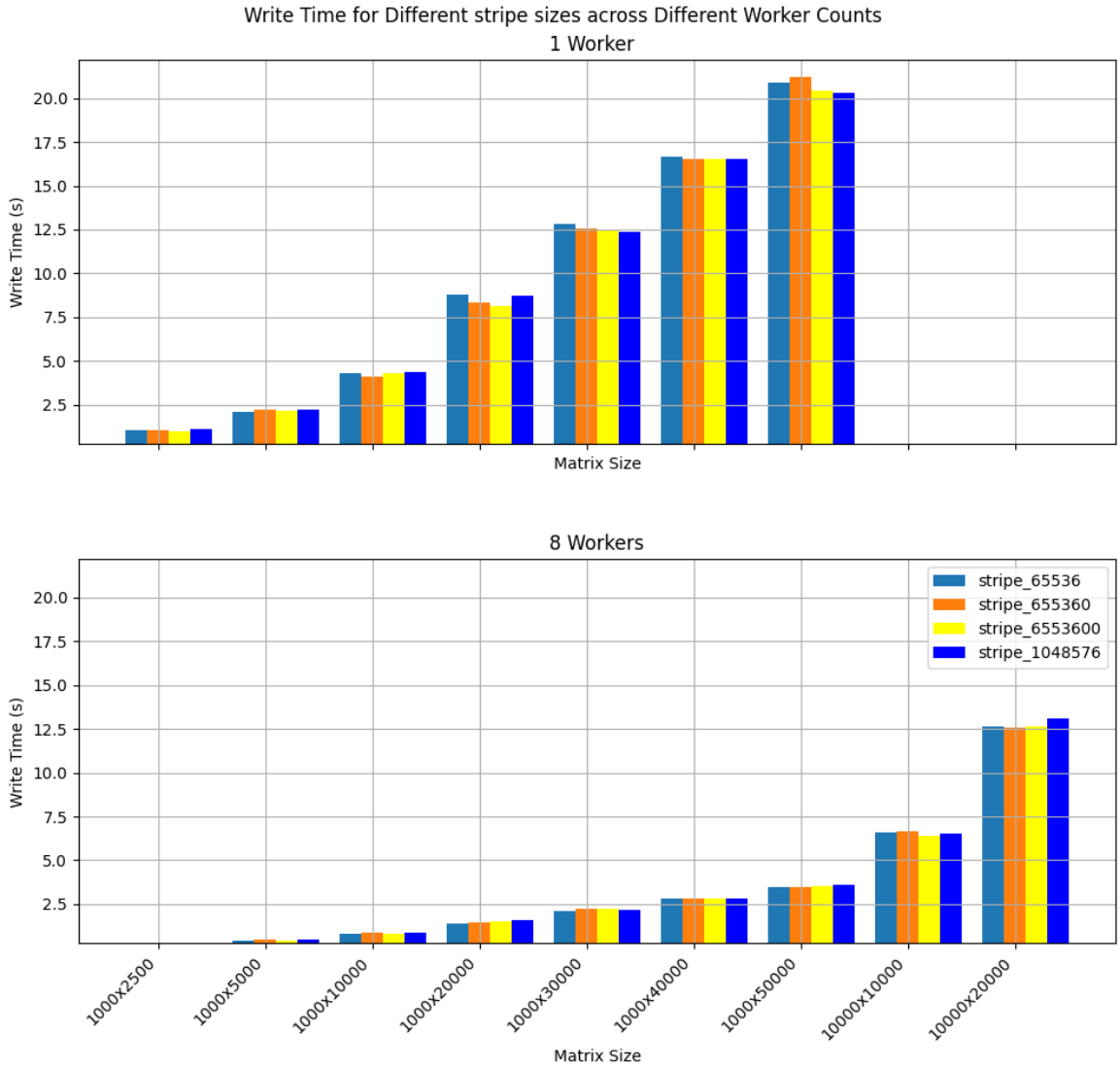


Figure 4.6: Comparison of different stripe size values for 1 MB buffer. Write time vs. number of worker nodes and matrix size (CSV files).

An additional consideration was whether the I/O buffer of 1 MB was too big. Figures 4.7 and 4.8 show the same experiments but with a smaller I/O buffer of 100 KB. We observe that the stripe size does not affect performance, for both the 1 MB and 100 KB buffers. Moving forward, we use the recommended value of 1 MB for the stripe size and the 1 MB buffer.

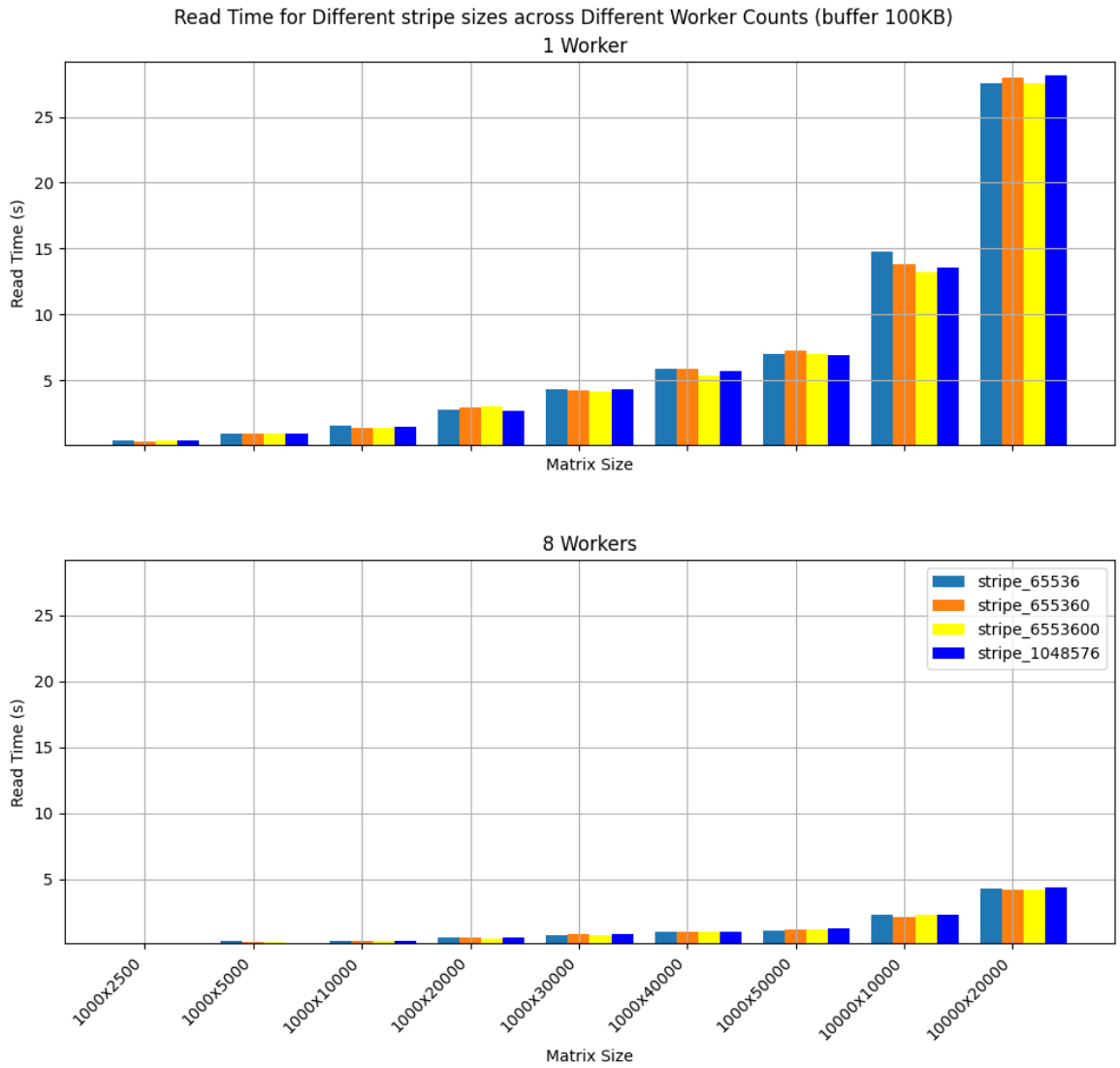


Figure 4.7: Comparison of different stripe size values for 100 KB buffer. Read time vs. number of worker nodes and matrix size (CSV files).

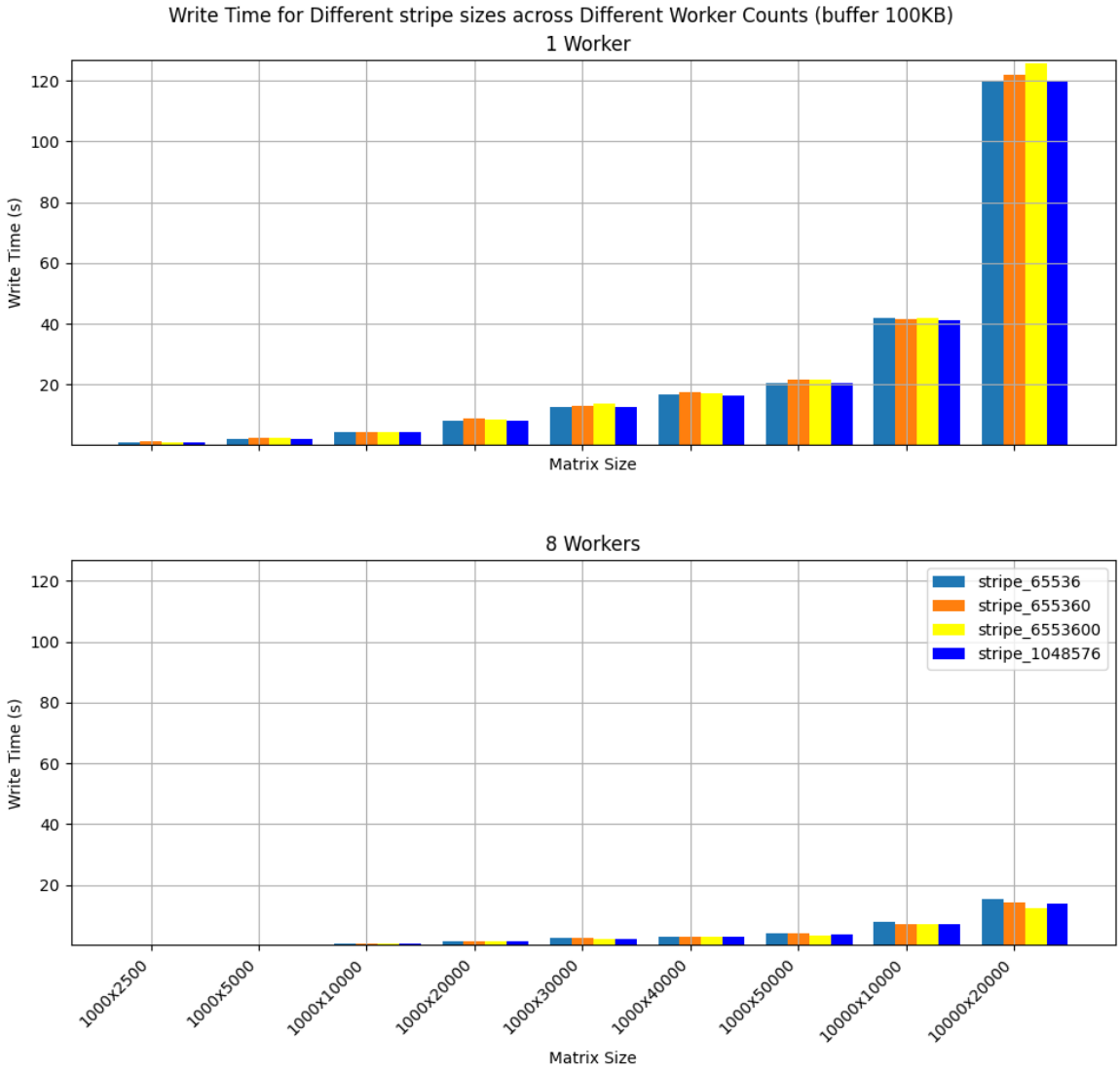


Figure 4.8: Comparison of different stripe size values for 100 KB buffer. Write time vs. number of worker nodes and matrix size (CSV files).

4.4 Truncating the file before writing

In this experiment, we truncate the CSV file to its expected size after its creation, before any write operation is performed. This is done to ensure that all future write calls to this file will be to preallocated regions, avoiding the need to extend the file on the OS level. The size of the file can easily be calculated since we are using the padding method described earlier. After the file creation, the coordinator truncates it to the desired size with the `ftruncate` systemcall via `ftruncate(fd, fileSize);` It should be mentioned that the implementation for the read kernels is the same, so the differences in figure 4.9 are due to randomness. This figure is included only for completeness and no meaningful results can be drawn from it. Figure 4.10 shows the comparison of write times between the implementation using `ftruncate` and the one without it.

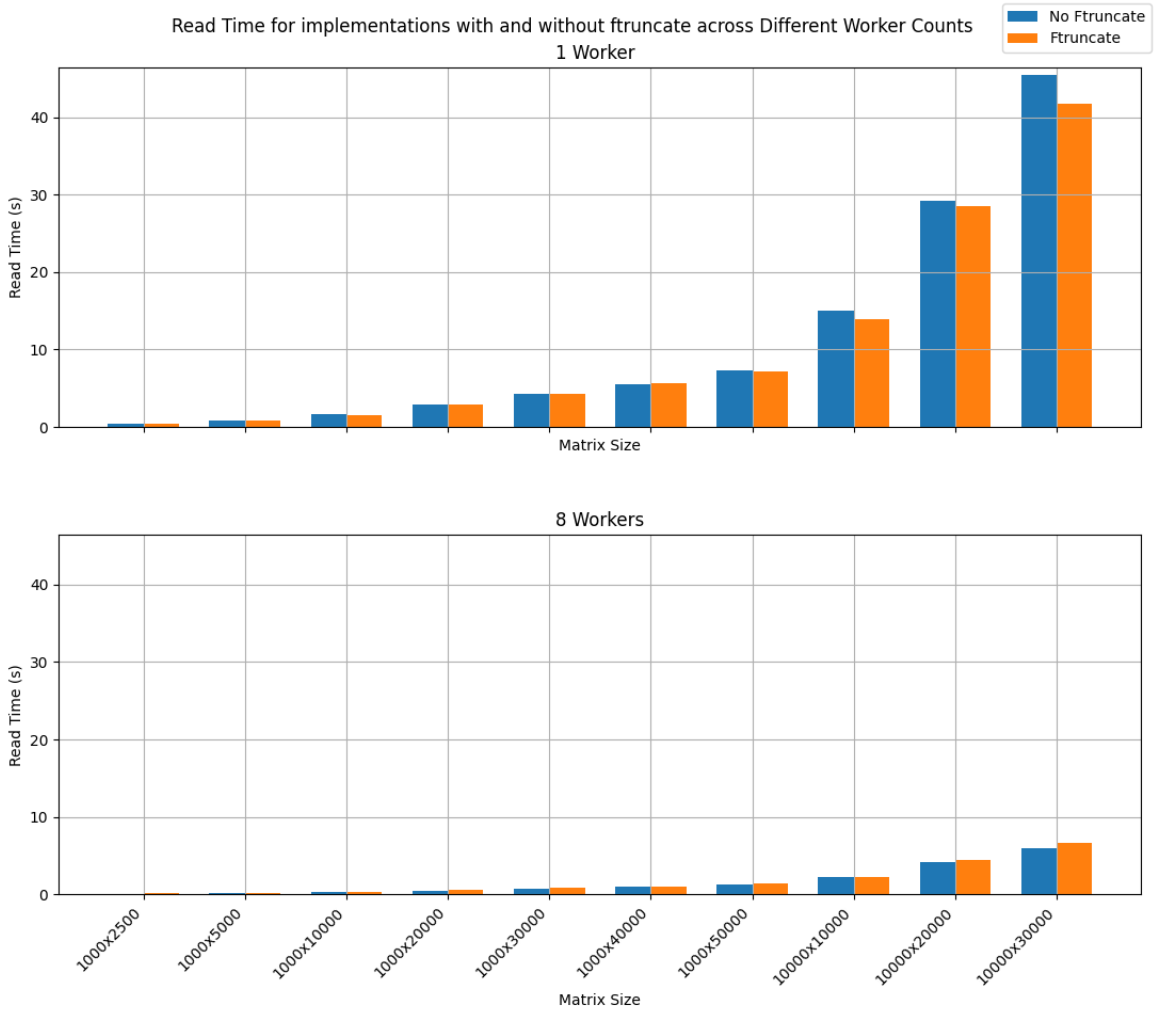


Figure 4.9: Comparison of `ftruncate` and not truncated implementations. Read time vs. number of worker nodes and matrix size (CSV files).

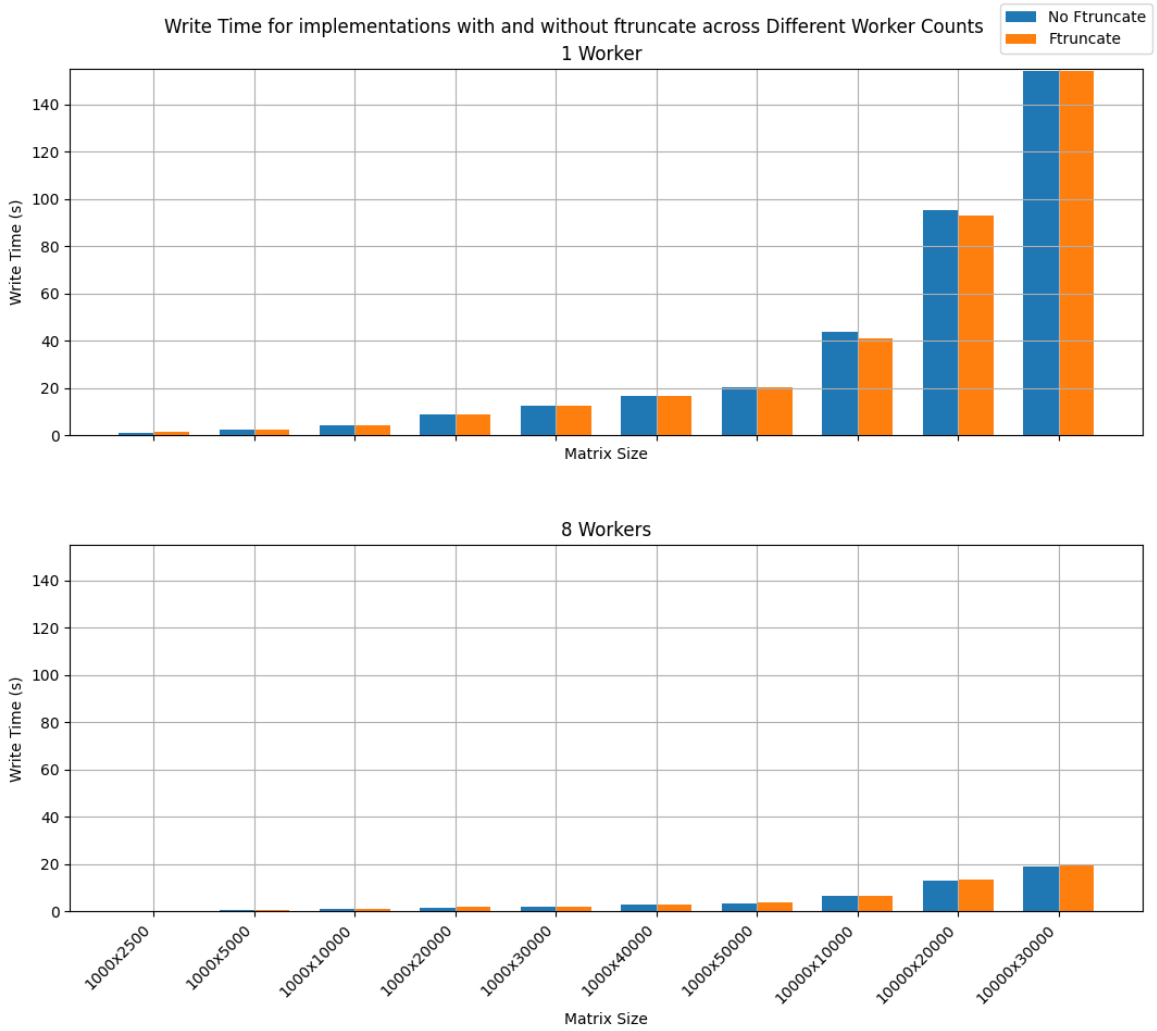


Figure 4.10: Comparison of `ftruncate` and not truncated implementations. Write time vs. number of worker nodes and matrix size (CSV files).

4.5 Stripe Alignment

This implementation tries to occupy only one OST per system call, on the application level. To do this we align each system call with a stripe of the file. This means that as many calls as possible are in the form (e.g. for read kernels) `read(stipe_size*multiplier)`. To achieve this, the first call must generally be of a size different than buffer size, to make sure that future calls are aligned. Since we observed in the previous experiment that truncating the file does not alter performance, we use the version with `ftruncate` here.

Figures 4.11 and 4.12 show the results for read and write kernels respectively.

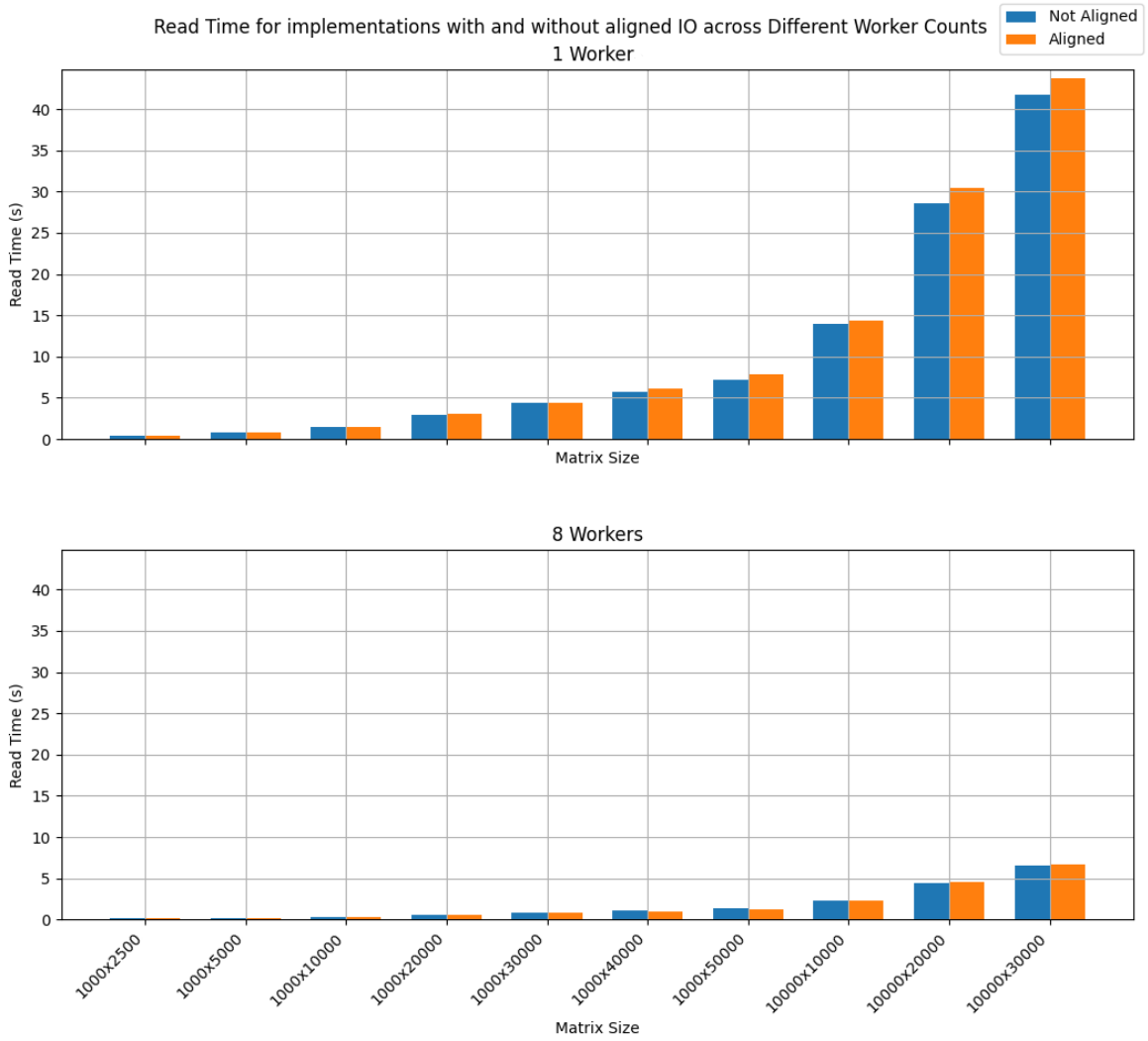


Figure 4.11: Comparison of aligned and not aligned implementations. Read time vs. number of worker nodes and matrix size (CSV files).

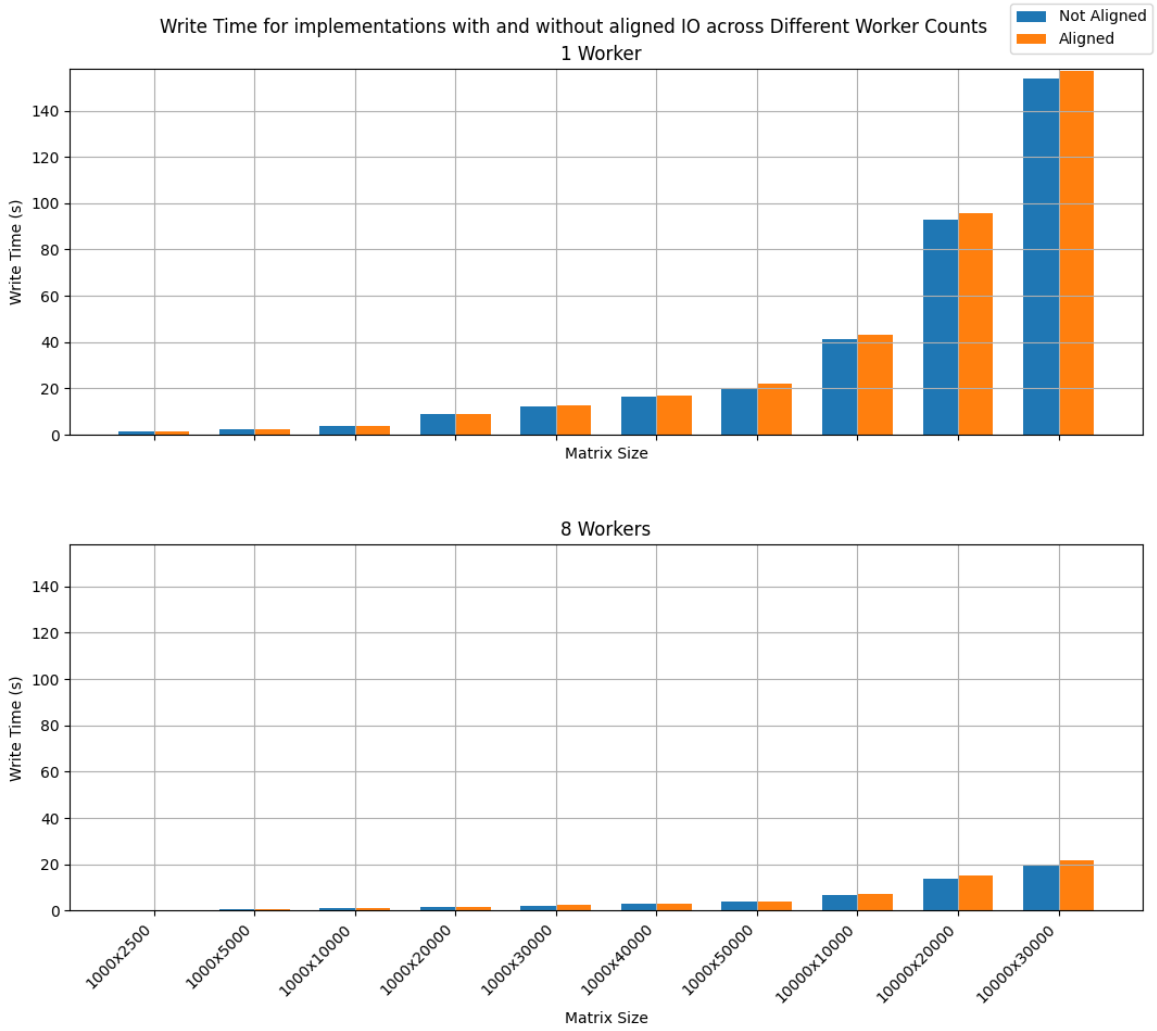


Figure 4.12: Comparison of aligned and not aligned implementations. Write time vs. number of worker nodes and matrix size (CSV files).

Chapter 5

Results

From the experiments above we can draw some conclusions regarding the performance of our implementation, as well as the various attempts to improve it further.

First and most importantly, we see that significant speedup is achieved compared to the existing implementation of the distributed runtime, which does not utilize a DFS. It is also evident that the implementation is scalable since increasing the number of workers reduces the execution time. From the first set of experiments, we conclude that the number of OSTs the file is striped across is not of significant importance since striping across 4 and 8 OSTs produces similar results. This means that a relatively small number of OSTs (4 or 8) can be used even with higher worker counts than the ones we have available in our environment (e.g. in the scale of 100 workers).
We see that the stripe size also does not significantly impact performance. The size of 1 MB is advised in the docs, and it is verified by our experiments that other values do not prove to be better.
Truncating the file to the desired size before performing write operations is logical but it does not affect performance, at least in our configuration.

One technique that was expected to alter performance but didn't is the stripe alignment technique. That is because each I/O operation only needs information from one OST to be completed. This, combined with the fact that other workers are simultaneously occupying the remaining OSTs, should result in a higher bandwidth utilization. The fact that performance remained the same even after trying to align read and write calls with OST stripes, could be explained by the buffering or page caching on the OS level. However, it is possible that a completely different bottleneck exists, for example, due to network latency, hardware limitations, or coordinator saturation. The limited control we have over the physical servers and their workload makes it difficult to further investigate those issues.

Chapter 6

Future Directions

We present some possible directions for future work related to this thesis

6.1 Intensive I/O algorithms

In this work we have used DaphneDSL scripts that read and write a matrix to the Lustre file system. This shows promising results and demonstrates the capabilities of our implementation. As a next step we would like to benchmark the Lustre integration with an algorithm that does intensive I/O. Some examples could be matrix operations algorithms, such as matrix multiplications, that store intermediate results to disk.

6.2 Direct I/O

Linux systems provide the `O_DIRECT` flag to perform direct I/O operations [1]. This mode however expects all buffers to be memory aligned, and it is not straightforward how this can be achieved in accordance to the matrix distribution. Even though in most cases this mode degrades performance, it would be interesting to see whether in the direct I/O mode the different striping configurations affect the results, as opposed to the current implementation.

6.3 Lustre comparison with HDFS

Recently the integration of HDFS with DAPHNE has been completed [43]. We would like to compare the two implementations, running benchmarks on the same cluster. The goal would be not only to compare the different file systems, but also their performance specifically in the DAPHNE workflows. This would allow us to better understand the performance and the limitations of each of the integrations, as currently we have only compared the DFS implementation with the existing (non DFS) one.

[1] https://man7.org/linux/man-pages/man2/open.2.html

Chapter 7

Conclusion

This thesis has demonstrated the successful integration of the DAPHNE runtime with the Lustre file system to enhance distributed runtime performance. We have developed specialized kernels that support I/O operations to CSV and Daphne Binary Data Format (dbdf) files. Through extensive experimentation, it was shown that Lustre-based kernels provide measurable improvements in read/write performance over existing solutions. It was also shown that the design is highly scalable, as performance increases when additional worker nodes are added. Findings indicate that while stripe size adjustments and pre-allocating file space had limited impact, the overall integration of Lustre with DAPHNE resulted in a substantial increase in system scalability and efficiency.

Looking ahead, several promising directions remain for future work. Investigating the effect of this integration with intensive I/O algorithms, exploring direct I/O mechanisms, and conducting comparative studies with other DFS solutions such as HDFS will provide further insights into enhancing DAPHNE's distributed runtime performance.

Βιβλιογραφία

[1] Piyush Agarwal, Harry Li και UT Austin. “A Survey of Secure, Fault-tolerant Distributed File Systems”. Στο: (Ιαν. 2008).

[2] *Big Data Storage Models Overview - Lustre, GlusterFS and Ceph - Bizety: Tech Research*. url: https://bizety.com/2019/04/09/big-data-storage-models-overview-lustre-glusterfs-and-ceph/ (επίσκεψη 23/01/2025).

[3] J Blomer. “A Survey on Distributed File System Technology”. Στο: *Journal of Physics: Conference Series* 608.1 (Απρ. 2015), σ. 012039. doi: 10.1088/1742-6596/608/1/012039. url: https://dx.doi.org/10.1088/1742-6596/608/1/012039.

[4] Peter Braam. “The Lustre Storage Architecture”. Στο: *CoRR* abs/1903.01955 (2019). arXiv: 1903.01955. url: http://arxiv.org/abs/1903.01955.

[5] Inc. Cluster File Systems. “Lustre : A Scalable , High-Performance File System Cluster”. Στο: 2003. url: https://api.semanticscholar.org/CorpusID:16120094.

[6] George Coulouris κ.ά. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. isbn: 0-13-214301-1.

[7] Lonnie D Crosby. “Performance Characteristics of the Lustre File System on the Cray XT5 with Respect to Application I/O Patterns”. Στο: (2009).

[8] Lonnie D Crosby. *Performance Characteristics of the Lustre File System on the Cray XT5 with Respect to Application I/O Patterns*. en. 2009. url: https://cug.org/5-publications/proceedings_attendee_lists/CUG09CD/S09_Proceedings/pages/authors/archive%20files/13A-LCROSBY-PAPER.pdf (επίσκεψη 16/12/2024).

[9] Patrick Damme κ.ά. “Daphne: An open and extensible system infrastructure for integrated data analysis pipelines”. Στο: *Conference on Innovative Data Systems Research*. 2022.

[10] *DAPHNE Binary Format documentation*. GitHub. url: https://github.com/daphne-eu/daphne/blob/main/doc/BinaryFormat.md (επίσκεψη 07/02/2025).

[11] Suman De και Megha Panjwani. “A Comparative Study on Distributed File Systems”. Στο: Απρ. 2021, σσ. 43–51. isbn: 978-3-030-68290-3. doi: 10.1007/978-3-030-68291-0_5.

[12] Phillip Dickens και Jeremy Logan. “Towards a High Performance Implementation of MPI-IO on the Lustre File System”. en. Στο: *On the Move to Meaningful Internet Systems: OTM 2008*. Επιμέλεια υπό Robert Meersman και Zahir Tari. Τόμ. 5331. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, σσ. 870–885. isbn: 978-3-540-88870-3 978-3-540-88871-0. doi: 10.1007/978-3-540-88871-0_61. url: http://link.springer.com/10.1007/978-3-540-88871-0_61 (επίσκεψη 10/12/2024).

[13] Patrick Farrell. *Shared File Performance Improvements*. en. url: https://www.opensfs.org/wp-content/uploads/2015/04/Shared-File-Performance-in-Lustre_Farrell.pdf (επίσκεψη 16/12/2024).

[14] *Filesystems in IO500*. url: https://www.opensfs.org/wp-content/uploads/2020/04/Lustre_IO500_v2.pdf (επίσκεψη 11/02/2025).

[15] John Fragalla, Bill Loewe και Torben Kling Petersen. “New Lustre features to improve Lustre metadata and small-file performance”. en. Στο: *Concurrency and Computation: Practice and Experience* 32.20 (2020). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5649, e5649. issn: 1532-0634. doi: 10.1002/cpe.5649. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5649 (επίσκεψη 24/11/2024).

[16] Anjus George κ.ά. *Understanding Lustre Internals. Second Edition*. Αδημοσίευτη ερευνητική εργασία. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), Σεπτ. 2021. doi: 10.2172/1824954. url: https://www.osti.gov/biblio/1824954.

[17] Quentin Guilloteau, Jonas H. Müller Korndörfer και Florina M. Ciorba. “Seamlessly Scaling Applications with DAPHNE”. Στο: *COMPAS 2024 - Conférence francophone d’informatique en Parallélisme, Architecture et Système*. Nantes, France, Ιούλ. 2024. url: https://hal.science/hal-04637841.

[18] Jaehyun Han, Deoksang Kim και Hyeonsang Eom. “Improving the performance of lustre file system in hpc environments”. Στο: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2016, σσ. 84–89.

[19] Richard Henwood και Andreas Dilger. *Why Use Lustre*. 2021. url: https://wiki.whamcloud.com/display/PUB/Why+Use+Lustre (επίσκεψη 23/01/2025).

[20] Mark Howison κ.ά. “Tuning HDF5 for Lustre File Systems”. Στο: 2010. url: https://api.semanticscholar.org/CorpusID:15211704.

[21] Ilin Tolovski κ.ά. *DAPHNE Final Project Report*. Αδημοσίευτη ερευνητική εργασία. KNOW, 2025. url: https://daphne-eu.eu/wp-content/uploads/2025/01/D1.6-DAPHNE-Final-Project-Report-1-1.pdf (επίσκεψη 06/02/2025).

[22] *Introduction to Lustre Architecture*. url: https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf (επίσκεψη 01/02/2025).

[23] Zoi Kaoudi και Jorge-Arnulfo Quiané-Ruiz. “Unified data analytics: state-of-the-art and open problems”. en. Στο: *Proceedings of the VLDB Endowment* 15.12 (Αύγ. 2022), σσ. 3778–3781. issn: 2150-8097. doi: 10.14778/3554821.3554898. url: https://dl.acm.org/doi/10.14778/3554821.3554898 (επίσκεψη 14/12/2024).

[24] Jeremy Kepner κ.ά. “Lustre, hadoop, accumulo”. Στο: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. Σεπτ. 2015, σσ. 1–5. doi: 10.1109/HPEC.2015.7322476. url: https://ieeexplore.ieee.org/abstract/document/7322476 (επίσκεψη 24/11/2024).

[25] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010. isbn: 978-1-59327-291-3. url: https://books.google.gr/books?id=Ps2SH727eCIC.

[26] David Knaak και Quincey Koziol. “Tuning HDF5 for Lustre File Systems”. Στο: (). url: https://www.academia.edu/15944905/Tuning_HDF5_for_Lustre_File_Systems (επίσκεψη 16/12/2024).

[27] Eliezer Levy και Abraham Silberschatz. “Distributed file systems: concepts and examples”. en. Στο: *ACM Computing Surveys* 22.4 (Δεκ. 1990), σσ. 321–374. issn: 0360-0300, 1557-7341. doi: 10.1145/98163.98169. url: https://dl.acm.org/doi/10.1145/98163.98169 (επίσκεψη 24/11/2024).

[28] Wei-keng Liao. “Design and Evaluation of MPI File Domain Partitioning Methods under Extent-Based File Locking Protocol”. Στο: *IEEE Transactions on Parallel and Distributed Systems* 22.2 (Φεβ. 2011), σσ. 260–272. issn: 1045-9219. doi: 10.1109/TPDS.2010.74. url: http://ieeexplore.ieee.org/document/5445094/ (επίσκεψη 24/11/2024).

[29] *LUG2019-Lustre_Overstriping_Shared_Write_Performance-Farrell*. url: https://www.opensfs.org/wp-content/uploads/2019/07/LUG2019-Lustre_Overstriping_Shared_Write_Performance-Farrell.pdf (επίσκεψη 16/12/2024).

[30] "Lustre* Software Release 2.x - Operations Manual". Στο: ().

[31] Sandra Mendez κ.ά. *Best Practice Guide - Parallel I/O*. eng. Αδημοσίευτη ερευνητική εργασία. Zenodo, Φεβ. 2019. doi: 10.5281/zenodo.4700698. url: https://zenodo.org/records/4700698 (επίσκεψη 16/12/2024).

[32] Michael Moore. "Exploring Lustre Overstriping For Shared File Performance on Disk and Flash". Στο: 2019. url: https://api.semanticscholar.org/CorpusID:209465945.

[33] Maryam M Najafabadi κ.ά. "Deep learning applications and challenges in big data analytics". en. Στο: *Journal of Big Data* 2.1 (Δεκ. 2015), σ. 1. issn: 2196-1115. doi: 10.1186/s40537-014-0007-7. url: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-014-0007-7 (επίσκεψη 24/11/2024).

[34] Xueting Pan, Ziqian Luo και Lisang Zhou. *Navigating the Landscape of Distributed File Systems: Architectures, Implementations, and Considerations*. arXiv:2403.15701. Μαρ. 2024. doi: 10.48550/arXiv.2403.15701. url: http://arxiv.org/abs/2403.15701 (επίσκεψη 24/11/2024).

[35] Prabhakaran Murugesan. *Seminar Presentation for ECE 658 - Distributed File Systems*. url: https://www.engr.colostate.edu/ECE658/2013/onlinepresentation/Prabhakaran/Prabhakaran.pdf (επίσκεψη 22/01/2025).

[36] Prabhat και Quincey Koziol. *High Performance Parallel I/O*. 1st. Chapman & Hall/CRC, 2014. isbn: 1-4665-8234-0.

[37] *Practical Examples for Efficient I/O on Cray XT Systems (CUG 2009)*. SlideShare. 21 Μάι. 2009. url: https://www.slideshare.net/slideshow/larkin2009slides/1469541 (επίσκεψη 18/12/2024).

[38] Aiswarya Raj, Jan Bosch και Helena Olsson. "Data Pipeline Management in Practice: Challenges and Opportunities". Στο: Νοέ. 2020, σσ. 168–184. isbn: 978-3-030-64147-4. doi: 10.1007/978-3-030-64148-1_11.

[39] Galen Shipman κ.ά. "Lessons learned in deploying the world's largest scale Lustre file system". Στο: *The 52nd Cray user group conference*. 2010.

[40] Abraham Silberschatz, Peter Baer Galvin και Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018. isbn: 978-1-118-06333-0. url: http://os-book.com/OS10/index.html.

[41] Andrew S. Tanenbaum και Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. isbn: 0-13-239227-5.

[42] Tran Doan Thanh κ.ά. "A Taxonomy and Survey on Distributed File Systems". Στο: *2008 Fourth International Conference on Networked Computing and Advanced Information Management*. Τόμ. 1. 2008, σσ. 144–149. doi: 10.1109/NCM.2008.162.

[43] Dimitrios Tsoumakos κ.ά. *DAPHNE: D4.3 Improved DSL Runtime Prototype and Overview*. Αδημοσίευτη ερευνητική εργασία. ICCS, 2023. url: https://daphne-eu.eu/wp-content/uploads/2023/12/D4.3-Improved-DSL-Runtime-Prototype-and-Overview-.pdf (επίσκεψη 26/08/2024).

[44] Weikuan Yu κ.ά. "Exploiting Lustre File Joining for Effective Collective IO". Στο: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. Μάι. 2007, σσ. 267–274. doi: 10.1109/CCGRID.2007.51. url: https://ieeexplore.ieee.org/abstract/document/4215390 (επίσκεψη 24/11/2024).

[45] Tiezhu Zhao κ.ά. "Evaluation of a Performance Model of Lustre File System". Στο: *2010 Fifth Annual ChinaGrid Conference*. ISSN: 1949-1328. Ιούλ. 2010, σσ. 191–196. doi: 10.1109/ChinaGrid.2010.38.

Appendix A

Code segments

```
G = readCOO("./AuthorCOO.csv"); // n-by-n boolean matrix
n = nrow(G); // get the number of vertexes
maxi = 100;
c = seq(1, n); // init n-by-1 matrix of vertex IDs
diff = inf; // init diff to +Infinity
iter = 1; // iterative computation of connected components
while(diff>0 & iter<=maxi) {
    u = max(rowMaxs(G * t(c)), c); // neighbor propagation
    diff = sum(u != c); // # of changed vertexes
    c = u; // update assignment
    iter = iter + 1;
}
```

Figure A.1: DaphneDSL program to compute the connected components of a co-author graph [9].

```
t1 = now();
a = readMatrix($G);
t2 = now();
t3 = now();
writeMatrix(a, $D);
t4=now();

// Print elapsed times in seconds.
print("read time[s]: ", 0, 0);
print((t2 - t1)*10.0^(-9), 0, 0);
print("write time[s]: ", 0, 0);
print((t4 - t3)*10.0^(-9), 0 ,0);
```

Figure A.2: DaphneDSL script that reads/writes from the provided G/D parameters respectively.

```
A = rand($R, $C, 0.0, 1.0, 1.0, 42);
writeMatrix(A, $G);
```

Figure A.3: DaphneDSL program to create a random matrix with R rows and C columns and write it at location G.