UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Bridging the Gap Between Object Stores and HPC

Alex Barceló-Cuerda

Advisors:
Dr. Anna Queralt
Dr. Toni Cortes

Dissertation submitted to the Department of Computer Architectures (DAC) in partial fulfilment of the requirements for degree of Doctor of Philosophy in Computer Architectures

December 21, 2023

Acknowledgements

I want first and foremost to thank both Anna and Toni. This thesis would not have materialized if not for them. Toni has been able to impart healthy research habits (and point out my bad ones!) while Anna has (successfully?) grounded my broken creativity into a more palatable style –and she has always been able to make the good questions, which instigated good answers and good research. All this on top of their hard job as advisors.

There are a lot of friends and colleagues whom I want to thank, but that will be done through backchannels and over a pint of beer –thank you :)

I want to think that the radical positivity of my sister has helped me push through the PhD student years. And I have to apologize to my mother, who will want to read this document and she is in for a ride.

$+\alpha\rho$

Abstract

Efficient data management is a fundamental aspect of application workflows, particularly in the context of High-Performance Computing environments. This thesis examines the potential of object stores as a robust candidate for HPC storage systems.

The first contribution of this thesis is the conditioning of a distributed object store combined with the architectural design of its integration with a computational framework. The resulting object store can efficiently use HPC resources. This is achieved by leveraging active capabilities and results in an increase of data locality and execution performance.

After this first contribution –the fruit of which is a software stack architecture, in addition to its evaluation characterization– we propose two additional distinct contributions (second and third contributions of this thesis) that showcase the potential of such an active object store in the realm of HPC. For the second contribution, we introduce a new software mechanism for iterating datasets and performing distributed execution on this software stack. Our proposal is able to abstract the relationship between data distribution and task distribution while delivering performance benefits in a wide variety of scenarios. This is achieved without sacrificing the programmability of the task based programming model.

The third contribution of this thesis is related to the integration of a hardware technology, the non-volatile memory (NVM). The main idea is to leverage the active aspect of the object store and combine it with the byte-addressable nature of these new kind of memories. The characteristics of NVM allow to perform in-place computation –without needing to stage-in data from this persistent memory layer– and the active mechanism is able to invoke the execution next to its data. These two aspects intertwine and allow the object store to manage NVM space while boosting data locality and reducing overall memory footprint.

All contributions are implemented as part of dataClay (the storage system, an object store). This thesis includes the implementation and evaluation of these proposals with commonly used scientific applications. In addition to completing and evaluating the integration of dataClay and COMPSs (the execution framework), the software mechanisms for iterating datasets are also implemented and evaluated with Dask (another task-based execution framework). The different scenarios explored showcase the benefits that our contributions bring to the proposed software stacks within the HPC ecosystem.

Keywords: object store, active storage system, data locality, non-volatile memory, persistent memory, HPC, distributed computing, dataClay, COMPSs, Dask.

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we aim to demonstrate the utility of object stores in High-Performance Computing (HPC) environments. On top of the design and conditioning of a full object store storage stack (first contribution), we propose two additional contributions that elevate the efficiency and performance of object stores when running HPC applications: the first one focuses on software optimizations during the distributed dataset iteration and execution; the second one explores the potential and enhancements at the storage system through the integration of a new storage tier: Non-Volatile Memory (NVM). Our work addresses a critical part of HPC –the storage system– and these contributions have the potential to drastically change the effectiveness of object stores in this domain.

1.1 Motivation

In the last decades there has been a trend of scale-out strategies in order to cope with the exascale needs and the explosive growth of data generated and consumed by applications. Handling efficiently the scale-out, that is, increasing the total capacity by increasing the number of nodes, is a hard issue that has required a lot of efforts, both from academia and from industry. HPC is one of the fields that have experienced an incessant growth. This growth (more computing resources for HPC applications, more efficient hardware, larger datasets, more complex applications, and larger data processing capabilities) can translate into a positive impact on society. Improvements on the HPC ecosystem can ripple through other fields, bringing progress to domains such as climate and weather prediction, personalized medicine, complex ecosystems simulation. . . [34]

The exponential growth in the HPC field has catalyzed innovation and advancement in several research fields. Storage systems is one of the fields that has to cope with the data scalability and we will be discussing both the software and hardware aspects of the evolution that it has beared.

On the software side we can observe the appearance and evolution of distributed file systems such as Lustre[14] (started in 1999), PVFS[72] (initially released in 2003 and then evolved into OrangeFS in 2011) and GPFS[74] (which started as the *Tiger Shark file system* in 1993). However, most POSIX features in these distributed filesystems –e.g. their strong consistency, ACL, timestamps– are not really needed by HPC applications[54]. This kind of feature creep has motivated research and storage architectures focused on the object store[78] as a way to increase scalability of the storage system.

Object store systems started in the cloud computing arena –e.g. the key-value stores Dynamo[29] and Cassandra[25], which originated from within Amazon and Facebook respectively. This storage architectures have proven to be invaluable agents in the explosive growth experienced in the cloud arena[5, 10, 63]. We can also find them in different scenarios, such as HPC [1] or data analytics [73].

Also on the software side, the goal of improving overall performance on storage systems has motivated data locality schemes and similar strategies that move computation close to the data. This can be found in software projects such as Spark[96, 97] and other data-analytics frameworks. Similar strategies have also been used by in-situ analytics frameworks, where low-latencies are achieved thanks to the closeness of data and computing resources. One concept that we will focus on is the *active* feature of the software stack. This is not exclusive of the HPC world, as we can see in the SAMOS[35] project from 1992 –which coined the term *active* referring to database systems. Another example of an object store (and also the one that we will be focusing on in this thesis) is dataClay[58]. Generally, *active* systems are able to take advantage of data locality, as the execution routines are –by design– adjacent to their objects. The notion of *active* can also be applied to hardware, for instance as proposed by IBM in their Active Memory Cube design[60] which describes a "processing-in-memory architecture" at the hardware level.

From the hardware perspective it is relevant to highlight the appearance of Non-Volatile Memory (NVM) devices, which blur the gap between memory and storage tiers in the hierarchy. Those devices will trigger a significant change on how we architect storage systems and applications. Their byte-addressable interface and their performance, much closer to memory than to disk, open the door to computing directly on the stored data without having to bring it to RAM –as required for drives or any other kind of storage device. This can have a tremendous impact in *active* architectures, given that computation can be done directly from persistent memory –without the stage-in.

In order to support the exascale era, HPC needs not only the hardware but also the tools to use that hardware –i.e. programming models, algorithms, libraries. . . to make use of said resources. This thesis focuses first and foremost on the storage systems, but also on its close relationship with the programming model and execution. In this thesis we will be conditioning an object store for HPC and showing the architecture design, the novel contributions to the software stack and showcasing the performance of such storage system.

1.2 Research questions

The following research questions have been raised during the inception of this thesis:

RQ1 How can object stores be more efficiently used within HPC environments?

Which steps are needed in order to condition an object store for HPC? Which programming models can leverage this storage architecture?

RQ2 **How can software improvements be supported by object stores in order to improve performance?** For instance, how can the execution unto a distributed dataset be improved through new iteration patterns at the programming model level?

RQ3 **How can hardware improvements be supported by object stores in order to improve performance?** For instance, how can new NVM devices be supported from the storage system and simultaneously leveraged during application execution?

RQ4 **How much of an improvement can we expect when maximizing data locality during the execution of HPC applications?** What impact can we observe due to the *active* mechanism on the storage system? What are the benefits that persistent byte-addressable memory can offer?

1.3 Thesis statement

An object store leveraging active capabilities is a versatile and fast storage system architecture, suitable for HPC, and can efficiently take advantage of further improvements both on the software side (such as better iteration strategies during distributed execution) and hardware side (such as the availability of byte-addressable non-volatile memory devices).

1.4 Methodology

As the title of this thesis suggests, HPC is the main environment under consideration. This means that performance will be a basic metric of fitness of any software block. Raw performance is not the only aspect that we will discuss, but it is highly relevant. The contributions focus around new improvements and mechanisms from the object store point of view, and those will be validated through an iterative methodology:

1. Requirement analysis.
2. Design and implementation on the object store.
3. Design and implementation of HPC applications that can validate or challenge the implementation.
4. Experimental evaluation of the applications and their performance.
5. If the results are not satisfactory (there is a discrepancy between the expected performance and the obtained one), review the new discovered factors and go back to step 1.

6. Summarize the improvement, its implementation and its impact on the HPC environment.

This methodology applies to Chapters 4, 5 and 6.

The object store mentioned above will be dataClay, an already established active object store. In Chapter 3 we will present with more detail this storage system as well as the execution frameworks used for integration and validation.

1.5 Contributions

In order to endorse the thesis statement and answer the research questions exposed, we present the following contributions:

C1 Design the integration between a Python object oriented storage system and a distributed task-based programming model.

C2 Propose the programming interface required to improve the iteration and execution across distributed datasets through object store cooperation.

C3 Propose the mechanism to support non-volatile memory from an active object store.

1.6 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 surveys different technologies that are closely related to this thesis. This includes both *enabling* technologies (which have made possible our contributions) as well as *competing* technologies (which can be analyzed as alternatives to the solutions we propose).

Chapter 3 dives into the pre-existing frameworks that we will be focusing on. The contributions of this thesis are integrated into the software technologies explained in this chapter.

Chapter 4 starts to present and discuss the contributions of this thesis. This chapter will showcase the design decisions and steps that were necessary in order to condition the object store for HPC. The result of this conditioning is evaluated through a series of scientific applications.

Chapter 5 contains the design, implementation and evaluation of the software mechanisms for iterating datasets and performing distributed execution.

Chapter 6 contains the design, implementation and evaluation of a mechanism to make use of non-volatile memory from the active object store.

Chapter 7 serves as a comprehensive summary of the key contributions and significant findings presented throughout this thesis, as well as their current impact in the scientific community. It also offers an exploration of avenues for future research and development.

Chapter 2

State of the art

The HPC field is ever-changing, always following developments that affect both the hardware and the software environments. New hardware technologies have the potential to disrupt the status quo of HPC clusters and the approach to certain problems. Equally important is the software counterpart: paradigm shifts and new algorithms are phenomena capable to deeply affecting the way HPC applications are developed and their effective performance.

Object stores have penetrated to the HPC ecosystem more recently (when compared to other ubiquitous technologies present in HPC) but their presence may be capable of bringing a wave of disruption to not only data management but also distributed computing. The relationship between object stores and HPC is a cornerstone of our work.

In this chapter we will first look into the set of enabling technologies –technologies that are either influential or foundational for this thesis. After this, we will discuss the competing technologies –alternative solutions that overlap with the contributions of this thesis– and how do they compare with our proposal.

2.1 Enabling technologies

First of all we will discuss the different technologies that have paved the way for this thesis. This includes some technologies and software stacks that have been used during this thesis as well as hardware and software that has contributed to the object stores presence in the HPC field.

2.1.1 Distributed storage

The notion of horizontal scaling is an established practice that has been ingrained in the HPC arena for decades. It is logical that this kind of scale out –multiple interconnected machines instead of a single big one– also affects the storage architecture. Historically, distributed file systems have been the default technology for HPC clusters. In 1993 the *Tiger Shark file system* appeared, which later would become GPFS[74]. PVFS[72], released in 2003, become OrangeFS. Lustre[14], another distributed file system commonly used in HPC clusters, dates from 1999. Distributed file systems tend to be nearly POSIX compliant and provide a robust and stable interface (the one defined by the POSIX standard), something important for stability and compliance with existing applications.

The constraints imposed by the file system interface can become a burden in HPC environments. Lookup operations and hierarchy can become complex (with inodes, folders, file names, access times, etc.), and they lack both metadata or extensions (e.g. versioning, replication. . .). These reasons have motivated researchers to look into alternative storage systems such as the object store[54, 78]. The object store uses a more flexible interface that provides a lot of merits. Object access is monolithic through *get/put* operations which simplify sharing and multi-writer configurations. Object stores can easily have a scalable architecture. In the cloud arena they have been strong agents for a long time, as we can see by the success and ubiquity of S3 [5] –which comes from the Amazon ecosystem– which has become a *de facto* object store standard. The OpenStack group has Swift [63] as their in-house object store. MinIO[59] is another example of a distributed object storage compatible with the S3 interface. Object stores have been penetrating into the HPC ecosystem for a while now. Ceph[90, 10] is a distributed file system targeted for high-performance applications which includes RADOS[91], an object store interface. Intel has also contributed to this field by publishing DAOS[15], an object store that has been "designed from the ground up for massively distributed Non Volatile Memory" [1].

The proliferation of these storage stacks has been key in the viability of using object stores in the HPC ecosystem. Their success proves that their flexibility is a valuable asset. A key enabling technology for this thesis is dataClay[58], a distributed active object store. It combines the strengths of the object store and extends the interface with some active capabilities –a mechanism that improves data locality and overall performance during application execution. In the following chapter we will discuss with more detail its features and impact in the HPC context.

2.1.2 Task-based programming models

Task-based programming models have proven to be a robust and versatile way to approach development of applications for distributed environments. They provide natural programming patterns and achieve a high degree of performance. However, execution on this paradigm can be very sensitive to granularity –i.e., the quantity and execution length of tasks. Granularity is often linked with the block size of the data, and finding the optimal block size has several challenges, as it requires inner knowledge of the computing environment.

Dask[71] is an example of a task-based distributed framework. It self-describes as a "flexible library for parallel computing in Python". It has a very flat learning curve for Python developers, as it draws inspiration from commonly used data structures (such as NumPy *Arrays* and Pandas *DataFrames*). With those data structures in mind, Dask provides their distributed counterparts (e.g. *Dask Array* and *Dask DataFrame*) with a similar interface. Its flexible task scheduling mechanism is outlined in Figure 2.1. Aside from all the built-in algorithms (provided by Dask and by its machine learning library Dask-ML[27])

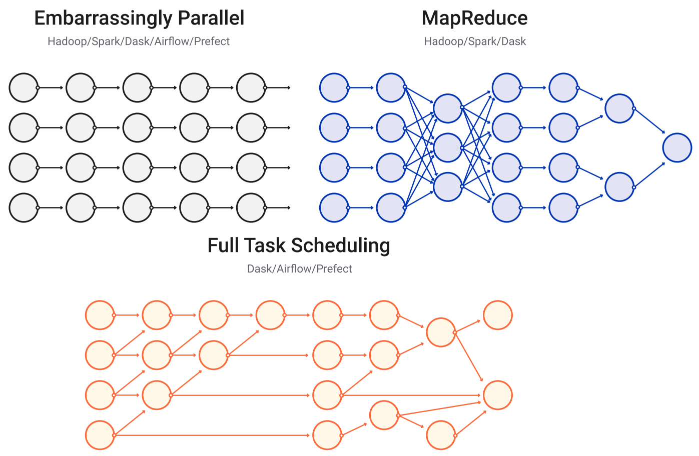


Figure 2.1: Different methods for task scheduling, from Dask documentation[8]

it is possible to develop different applications through the lower-level task invocation and make use of the API of their data structures; Dask documentation[8] discusses all of that and gives detail on the blocking applied to the data structures. Dask leverages the native Python representation of objects, and is able to exploit the in-memory data structures during computation.

Another task-based programming model is PyCOMPSs[79, 26]. This programming model tackles the burden of distributing tasks through a minimal set of annotations from the developer (i.e., minimizing the burden on the developer's shoulders). It is as flexible as Dask (i.e. it is a *Full Task Scheduling* software, following the nomenclature shown in Figure 2.1). In addition to that schedule flexibility, it is fully capable of managing read-write (i.e. `INOUT`) dependencies –a programming pattern that is full of caveats and outright discourage while using Dask. The COMPSs framework can be used from within Java, Python and C/C++.

Our contribution **C1** will cover the design and architecture related to the integration between dataClay and COMPSs (see Chapter 4). Contribution **C2**, which focuses on the iteration and execution across distributed data, will be evaluated on top of both COMPSs and Dask; this will be presented in Chapter 5.

2.1.3 Non-Volatile Memory Hardware

Aside from a continuous general growth found in the HPC environment (i.e. the progressive milestones of faster computation, faster connectivity, bigger storage, higher bandwidth, etc.), the hardware ecosystem has seen certain disruptive technologies that have changed the storage architecture. We will focus on the non-volatile memory, and particularly, the Intel Optane DC Persistent Memory devices[46]. In Figure 2.2 we can see the general outline of the existing storage technologies. On the top we see the fastest and most expensive (in

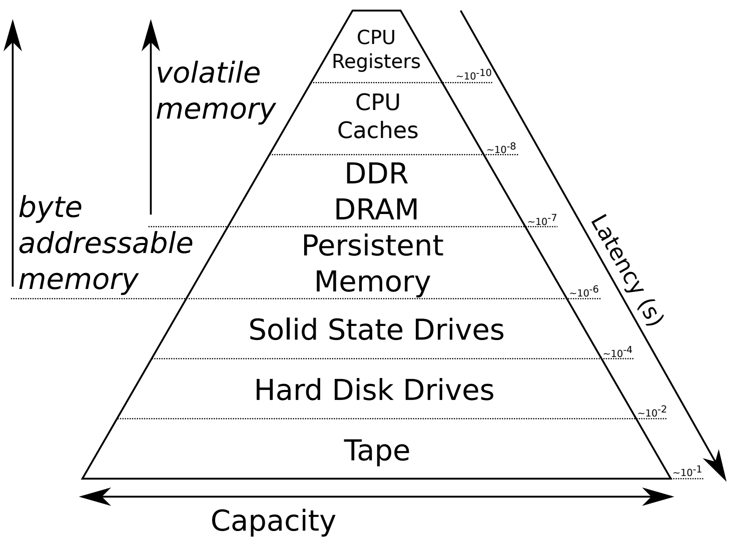


Figure 2.2: Memory hierarchy pyramid

terms of cost per byte) technologies. On the bottom, we see the cheapest solutions in terms of storage density and cost, but is also the slowest in terms of latency.

The Persistent Memory sits right between the fastest drives (persistent storage, block devices) and the slowest volatile memory (the DRAM). Comprehensive evaluations of the Optane DC Persistent Memory Module have been conducted, like the work of Izraelevitz[47] or the one by Weiland[92]. These kind of evaluations are key in order to understand the behaviour and potential of this technology. We can also find an *empirical guide* evaluation performed by Yang *et al.*[94]. Those evaluations show the often overlooked complexity of the performance behaviour of the OptaneDC persistent memory. One can find extensive information about the latency and bandwidth characterization in those documents.

This technology is able to unlock a lot of potential on object stores due to its performance capabilities and scalability. We have also mentioned DAOS[1], an object store that can leverage Optane DC non-volatile memory resources, but our contribution **C3** aims to support this kind of memory by coordinating with the *active* capabilities of the object store.

2.2 Competing technologies

There are different technologies and existing projects that compete with one or several aspects of our proposal. In this section, we will look at different aspects that directly relate to our proposal and discuss the alternative technologies and pre-existing research for each of those specific aspects.

2.2.1 Active features in storage systems

One of the software pillars in this thesis is dataClay, which is a distributed active object store. There are other storage systems that have active features and there is research related to the active mechanism that is also relevant in this thesis.

We have already mentioned Ceph RADOS[91] as a distributed object store. However, Ceph also has the concept of *object classes* [32], a feature that can be used during build/deploy stage of the storage system. These shared object classes are able to improve data locality by the execution of routines (previously deployed into the storage nodes, aka as OSDs) within the storage infrastructure itself instead of the client –which is an *active* feature of the storage system.

Another project that extends an object store with active features is Storlets [62], which belongs to OpenStack. Its purpose is to "extend Swift with the ability to run user defined computations –called storlets– near the data (...)". This describes the *active* feature of an object store as a means to improve data locality. This feature can be used during runtime, but their trigger mechanism is limited to specific operations on the object store (get, put, copy) and the executed code can only involve the object that is being accessed.

The aforementioned active features showcase the strengths of an active object store, but they cannot be used for arbitrary application flows. For the Storlets project, the scope is limited due to the trigger mechanisms. For the Ceph Object Classes, the classes must be defined prior to deployment. These limitations hinder the versatility of those solutions.

In the context of DAOS, the work of Lofstead *et al.* [55] proposes an architecture for a future exascale storage system, using DAOS as backend. It is relevant to mention the *Function and Analysis Shipping* mechanisms described in that article. The main idea behind those code shipping mechanisms is the avoidance of data transfers –which results in an improvement on data locality. However, those mechanisms are not integrated into the object store: DAOS is used for persistence (in I/O nodes) while execution is done in the compute nodes on top of HDF5. The goal of improving data locality through the shipping of code is shared with our objectives, but our design goes one step beyond and integrates this code shipping and code execution into the object store, resulting in an *active* object store.

We can see other storage systems that strive to improve performance by shipping code where data is. The ActiveSpaces framework [30] is an example: a storage system capable of moving (and executing) code where the data is being staged. Warren *et al.* [89] discuss an object/array centric approach that is able to use shared libraries or application executables in order to perform what we will be calling *active* operations. Those approaches show the advantages of active strategies; however they provide a low-level interface and expect extensive knowledge (from the application developer) regarding the architecture in terms of network and computation resources.

Up until now we have discussed the active features from a software point of view. However, this goal can also be achieved from the hardware side, as shown in Intelligent Disks (IDISKs [52]) and also discussed by Kannan *et al.* [50, 51]. That last line of research evaluates the active paradigm –on an architecture that includes additional hardware compute units– with NVRAM technologies. Nider *et al.* discuss *Processing in Storage Class Memory* [61], an article that shows promising results on this main idea of processing in NVM. The data locality obtained during execution in those articles resembles the one that we will be presenting, but our infrastructure will be based in a *software* storage system (an *active object store*).

2.2.2 Distributed data and distributed execution

One of the interactions that we will discuss extensively during this thesis is the link between the task-based programming model and the object store –always under the context of HPC. This integration is the first contribution of this thesis (**C1**). However, different programming models have similar goals regarding execution on distributed data. The most iconic one would be the the map-reduce programming model (MapReduce[28]). An application programmed following this programming model is divided into *map* tasks (which transform blocks, and are applied to all input blocks of the input) and *reduce* operations (which agglutinate multiple input blocks into single outputs). This general process is depicted in Figure 2.1. When using this software stack, the framework is responsible of managing the distribution of data and the data distribution, and has full jurisdiction regarding internal runtime optimizations.

Spark[97] is a widely used software stack that aims to be a unified engine for large-scale data analytics. It draws inspiration of the potential and benefits of the MapReduce programming model but aims to be more flexible and improve its performance. Regarding its storage counterpart, we can see that Apache HBase[88] is a distributed, scalable, big data store that is tightly integrated with the Spark software stack.

However, using either Spark or MapReduce requires to adapt (and rewrite) existing algorithms into using the explicit primitives and structures provided by the framework[26]. One of our contributions (**C2**) is to maintain the explicit iterations at the programming model level and simultaneously offer an enhancement for the iteration control structures, with a minimal impact on application programmability.

If we look at programming frameworks that expose lower level language primitives we can find Charm++[49], a parallel programming language based on C++. This object oriented language offers all the elemental primitives that we could expect, including iteration structures. This means that any enhancement that we discuss regarding iteration enhancement could also be implemented on top of Charm++. However, Charm++ does not include neither scheduling mechanisms nor dependency management comparable to the ones available in commonly used task-based workflows (such as COMPSs, the programming model that we will be using in this article, or Dask as previously discussed in this section). This implies a higher development effort for Charm++ applications; however, from the point of view of the iteration structures and their enhancement, using lower level elemental primitives yields no additional benefits.

2.2.3 Loop parallelization

One of our goals is to leverage data distribution and the task-based programming model. We tackle this (contribution **C2**) from the point of view of the storage system and its synergies with the programming model. However, there is relevant previous work in parallelization of loops and scheduling of work, looking at it from the point of view of the execution runtime. A foundational article on this field was published by Hummel *et al.* with the Factoring[44] method.

The work on Factoring has been extended with a focus on smarter and more complex scheduling mechanisms, e.g. with adaptative weighted factoring[12] or by adding also dynamic load balancing[24]. All this research shows the relevance of the granularity of tasks, an issue that we also aim to address. However, they tackle this issue for complex applications by discussing the method and its direct application into the application. Our focus is on a more generic mechanism (which we will discuss and evaluate in Chapter 5) that coordinates with the task-based programming model and is aware of the data placement.

We believe that certain existing research on the Factoring method could be used in order to further improve the benefits of our proposal. The mechanism that we propose is a foundation upon which could be augmented and refined with further improvements and heuristics such as the ones shown in previous research.

2.2.4 Block size conditioning

Our contribution **C2** proposes a design and mechanism that decouples granularity of data from granularity of tasks. There are already certain research and approaches to achieve this decoupling.

If we first look at the Spark ecosystem, we can find the Resilient Distributed Dataset (RDD)[95]. This interface provides a higher level abstraction; the application does not handle blocks but interacts with the whole RDD, and the distribution of data is managed by the framework. With this mechanisms, Spark does address the issues of blocking and data distribution, but restricts the application developer to their programming primitives as well as to the data structures provided by the framework. This requires the application developer to adapt their code to the new data interface and primitives.

Dask offers certain mechanisms to ease the management of chunk size, but does not hide the blocks themselves. However, the documentation does acknowledge the performance pitfalls due to the relationship between data granularity and overhead (*Select a good chunk size*, on Dask documentation *Array > Best Practices*[7]):

> A common performance problem among Dask Array users is that they have chosen a chunk size that is either too small (leading to lots of overhead) or poorly aligned with their data (leading to inefficient reading).
>
> While optimal sizes and shapes are highly problem specific, it is rare to see chunk sizes below 100 MB in size. If you are dealing with float64 data then this is around (4000, 4000) in size for a 2D array or (100, 400, 400) for a 3D array.

> You want to choose a chunk size that is large in order to reduce the number of chunks that Dask has to think about (which affects overhead) but also small enough so that many of them can fit in memory at once. Dask will often have as many chunks in memory as twice the number of active threads.

One of the main mechanisms offered by Dask for managing the data granularity is the *rechunking* mechanism, usable through the `rechunk` method in the Dask Array data structures. By reestructuring the data, it is possible to set up a more adequate block size (however, note that the optimal block size will depend on a variety of factors such as the algorithm complexity, the numerical implementation of the tasks and the available computational resources during runtime). With this in mind, we can consider the `rechunk` method a competitor of contribution **C2**; this competitor will be evaluated and compared to our implementation in Chapter 5.

2.2.5 Multi-layered memory tiers

The presence (and abundance) of several different memory tiers is something that can be exploited. Each memory tier has its strength and its weaknesses (typically, a trade-off between cost, bandwidth, latency). The dataClay object store already leverages RAM (for execution and caching) and disk (for persistence); our contribution **C3** addresses the additional non-volatile memory layer and how to use it efficiently.

When there are multiple memory tiers, it is possible to use multiple strategies for combining them. Even going so far as ignoring certain layers –an example of this is the RAMClouds infrastructure [64]. This is a high performance storage system that achieves great scalability and very low latency by only using the DRAM, ignoring the persistent storage tiers. This is an approach that is made possible by the growth of available DRAM in storage hardware.

Data Elevator [31] is an example of a software library that attempts to address the variety present in the storage hierarchy. The library performs the movement of data between the different memory tiers, and it does so in an efficient way that is transparent from the application point of view.

The appearance of non-volatile memory devices created new research lines and also rejuvenated a whole lot of existing ones. For example, Fan *et al.* propose the *Hibachi cache* [98] and discuss the improvements of a cooperative cache (which uses DRAM and NVRAM) between different memory tiers. Hermes [53] is an example of a buffering system which is aware (and takes advantage) of all the different tiers in the storage hierarchy – including NVRAM. In the distributed file systems field there have also been some efforts on improving distributed file systems thanks to the new NVM tier, a topic that has been explored and evaluated by Herodotou and Kakoulli [43]. A theoretical approach combining the three memory tiers (DRAM, NVM and SSD) for database systems is discussed by Alexander van Renen *et al.* [85].

All those research topics focus on a low-level integration, with special highlights on the caching policies of those systems and their performance from within the storage system. Our main focus is on the higher level integration, specifically on how an *active* object store would be able to take advantage of, first of all, memory and disk, and optionally the NVM tier. Ultimately, this increases data locality by reducing data movements to a strict minimum.

Chapter 3

Software Foundation

This thesis describes a series of contributions with a focus on the object store. Note that the ideas behind our contributions are not restricted to the dataClay object store. For instance, in Chapter 5, the contribution discussed there is implemented and evaluated with both dataClay *and* Dask.

In this chapter we will start by presenting the dataClay object store. All contributions of this thesis have been implemented with and around dataClay in order to validate and evaluate them. We will also introduce here the two execution frameworks (the ones that we will use in the evaluation sections): COMPSs and Dask.

The first execution framework, COMPSs, is a task-based distributed framework. Part of our first contribution **C1** revolves around the integration of dataClay and COMPSs. Our contribution on improving the iteration and execution across distributed datasets (contribution **C2**) will be built upon the COMPSs & dataClay integration. The state of the art integration between the two is presented in this chapter.

The other execution framework that we will present in this chapter is Dask. The contribution **C2** can also be applied unto Dask, which is a different infrastructure to COMPSs & dataClay. This showcases that the design and contribution ideas are generic, and they can be used with other software environments –not restricted to dataClay and COMPSs.

3.1 dataClay

The storage framework that we will be using in this thesis is dataClay, a distributed object-oriented data store[57]. This object store leverages object-oriented paradigm philosophy; that means that the notion of *object* follows the definition of the object-oriented programming (i.e. an abstraction that encloses data *and* code) as opposed to the objects in object stores (which are typically schema-less packs of binary data).

From the application point of view, the “database schema” (for a lack of better term) can conceptually be described through an UML diagram as the example shown in Figure 3.1.

From the developer point of view, the data model is defined through regular classes. Following the example in the Figure 3.1, the actual implementation would include code for

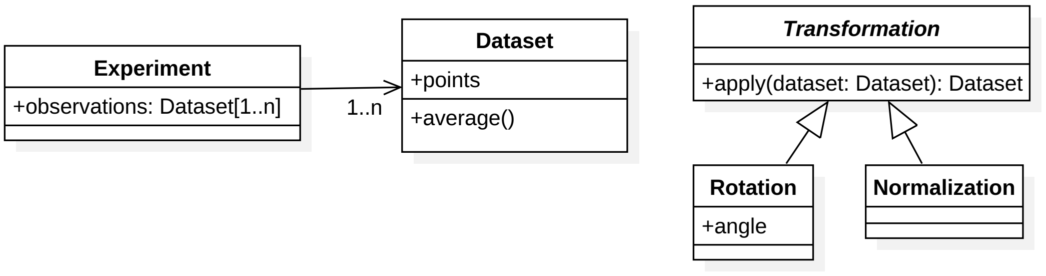


Figure 3.1: UML diagram of some sample classes

classes for `Experiment`, `Dataset`, `Transformation`, `Rotation` and `Normalization`. The class definition would be done in the native language and dataClay would use that information for the internal representation of the objects. The sample Java definitions are shown in Listing 3.1.

```java
import java.util.List;

class Dataset {
    public List<Double> points;

    public double average() {
      // Implementation of the method
      // ...
    }
}

class Experiment {
    public List<Dataset> observations;
    // ...
}

abstract class Transformation {
    abstract void apply();
}

class Rotation extends Transformation {
    public double angle;

    @Override
    void apply() {
        // Implementation of the method
        // ...
    }
}

class Normalization extends Transformation {
    @Override
    void apply() {
        // Implementation of the method
        // ...
    }
}
```

Listing 3.1: Java definition of the sample classes

3.1.1 Active features

A main feature of the chosen object store –a dataClay feature that is key for building our contributions– is its *active* capabilities. This mechanism enables the execution of code from inside the object store; that code will be able to access, modify and/or process objects residing within the object store.

We will refer to the function being executed in the object store as the *active method*. In order to have a flexible and versatile solution, those methods should be application-specific. This implies that there should be some code-shipping mechanism that allows the application developer to send the code for those *active methods* to the storage system space; we will consider a registration mechanism that happens on application initialization. The aforementioned strategy is not the only way of achieving our goal, but we believe that doing this *active method* registration on application initialization is a reasonable approach: the *active method* code definition/registration can be application-driven and this process is performed outside the execution critical path –i.e., it does not impact application performance.

During application execution, after the *active methods* are registered, the application can invoke those methods. This resembles a Remote Procedure Call mechanism; but the gist of this execution (the *active* feature) is that the object store will have the objects directly accessible and thus will be able to access them efficiently and execute the code on them, and then return the result –this, in most cases, avoids the needless serialization, transfer and deserialization of the whole object.

The *active methods* –the code executed by the active storage system– will typically be a fairly precise and narrow data-centric code. Note that the characteristics of the shipped code are merely a guideline, as the discussion in this thesis imposes no constraints regarding the complexity and shipping of said code. As a rule-of-thumb, any function that is data intensive is a good candidate to be run as an active method –as that results in an improvement due to reducing the bandwidth usage. Or, focusing on execution time: if serialization and data transfers is a high percentage of total time, it is beneficial to add locality improving mechanisms –such as the *active* methods– into the application as to eliminate this portion of execution time.

Example

Let's consider a common and simple data intensive operation: an average over all points from a dataset. The pseudocode for performing it is shown in Listing 3.2. In this snippet we can see a certain `dataset` object containing a set of `points`. If `dataset` is a persistent object (in any store platform) then the operation `dataset->points` results in a data transfer, as the set of points must be transferred from the storage to the caller.

```
dataset = new Dataset()
...
foreach p in dataset->points:
  roll_avg += p
  n += 1

result = roll_avg / n
```

Listing 3.2: Performing an average over a dataset

This operation is a perfect candidate for an *active method*. The operation processes a lot of data (it must go over the whole dataset) but it has no input and the output is small. An `average` RPC operation will thus be small. With dataClay, we can easily achieve this remote execution by including the `average` definition as a method into the data model, i.e. the `Dataset` class. The result can be obtained through a `dataset->average()` call –which, internally, would perform a RPC invocation, evaluate it remotely, and return the final result.

There is no negative impact on programmability: the code is the same, but it is in another place. In fact, having methods (such as this `average` example) in the class definition result in a more cohesive object oriented programming pattern, which increases readability and eases code maintenance.

3.2 COMPSs

COMPSs framework[11] provides a task-based programming model that can be used for the development of distributed applications. We have chosen it because it has a powerful and versatile scheduler as well as a runtime that is able to exploit the parallelism of applications.

Applications developed with COMPSs resemble sequential applications. This is by design: sequential programming is easy and a programming model focused on sequential development has a welcoming learning curve. In this thesis we will be using PyCOMPSs[79] (the Python bindings) and the annotation of code is done as shown in Listing 3.3.

```
@task(data1=INOUT, data2=IN)
def add(data1, data2):
  data1.append(data2)
```

Listing 3.3: Use of `@task` decorator for defining PyCOMPSs tasks

We can see that adding the `@task` decorator to a user function converts it to a task. Once the function is called, the information of the invocation is sent to the scheduler –which results in an asynchronous operation from the point of view of the application. In addition to the asynchronous invocation of tasks, COMPSs will also manage the dependencies between tasks. Additional information of the parameters is given by the application developer through the decorator (as shown in the example snippet, with more details available in the COMPSs documentation [22]).

3.2.1 dislib

A relevant project, developed by the same research group as the one behind the COMPSs framework, is the `dislib`[4] library. The applications shown in Chapter 4 and Chapter 5 will use it. It is a distributed library implemented on top of the PyCOMPSs programming model. In addition of it being already integrated with COMPSs, there are two main aspects that have driven us to use this library: its data model (i.e. the *dislib array*) and the algorithms that it implements.

First of all, let's look at the data model: the *dislib array* data structure –a two-dimensional array. In general, these data structures (two-dimensional arrays) are the most widely used in scientific applications and machine learning. The *dislib array* has a set of functions and facilities that eases working with them –without hiding the underlying blocks. The interface of the *dislib array* resembles the interface offered by the commonly used *numpy array.*

The other key aspect of *dislib* in the context of this thesis is the distributed algorithms implemented in this library (like k-means, Cascade SVM or k-Nearest Neighbors, all of them used for evaluation in later chapters). Its focus is on machine learning algorithms, and it is greatly inspired by scikit-learn.

3.3 COMPSs & dataClay integration

The integration between COMPSs and dataClay has existed for a while, and its architecture has been discussed in previous literature[58] (see also Martí's thesis[57] and/or dataClay source code[18] for more implementation details).

At a high level, the integration takes care of interacting with dataClay objects. Note that COMPSs is a distributed execution framework with transparent dependency management –including read and write dependencies. Managing dataClay objects means that dataClay object dependencies should work seamlessly from the point of view of the developer.

To achieve the integration between the two frameworks a new interface was defined, the Storage API. This architecture is shown in Figure 3.2. The depicted architecture shows how the application can access the objects (just as it would do normally) but also shows that COMPSs can interact with the storage layer on behalf of the application –and this allows COMPSs to manage dependencies. The main example of this integration would be the implementation of the renaming technique[56], which results in new versions to be created before write operations as to avoid Write after Read (WAR) data hazards. This implementation (in the COMPSs & dataClay context) is done through a `newVersion` function in the Storage Runtime Interface. COMPSs calls this method when the dependency management detects the data hazard and dataClay creates a copy of the object as to break the anti dependency.

Other functions in the Storage Runtime Interface include location querying; these operations are used by locality-aware COMPSs schedulers.

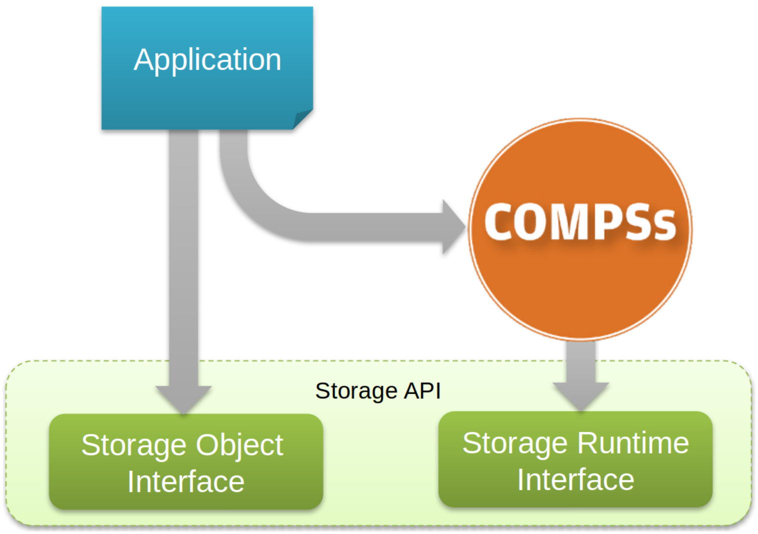


Figure 3.2: Integration architecture with COMPSs through the Storage API

3.4 Dask

Dask is (as they self-describe) a flexible library for parallel computing in Python[8]. It focuses on the data structures and primitives required to perform the parallel computing (both intra-node with shared memory as well as inter-node with network communication enabled by the Dask distributed library[6]. It has a flexible full task scheduling as shown previously in Figure 2.1.

3.4.1 Storage layer

Dask framework is not a storage system and it does not provide any kind of object store interface. It does provide some helper functions to stage-in and stage-out data from persistent storage (for instance, it can transparently read datasets from Amazon S3 or Google Cloud Storage, and it can also ingest HDF files). But Dask data structures are immutable and their lifecycle is limited to a single application execution.

However, if we look at data distribution, we can observe that Dask does distribute objects across its workers. This is done as means for performing distributed computation.

Internally, Dask leverages the native representation of its objects. This is satisfied as long as there is no memory pressure; until then, objects are held on each worker in a buffer referencing the objects in memory. When there is memory pressure, Dask spills the objects unto disk. This general behavior is very similar to the one used by dataClay.

3.4.2 Computation layer

Each Dask worker can be instantiated in a different node and each of them can have a set of objects; this is synonym with having the dataset distributed across nodes. From this starting point, Dask runtime can achieve data locality on computation stages by scheduling the tasks to where data is.

Dask ecosystem includes commonly used algorithms; they are found either built-in (for example in the Dask Array class) or in Dask-ML[27]. However, Dask also offers lower-level primitives to achieve parallel computation. Application developers can use two different programming interfaces, both supported by Dask: `dask.delayed` or (the more flexible and less handholded) `concurrent.futures`.

From a high-level point of view, the two approaches are equivalent and can both be considered task-based programming models; they are also equivalent to the COMPSs programming model. This does not mean that they are interchangeable from the implementation point of view, but they are similar enough from the programming model perspective.

In general, the resulting execution flow that we get from Dask applications is very similar to the approach that we get when executing COMPSs & dataClay. There are several key differences (the more relevant ones, as already mentioned, are that Dask is limited to immutable objects and that it does not address the persistence of data).

Chapter 4

Conditioning dataClay for HPC

In this chapter, we will discuss the steps that were required in order to suit dataClay software to the HPC ecosystem –what constitutes our first contribution **C1**. The original Java implementation, albeit perfectly functional, proved to be impractical for certain HPC use cases. One language that is being adopted by a lot of HPC applications and is very rich in terms of libraries and ecosystem is Python. This situation, combined with the already existing Python support in COMPSs side –i.e. PyCOMPSs[79], its Python bindings– made using Python a natural course of action.

Once the final architectural design of the dataClay and PyCOMPSs has been discussed, this chapter also includes an assessment of this result through three scientific applications. The analysis is both qualitative –by discussing the source code modifications– and quantitative –by evaluating performance and impact of the *active* features of the object store.

4.1 Python support

As already stated, adding support for Python –in the context of HPC– was a natural decision:

- There is a large ecosystem of scientific Python code focusing on HPC –be it data mining, big data analysis[77], artificial intelligence, machine learning[70], etc.
- COMPSs already had Python support.

In the previous chapter we mentioned dataClay, an object store with active features. Storing objects is a language-agnostic process, but defining active methods is clearly language-specific: the application developer must provide the source code (or shared object or intermediate format... it must provide *something*) for the methods. With this in mind, we can extend the general definition of a *backend* (which, in a storage system, means something that holds/stores objects) into a concept with a broader sense (it also *executes* the active methods).

The most robust approach is to have language specific backends. In the scenario we are discussing, this means having a Python backend; such backend will trivially support Python methods and Python objects. Of course this also means that there is the need to implement the Python backend –instead of, say, a more lightweight language-specific binding for the client. To sum up, having a Python backend has the following consequences:

1. The server leverages the native representation of objects.
2. The active methods are written in the native language and executed in dataClay space.

And, in our case, this is implemented by having (on server side) Python objects in memory and execute Python methods in a Python interpeter.

In this section we will give more insights in the synergies and constraints that we have considered while designing and materializing the Python backend.

4.1.1 Third-party library support

It is not enough to simply "support Python". We must support the ecosystem of code and libraries that are used in Python and HPC –and that means supporting third party libraries. The ecosystem of HPC libraries is very unique because Python is a "slow" interpreted language –but that is not relevant for the HPC context: nobody in the Python community, much less in the HPC community, expects to find a pure-Python implementation of, say, a matrix multiplication. The third-party libraries will perform all the heavy-lifting, and most of those libraries will be using optimized low-level implementations done in C, C++, or Fortran.

Up until this point we have been referring to Python in a generalist way. If we wanted to be precise, we should differentiate between **Python** (language reference[68] and standard library[69]) and **CPython** (the reference implementation of a Python interpreter using the C programming language[67]). Making this distinction is relevant at this point, as other implementations may have advantages for certain scenarios; unfortunately, most third-party libraries are implemented with CPython in mind, and other interpreter implementations tend to have limited support for libraries that require external linked libraries and/or are implemented in the C language.

With that in mind, CPython is the one that we will be supporting, and thus, dataClay backend will be done with CPython in mind. From here on we will be using the term Python in the broader sense (language, interpreter, implementation).

4.1.2 Multithreading

Another unique behavior of Python, when compared to other programming languages, happens during multithreaded operation. Python does have multithreading support; however, the Python interpreter[1] cannot interpret in parallel. At the time of writing this document,

[1]Technically, that is specific to the *CPython implementation*, which is the one that we will be using.

Python uses a GIL (Global Interpreter Lock). This GIL ensures that only one interpreter thread is working at a time, and that lock is global (hence its name, Global Interpreter Lock). The rationale behind the GIL is rooted in some implementation details, specially in the reference counting style implementation of the garbage collector. An in-depth discussion on the GIL, its implication, and the pitfalls of removing it has already been done by Larry Hastings in PyCon 2016[42].

Fortunately, the GIL is not an issue for most critical paths in HPC. There are multiple situations where having a GIL does not matter, and the ones that are relevant for dataClay and HPC are:

All networking Any socket sending or receiving data will release the GIL. Any wait for data on a socket will release the GIL. This means that using threads for networking results in the parallelism one would expect.

All I/O Similarly to the network, all read and store operations will release the GIL. This means that for I/O-heavy routines, multithreading will effectively parallelize those operations.

***Most* C code** For Python libraries that have C implementation, there are low-level directives to release and reacquire the GIL. We can assume that widely used libraries are mostly well-coded and release the GIL in all the appropriate points.

The existence of the GIL is something that caused concern for implementing the dataClay backend in Python. Being aware of GIL issues is a burden that one must diligently carry while benchmarking applications. All the design that we discuss in this thesis has taken that into account, and all the evaluations have been done in such a way that the impact of the GIL is nullified or minimized. Fortunately, most mainstream Python libraries already do a good job on managing and sidestepping the GIL, so the applications shown in this thesis did not require intervention in that regard (with a notable exception that will be explicitly discussed later in 4.3.3).

4.1.3 Network library

A stable framework, widely used in Java applications, and originally used in the first dataClay release, is Netty[81, 58]. However, when trying to port the communication protocol to Python we observed that Python support was poor or nonexistent and reimplementing it from scratch would result in a long and bug prone process with no benefit.

In the HPC arena there are specific network protocols that may be used instead. Mercury[76] is one of such projects, a Remote Procedure Call framework self-described as *specifically designed for use in High-Performance.* MPI I/O is another example of a data-centric approach, a high-level interface that aims to standarize the parallel I/O operation description (as its name suggest, in the context of an MPI application).

A different alternative, widely used in the cloud and microservices arena, is gRPC[41]. It has very good multi-language support (including both Java and Python) and has very clear semantics on the operations (the Remote Procedure Call itself) as well as on the payloads (described through Protocol Buffers[38], a language-neutral syntax for describing and serializing messages).

The final choice was to use this gRPC library; although it may not yield the most performance amongst all the available options, its strengths are its robustness and its high versatility. Moreover, its popularity (at least on the cloud arena) means that there is an extensive quantity of documentation and development resources.

4.1.4 Schema and typing

Objects that are defined and stored in dataClay have schema. The schema used by dataClay is, in general, very aligned with the object oriented paradigm of the underlying programming language. From a database-centric point of view this is loosely defined, as there is not a set of supported database types but instead an effort to support the types that the language can define. From an object oriented point of view the explanation is straightforward: the attributes of the class have types, and those are language types.

The starting point, the Java implementation of dataClay, comes naturally because Java is a *strong statically typed* language. Python has strong types too, but it uses *dynamic typing*. There is the need to augment the class definition to include static information about the class attributes types –as this is needed for optimization related to serialization and disk persistence of objects.

In this subsection we will discuss the “classic” approach, which is the one that we designed and developed for the first dataClay Python bindings. This “classic” type declaration has been the stable one for a while. Given that Python has evolved a lot in those years, we also follow with a discussion on the main alternative (the “modern” approach) for including typing information going forward.

Python 2.7 compatibility

Historically, Python did not include any mechanism to declare or define typing in the source code. The language standard and the best practices changed a little bit with Python 3 and certain language features (which we will discuss later) but first we have to consider Python 2.7 and the “legacy” solution to typing information.

Although Python does not need nor use static type information, developers do. Code documentation would, typically, discuss types. For instance, the Google Python Style Guide[36] (in 2012) recommended to include typing information in the documentation of parameters and return values.

With that in mind, we can establish that even if type information is not mandatory, having it is not an “anti-Pythonic” practice. However, in the Python 2.7 era, there was no standardized mechanism for putting that information in the source code. Using comments was ruled out; there are precedents of that in certain documentation styles, but dataClay needs type information during runtime –and accessing comments from the runtime is not a common idiom. We settled for using the *docstring* –this field purpose is documentation, so we propose to augment it. This field is easily accessed from runtime and is present in classes. To ease parsing of this data, the `@ClassField` token was accorded and the final syntax is shown in Listing 4.1.

```python
class MyClass(DataClayObject):
    """
    @ClassField <name of variable> <type>
    """
    ...
```

Listing 4.1: Syntax for including type information in the docstring field

Python 3+ and going forward

One of the breaking features of Python 3 is the *function annotations* syntax (PEP 3107[93]). The immediate application for this new syntax was not defined; some years later it was proposed to use it for including *type hints*. This PEP 484, the one to describe the usage of *type hints*[86], gained traction amongst the community. An example of its impact can be seen in the Google Python Style Guide, which at first recommended to put type information in the description and documentation[36] while now recommends to familiarize oneself with the PEP 484 instead[37]. Using type hints does not result in a statically typed language –Python still uses dynamic typing– but they are appropriate for improving code documentation and they can also be used by development environments in order to aid the programmer.

This new syntax opens the door to a community-approved way of including type information. The catch is that the syntax is only compatible with Python 3. But it has been several years since Python 2 has reached Enf-Of-Life, so using Python 3 and making use of this feature is a good idea in order to move forward.

Listing 4.2 shows an example of the classic dataClay way of defining types in classes and functions. The follow-up Listing 4.3 shows the same dataClay class but making use of type annotations.

```python
class Vehicle(DataClayObject):
  """
  @ClassField n_wheels int
  @ClassField brand str
  """
  ...
```

Listing 4.2: Classic typing information on a dataClay class

```python
class Vehicle(DataClayObject):
  n_wheels: int
  brand: str
  ...
```

Listing 4.3: Modern (Python 3) typing information on a dataClay class

Both code snippets show a `Vehicle` class definition, and the “semantic amount of typing information” is the same. In both cases, the programmer intent is clear. But there is a huge

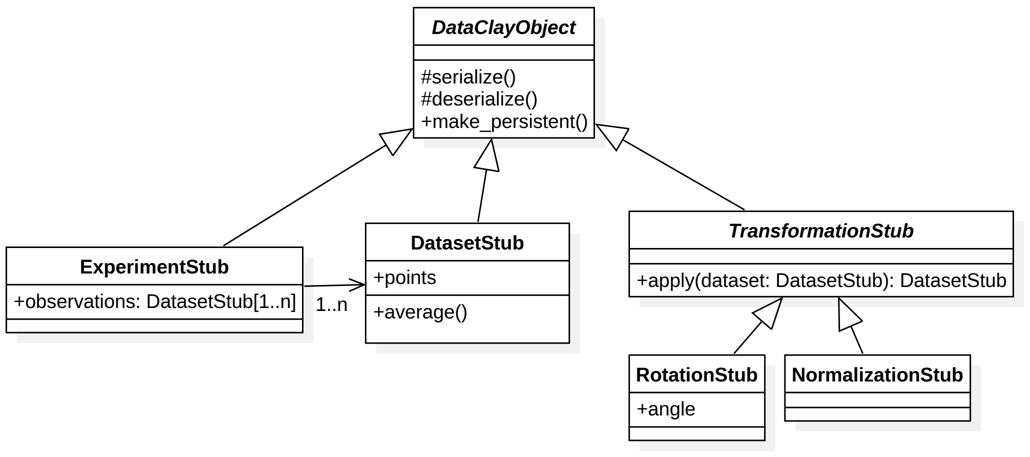


Figure 4.1: UML diagram of the stubs of the sample classes

advantage of using the Python 3 type hint syntax: a lot of Python developers are already using it and there is a lot of community effort towards it. Using the "classical" approach is more of a technical debt.

This has become the official approach starting on the dataClay 3.0 release. This release, besides dropping support for Python 2.x, focuses on bringing a more Python-friendly development cycle –which includes the use of the "modern" Python 3 syntax. Among the benefits of those changes we should mention the readability improvement, as well as an increase on compatibility with existing code. From the performance standpoint it will have no further repercussion, and the programming model impact is also negligible as the amount of typing information remains constant.

The rest of this thesis is based on the stable dataClay 2.x.

4.1.5 Language-agnostic stubs

Stub classes can be described as follows[58]:

> Stub classes (or stubs) implement the external schema representing the particular vision of a user for the classes registered in dataClay, having one stub per accessible class (...)
>
> [S]tub classes also contain the implementation of two private methods: *serialize* and deserialize.

If we consider some sample classes as previously shown as an UML diagram in last chapter (Figure 3.1), then we can consider the equivalent classes but in their stub form (Figure 4.1).

Although we show the stub classes, this is a technical representation of the *modus operandi* within dataClay. Philosophically speaking, the application developer cares about their original classes and the process of class registration should be as transparent as possible.

In the original implementation, everything was done through bytecode manipulation, and the information about classes was done through the classes themselves (and, extra metadata, private attributes). This is a good approach for Java, but is not versatile and cannot be easily ported to other scenarios. To address those shortcomings we require a different design, an approach that can be used by different languages with full semantic support for all the use cases.

We call this final design the *babelstubs*. This design is based on having a *meta* stub capable of containing all the metadata in a language agnostic way. That is a very fundamental but powerful abstraction. Moreover, making this abstraction layer tightly integrated with the metadata management present in dataClay could result in seamless transition between class metadata (language-agnostic in nature) and the class implementation (language-specific by definition).

There are several constraints in the language used in this *babelstubs*. They should be able to represent at least a UML –because they should include all the information and interaction between classes, methods and typing. This results in a non-trivial graph with different node classes. And in addition to that, the stubs must contain extra information about user information, visibility, access control and so on. The machine representation of the *babelstubs* should be able to accomodate these restrictions, and for that *babelstubs* are represented through YAML.

```yaml
Dataset: |
  !!es.bsc.dataclay.util.management.stubs.StubInfo
  applicantID: !!es.bsc.dataclay.util.ids.AccountID \
             '44991d8e-1f7e-481a-a348-23fbaa9d1390'
  classID: !!es.bsc.dataclay.util.ids.MetaClassID \
             '8d088f80-308e-4ffa-9d72-4c8ba0f6a5c2'
  className: Dataset
  contracts: !!set {...}
  implementations:
    average: &id005
      !!es.bsc.dataclay.util.management.stubs.ImplementationStubInfo
      className: Dataset
      ...
  implementationsByID:
    c72e6044-6b87-4924-b8ab-ad77a80a705c: *id005
    ...
  properties:
    points: !!es.bsc.dataclay.util.management.stubs.PropertyStubInfo
      ...
  ...
```

Listing 4.4: Example of a (stripped down) *babelstub* YAML file

Listing 4.4 shows a sample extract of the content of the *babelstubs* file. YAML is a human-friendly data serialization language for all programming languages. There are excellent and stable libraries for both Python and Java, and the YAML files are relatively easy to read and understand –something handy during debugging. It supports anchors and references, so it can express a non-trivial graph (a requirement, as it should represent a UML at the very least). In the example: the `ImplementationStubInfo` described in lines 10-13 is then referenced in line 15. It supports user-defined tags, which allowed us to seamlessly integrate it with the metadata management classes already defined in the Java codebase. The Java package that contains all the metadata classes is `es.bsc.dataclay.util.management.stubs` and we can see how the YAML references the different types (lines 2, 3, 5, 11, 18).

4.2 COMPSs integration

The first step towards PyCOMPSs integration has already been discussed in the previous chapter –in Section 3.3–: the integration between COMPSs and dataClay for Java applications. However, when developing Python applications, the integration required between PyCOMPSs and dataClay varies and requires some additional attention.

The initial design for this integration was already established (prior to this thesis) for PyCOMPSs as "Support for Big Data"[79]. This integration was established by defining an interface, called the *Storage API*. COMPSs runtime uses that interface in order to support a pluggable storage system. In this section we will review the relevant aspects that impact the design and implementation of the *Storage API* from the point of view of the object store.

4.2.1 Python translation

There are several routines that already existed in the dataClay Java codebase and now it was required to translate them to Python. Two aspects warrant our attention:

Initialization and finalization mechanisms COMPSs has different software parts interacting with dataClay, and they differ in their initialization requirements (from the point of view of dataClay). The initialization methods should be tailored to (e.g. adaptation to the multiple Python processes spawned in PyCOMPSs workers and to the main master Python in the master node).

API bindings The API interface already defined has some methods that are called from Java codebase (i.e. the methods that are used by the COMPSs scheduler) but there are some of them that must be implemented in the dataClay Python codebase (i.e. the methods that are called from the Python worker).

4.2.2 Interaction between decorators on methods

The `@task` decorator is used by COMPSs in order to define tasks. The `@dclayMethod` is used by dataClay to define methods that can be executed remotely, in an active fashion. Combining them should feel natural to the developer, and Listing 4.5 shows an example:

```python
class Vector(DataClayObject):
  """
  @ClassField values numpy.ndarray
  """
  @task(...)
  @dclayMethod(...)
  def eval_average(self):
    return self.values.mean()

# Main
for vector in matrix:
  f = vector.eval_average()
  ...
```

Listing 4.5: Simple application with a dataClay class containing a method that is also a COMPSs task

From a semantic point of view, the meaning and interaction of `@task` and `@dclayMethod` is quite clear: that method should be managed by COMPSs runtime and its execution should be carried by the dataClay backend (avoiding serialization). In that order.

From a technical point of view, the first (outermost) decorator is `@task` and thus it is PyCOMPSs that takes control and instantiates a task; the runtime is then responsible of scheduling it. When a worker starts the execution of that task, it will enter into the inner decorator, the `@dclayMethod` one. At this point, the dataClay library will perform a remote execution of that method.

Once described, the natural and intuitive behavior (for the application developer) is quite clear. The dataClay design requires that `@task` can be put on top of methods. Once that is guaranteed (i.e., supported by PyCOMPSs) then the design of dataClay makes sure that the decorators are compatible and the execution flow is, internally, the expected one –and the implementation is done according to this design.

After designing the necessary architecture and implementing it on dataClay, the COMPSs-dataClay integration is ready to be used for HPC Python applications. In the following sections we will discuss a series of applications and their evaluation in order to validate and characterize the performance of the dataClay conditioning and its integration with COMPSs.

4.3 Applications

This section will introduce three applications that will be used to evaluate the integration and the *active* feature performance: Histogram, k-means and k-Nearest Neighbors. Those applications showcase commonly used algorithms but, on top of that, their data access patterns and implementation idioms can be seen in a whole lot of other different data analytics, machine learning and scientific applications.

The source code for the applications discussed in this chapter is available on GitHub[21].

The main numerical library used in all of them is the NumPy library, which will be used either directly on the implementation or indirectly through libraries (`dislib` or `sklearn`).

The datasets for those applications are collections of n-dimensional points. The data structure that will be used to hold them are `dislib` native `Array` objects, a blocked data structure. Each block will represent a subset of points of the dataset and will, internally, be represented by a `numpy` two-dimensional array (with as many rows as points in the block and as many columns as dimensions per point). We will discuss the block size (the number of points per block) individually for each application in the evaluation section (Section 4.5).

4.3.1 Histogram

We will start with a very iconic and fundamental application, an n-dimensional histogram kernel. We have chosen this application because of its simplicity and because it is a good representative of embarrassingly parallel applications that are very memory intensive –the Histogram operation is able to ingest huge chunks of data and process them in a short period of time. Examples of other kernels that share similar characteristics are filtering and query operations.

The input dataset of this application (called `experiment` in the following code snippets) is a `dislib`'s `Array`[23], a blocked structure in which each block is a set of points. The embarrassingly parallel stage of the histogram evaluates a partial histogram for each block (using the `histogramdd` function of the NumPy library), and then the reduction stage merges those partial results into the final histogram result (with summation operations).

This application is memory intensive so data transfers play a key role on execution performance. First of all, let's show the baseline code (execution with COMPSs):

```python
@task
def partial_histogram(block, ...):
    return np.histogramdd(block, ...)

@task
def sum_partials(partials):
    return np.sum(partials, axis=0)

# Main application
partials = list()
for block in experiment._blocks:
    partial = partial_histogram(block, ...)
    partials.append(partial)
result = sum_partials(partials)
```

Listing 4.6: Original code for the Histogram application

With the use of *active* methods, the implementation source code moves a bit and becomes the following:

```python
class PersistentBlock(DataClayObject):
    ...
    @task(...)
    @dclayMethod(...)
    def partial_histogram(self, n_bins, n_dimensions):
        return np.histogramdd(self.block_data, ...)
```

```
# ---
@task(...)
def sum_partials(partials):
    return np.sum(partials, axis=0)

# Main application
partials = list()
for block in experiment._blocks:
    partial = block.partial_histogram(...)
    partials.append(partial)
result = sum_partials(partials)
```

Listing 4.7: Histogram application through the use of *active* methods

The code for performing the `partial_histogram` (the most data-hungry operation) is mostly unmodified, but now is declared as an *active* method. With this change, the task will be executed in the dataClay backend and thus there will be no data transfers.

4.3.2 *k*-means

The next application is the *k*-means clustering. It is a widely used clustering algorithm, either by itself or as a kernel within a bigger and more complex application. It is a good representative of machine learning algorithms, both in its semantic usage (clustering) as well as in its iterative implementation. We have chosen this application because it is based on a simple memory intensive microkernel at its core but it has certain complexity due to its reduction steps and overall iterative nature. This results in this application being a step above –in terms of complexity– when compared to the previous Histogram, which shares certain memory-intensive aspects but it is much more fundamental.

The input dataset of this application is the `dislib`'s `Array`, a blocked structure in which each block is a set of points. For each iteration, new centroids are evaluated by using the previous ones. This evaluation is done by aggregating points (which can be done in an embarrassingly parallel fashion) and evaluating the mean per each centroid (the merge stage).

This algorithm is called *Lloyd's algorithm* (sometimes referred as *standard algorithm*). Given its ubiquity and for the sake of brevity, we will refer to either the method, the algorithm, and the implementation, as simply *k*-means. This implementation is the one in the `dislib` library. It is shown in Listing 4.8. The main embarrassingly parallel task is the `_partial_sum` function. The reduction stage is encapsulated in the `_recompute_centers` call.

```
@task
def _partial_sum(row, old_centers):
    ...

class KMeans(BaseEstimator):
  def fit(self, x, y=None):
    ...
    while iterate:
```

```
    partials = list()
    for row_block in x:
      p = _partial_sum(row_block, old_centers)
      partials.append(p)
    self._recompute_centers(partials)
```

Listing 4.8: Relevant lines of the original k-means implementation (dislib codebase)

In order to take advantage of *active* methods, we will move the `_partial_sum` implementation into a method of the block class, as shown in 4.9. With this structure, the `_partial_sum` will have direct access to the block itself (i.e. in `self`). As in the previous histogram application, the strategy is to address the most data-hungry operations and avoid data transfers in their task execution.

```
class PersistentBlock(DataClayObject):
  ...

  @dclayMethod(...)
  def _partial_sum(self, old_centers):
    ...
```

Listing 4.9: Restructuring `_partial_sum` into an active method

4.3.3 *k*-Nearest Neighbors

The last application is the k-Nearest Neighbors, an implementation of the non-parametric supervised learning method[33]. The core of the algorithm is based on finding the k elements that are nearest a certain point p. The implementation has two distinct parts, named *fit* and *kneighbors* respectively, each with its own data. For each point of the *kneighbors* dataset, its k-Nearest Neighbors from the *fit* dataset are returned –hence the algorithm name. Typical implementations (e.g. the one we are using from the *dislib*, which is directly using the *sklearn* implementation) will generate tree lookup data structures during the *fit* stage and those tree lookup data structures will be used to efficiently find neighbors during the second *kneighbors* stage.

This algorithm is more specialized than previous ones, but even then, it is used in a wide variety of applications. For instance, it can be used as a method of classification (for labeled inputs) as well as a method of regression (for continuous inputs). It is also used in the computer vision field.

The first *fit* procedure takes the *fit* dataset and generates the tree lookup data structures. This general procedure is illustrated in Figure 4.2.

The latter *kneighbors* stage performs the lookup (for each input point on the *kneighbors* dataset) against all the lookup trees. Figure 4.3 shows how a single block of the *kneighbors* dataset is processed. The resulting number of tasks in this stage equals to the number of blocks in the *kneighbors* dataset times the number of tree lookup structures we have.

A final merge operation combines the partial results of the lookups into the result of the application.

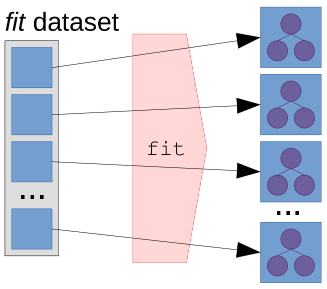


Figure 4.2: *fit* stage of the k-Nearest Neighbors application

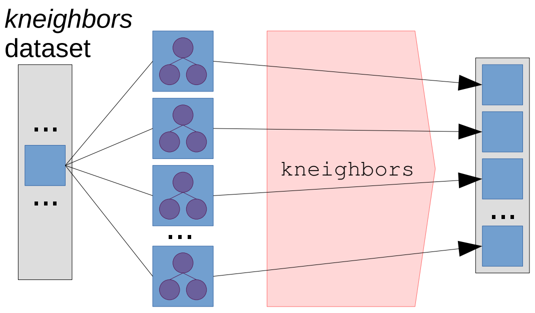


Figure 4.3: *kneighbors* stage of the k-Nearest Neighbors application (without depicting the final merge operation)

Given the complexity increase for this application, we will look with more detail into the implementation and the distribution of tasks. Let's start with the original implementation, the one available in the `dislib` library, shown in Listing 4.10.

```
class NearestNeighbors(BaseEstimator):
  def fit(self, x):
    for row_block in x:
      sknnstruct = _compute_fit(row_block)
      self._fit_data.append(sknnstruct)

  def kneighbors(self, y):
    indices = list()
    for q_row_b in y:
      queries = list()
      for sknnstruct in self._fit_data:
        q = _get_kneighbors(sknnstruct, q_row_b)
        queries.append(q)
      ind = _merge_kqueries(*queries)
      indices.append(ind)
    return indices
```

Listing 4.10: Abridged relevant code of the original NearestNeighbors implementation (dislib codebase)

We have three distinct relevant tasks:

1. `compute_fit`, i.e. the tree generation. Internally, this operation is performed by calling `NearestNeighbors.fit` from the `sklearn` library.

2. `_get_kneighbors`, which evaluates a partial k-nearest neighbors through the previous lookup trees.

3. `_merge_kqueries`, which does the final merge stage by joining (sorting and picking) partial results.

The main idea in this application is to make the *fit* data structures (the result of the first stage) persistent. The data structure is equivalent to the `NearestNeighbors._fit_data` elements seen in the original code, but now it is defined as follows:

```
class PersistentFitStructure(DataClayObject):
  ...
  @task(...)
  @dclayMethod(...)
  def _compute_fit(self, row_block):
    self.sknnstruct = sklearn.NearestNeighbors()
    ...

  @task(...)
  @dclayMethod(...)
  def _get_kneighbors(self, q_row_b):
    ...
```

Listing 4.11: Adaptation of the original NearestNeighbors implementation with dataClay

As one can see, the `compute_fit` is now an *active* method and thus the lookup data structures are populated within dataClay. Performing the lookup (what happens in the `_get_kneighbors` task, which now is also an *active* method) avoids transfering the data from the `sknnstruct` variable (attribute in the object oriented paradigm implementation on dataClay).

Parallel tree traversal

We have already discussed the impact of the Python Global Interpreter Lock (GIL) in 4.1.2. In general, its existence does not raise any issue in HPC applications.

However, this application is an exception to that rule due to the way read access is performed to the tree lookup data structures. Technically, it could be possible to implement the sklearn's KDTree implementation to be reentrant. However, this is not the current state, and concurrent access to the structure is executed in a sequential manner –something that we need to avoid for performance reasons.

To address that and be able to exploit parallelism, the implementations will include the snippet shown in Listing 4.12 inside the `get_kneighbors` task (during the *kneighbors* stage). The solution consists of duplicating the iteration data structure (not the full tree) and that is achieved by performing a shallow copy of the `NearestNeighbors` data structure followed by a copy of its `_tree` attribute (which is a KDTree instance).

```
# nn is sklearn.neighbors.NearestNeighbors
original = nn
nn = copy(original)
# nn._tree is a KDTree object
nn._tree = copy(original._tree)
```

Listing 4.12: Shallow copy of sklearn data structure to allow parallel execution

4.4 Hardware platform

All the experiments are executed in the MareNostrum 4 HPC cluster [16]. The nodes in this cluster have the following technical specs:

- 2×Intel® Xeon® Platinum 8160L CPU @ 2.10GHz
- 96GB of DRAM (12×8GB 2667MHz DIMM)
- 100 Gb/s Intel Omni-Path (between computing nodes)
- 10 Gb Ethernet (storage and management)

Each node contains a total of 48 (2×24) cores. The experiments in this chapter will be run in multiple nodes. The largest experiments will be done on a total of 16 computing nodes.

4.5 Evaluation

In this section we will be evaluating the performance of COMPSs & dataClay. The baseline is the COMPSs execution. This comparison will allow us to observe the effects of the *active* methods in different configurations. The evaluation for these applications will show the performance and scalability of the storage system through weak scaling experiments.

There is prior work evaluating COMPSs & dataClay against a COMPSs baseline (Martí *et al.*[58, 57]). This preexisting evaluation showed good results, but the focus was on the Java implementation of two applications: Wordcount and k-means. This thesis focuses on the COMPSs & dataClay integration for the Python version –once it has been conditioned for HPC, as described in this chapter– and this is the software stack that will be evaluated here.

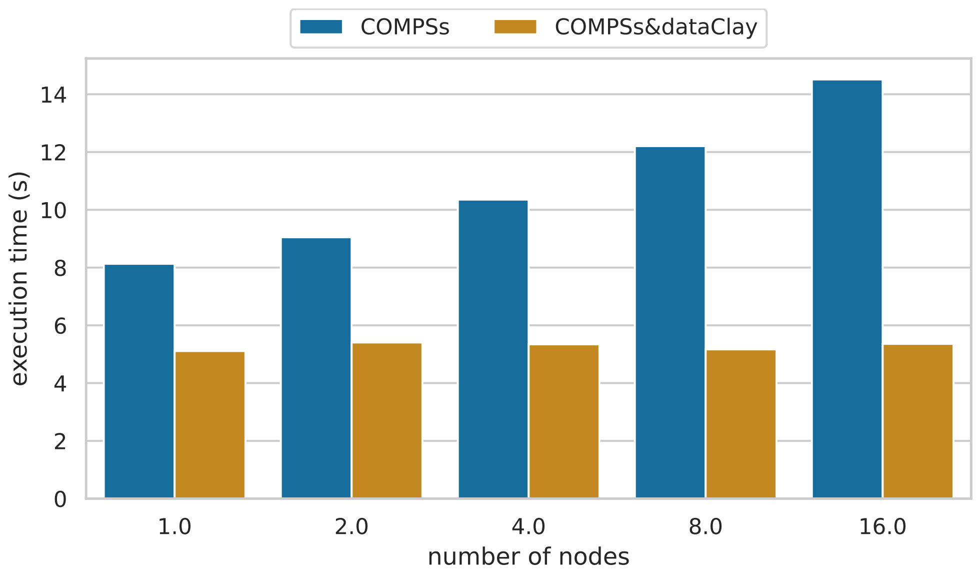


Figure 4.4: Histogram weak scaling from 1 node to 16 nodes, with 48 blocks per node (one per core)

4.5.1 Histogram

The first evaluation is on the Histogram, the most fundamental application that we will be discussing. We have to remind that Histogram is a memory intensive application (very fast execution times with a huge ingestion of data). Because of these characteristics, a big dataset size is used –however, the size has been set as for it to fit in RAM. The total memory footprint is the sum of the dataset size plus the runtime application requirements plus the framework (COMPSs in all scenarios, and dataClay is present when we are evaluating it). A size that could accommodate all our executions in all their different configurations was 880 million points (5 dimensions) per node. This results in a raw size of 33GB per node.

The scalability will consist on distributing the data and performing a weak scaling from 1 to 16 nodes. The data is divided as to have a block per core. The execution results for this settings are shown in Figure 4.4.

We can observe that COMPSs & dataClay shows good results, with a flat (ideal) scalability behavior. On the other hand, COMPSs is slower with worse scalability characteristics. The performance loss that we observe in the COMPSs executions is due to the additional data transfers and data serialization that happens on each task execution. The COMPSs implementation uses GPFS, and that degrades its performance when dataset increases (globally, a bigger dataset results in less caching and more transfers between nodes, which we observe as a degradation in scalability). In addition to that, each task requires to deserialize its input data structure, and that is a big cost for a memory-intensive kernel such as the histogram operation (which explains the performance gap that exists across all the configurations).

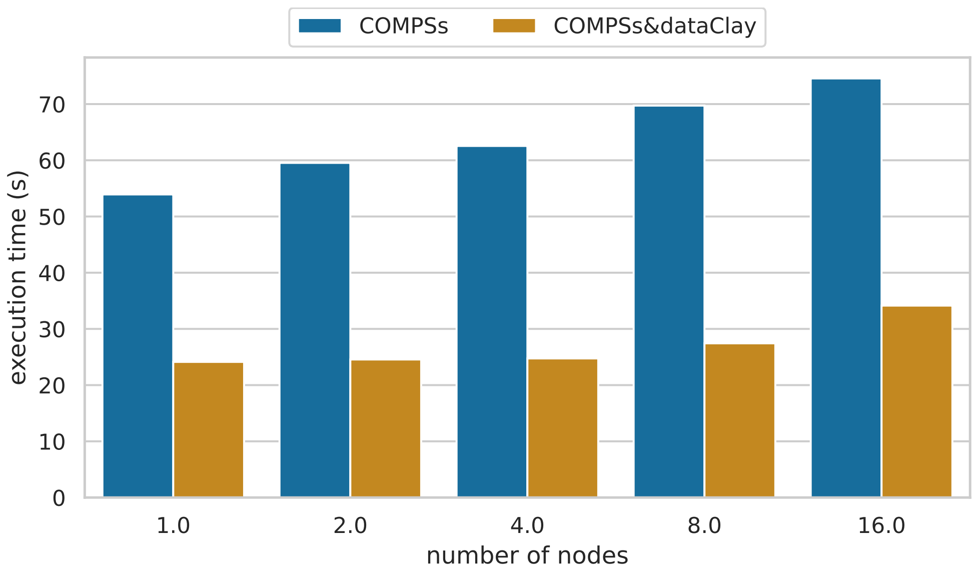


Figure 4.5: k-means weak scaling from 1 node to 16 nodes, with 48 blocks per node (one per core)

For this application we can see that avoiding data transfers is beneficial and that this improvement is more apparent as we increase the number of nodes and blocks. Due to the fast absolute execution time (around 10 seconds), the performance penalty is even more noticeable.

4.5.2 k-means

The next application is k-means, a memory bound application that is based on an iterative approach. The dataset, when compared to the previous application, is a little smaller: 11GB of raw data per node (74 million points of 20-dimensional points per node). We detected that the implementation used by the `dislib` library –in fact, the underlying `pairwise_distances` function of the `sklearn` library– is a memory hungry implementation which caused issues at bigger dataset sizes. We settled for this dataset sizes which ensures that all executions are consistent.

Once again, the evaluation will be performed from 1 to 16 nodes. Data distribution is set up to be a block per core, as to have it perfectly balanced across the available resources. The execution times are shown in Figure 4.5.

The difference between COMPSs & dataClay and dataClay execution is, more or less, 2× across all the executions. This application, albeit data-intensive, has more computation than the previous Histogram one. Another difference is the iterative nature of the application. Having several iterations increases the effect of the data locality –as the *active* methods are effectively eliminating the need of data transfers for all and every iteration.

The performance penalties that we observe in COMPSs executions are very similar

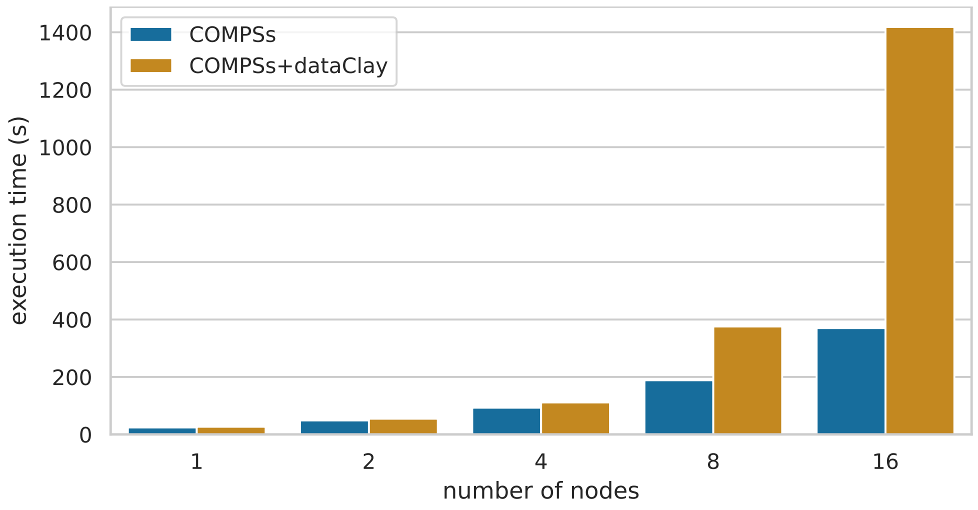


Figure 4.6: Nearest neighbors scaling from 1 node to 16 nodes

as to the ones discussed in the previous Histogram application. Once again, COMPSs is depending on the GPFS performance and is performing data deserialization for each task; we observe that this causes a degradation in scalability (COMPSs & dataClay is flatter than COMPSs). Moreover, the deserialization penalty is amplified due to the 10 iterations, which result in a high performance gap across all executions.

4.5.3 *k*-Nearest Neighbors

This application has two different datasets, both used as inputs: the *fit* dataset and the *kneighbors* dataset. In our executions, each block contains 500 thousand three-dimensional points –a manageable size, but big enough to result in over one second long single tasks. Using three-dimensional points is a natural choice for point clouds, but the implementation and our conclusions are generic and applicable also for higher dimensional points (that may be used in classification or regression).

Just as in the previous experiments, we proceed to execute this algorithm and analyze its weak-scaling performance. Figure 4.6 shows the execution time for this experiment.

The performance degradation of dataClay executions is substantial, growing up to more than $3\times$ the execution time of the COMPSs baseline. Having smaller data structures reduces the potential improvements of dataClay, while having tasks that access two separate data structures benefits the COMPSs internal flow. To delve deeper into this behavior, the following paragraph provides a more in-depth analysis.

Each block of the second dataset is evaluated against each lookup tree from the first stage. This results in a high reuse of data. The way COMPSs is configured in the cluster, those data structures are stored in GPFS and shared amongst the different nodes, which results in them being cached most times and being deserialized (effectively: replicated) in memory during task execution. When executing with dataClay, most accesses to this dataset results in a remote RPC and a serialization. The reason is the many-to-many data pattern: both lookup trees and blocks are distributed amongst the nodes, and the probability of a task resulting in same-node communication (i.e. a lookup tree and the block being in the same dataClay backend) diminishes when the number of node increases. This can be seen in the Figure with the increasing performance gap between dataClay executions and the COMPSs baseline.

This application shows the nuances when improving data locality: the best locality for complex interactions is not straightforwardly defined (at least, not necessarily *a priori*). In this case, dataClay implementation incurred in certain bottleneck due to the interactions between the intermediate data structures (lookup trees) and the *kneighbors* dataset. We see that the bottleneck becomes more and more apparent when the number of node increases, given that the number of data transfers explodes with the scaling.

The results for this last application are not ideal, but this same application will be useful to discuss the improvements of other contributions of this thesis, as we will see in the following Chapter 5.

4.6 Summary

Both the design –explained in this chapter– and its subsequent implementation constitute the contribution **C1**. The reference implementation can be found in GitHub[19] (which showcases the architecture and technical design) while the Python package is available on the public "Python Package Index" (PyPI)[20].

The evaluation shown in this chapter explores and analyzes the impact of the *active* methods across a variety of data-intensive scientific applications. In this kind of applications, data locality improvements are very important and the *active* features of dataClay are able to take advantage of data distribution during the execution thanks to the integration between COMPSs and dataClay.

We have obtained significant benefits for both Histogram and k-means, showing the potential of the *active* methods. Avoiding data transfers and being able to execute tasks where data is can reduce serialization costs and overall improve performance significantly. The good results in this applications justify the interest and potential of contribution **C1**.

However, we have also analyzed an application that degraded its performance while using *active* methods exclusively (the k-Nearest Neighbors). In this application, the specific data access pattern ended up degrading the dataClay performance when compared to the COMPSs baseline. This is due to a combination of the smaller data structures and the simultaneous data access from a single task to two separate data structures. The results suggest that just minimizing serialization and data transfers is not a silver bullet. In order to fully take advantage of the object store, the following chapters will show other improvements (contributions) that can be achieved with an object store for these and other scientific applications.

Generally, memory-intensive procedures with straightforward memory access patterns will benefit from data locality and thus they are perfect candidates for being executed inside *active* methods –which is expected, as *active* methods are effectively avoiding data transfers and serialization, executing the method where data is. More complex data access patterns –patterns that require memory transfers– can decrease the *active* method effectiveness. Similarly, increasing the computation cost of the procedures will reduce the impact of the storage system. A procedure that consumes minuscule data while being very complex in execution should not be executed where data is, it should be executed where fast processors are.

Chapter 5

Software Improvements: Enhanced Iteration

Working with distributed datasets and performing distributed execution is one of the cornerstone of HPC applications. Being able to efficiently perform this process can have a tremendous impact on overall performance. For this reason, there are a number of well-established approaches that address this –projects and research that we have briefly discussed in Chapter 2.

In all these prior work we see how the block granularity of data is an important factor –along its relationship with distribution of work. In this chapter we propose a mechanism that decouples the data distribution from the task granularity, all this while improving performance and maintaining good programmability. This is done by combining runtime knowledge on data placement with the execution scheduling. Being able to do this without having to reinvent the programming model (i.e., without requiring a full rewrite of applications nor a steep learning curve) is a novelty and constitutes contribution **C2**.

5.1 Distribution of data and computation

When using a distributed execution environment there is the need of dividing the dataset into parts (i.e. blocks). This is required in order to distribute computation across nodes. Figure 5.1 shows a simplified diagram where we can see the meaning of the block distribution. The distribution may have additional requirements, or the data structure may carry more nuisances, but the main idea is the distribution of those blocks.

This block concept may be called with a variety of terms, and may be more or less transparent to the developer –for instance, the *Resilient Distributed Dataset* or *RDD*[95] is a Spark data structure that transparently addresses the blocking needs of a distributed dataset and its distributed execution. The dislib array and the Dask array both use blocks and have a configurable and user-defined chunk size.

In task-based programming models, a usual scenario is to have a task for each block, as shown in Figure 5.2. Tasks may be more complex than that –e.g. by having multiple

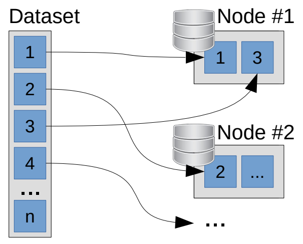


Figure 5.1: Diagram of the distribution of a dataset between different nodes

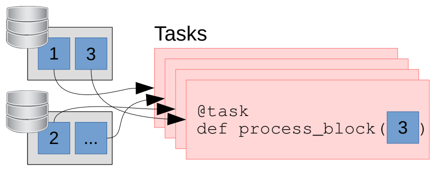


Figure 5.2: Diagram of a regular one-to-one execution of tasks to blocks (one task per block)

inputs. But, at a high level, we want to highlight the fact that the tasks will be using the blocks directly; this implies that the granularity of tasks depends on the block size. This is indeed what happens in both Dask and dislib algorithms.

Thus, applications need to define a block size, which is not trivially determined performance-wise. The level of fragmentation of the dataset will impact the scheduler overhead as well as the potential parallelism of the application. The optimal will depend on the implementation and the capabilities of the computing environment. Having a block per core is a good rule of thumb, but even that depends on the executing infrastructure (both on the number of nodes as well as on the CPU model in said nodes). The system memory and the algorithm requirements will also have an impact on the optimal block size; for instance, a memory hungry routine may result in starvation if blocks are too large. Those parameters are not necessarily known during application development time. And even if everything is known beforehand, different stages of the application may have a different optimal block sizes (due to different numerical routines, elasticity of computing nodes, latency, etc.).

5.2 SplIter

Our proposal consists on maintaining the blocks and enhancing the iteration with the *SplIter*, a software runtime mechanism that adapts both to the data placement of such blocks and to the computing capabilities of the environment. The *SplIter* works as follows: gather all the blocks that are located in a single node and yield **partitions**. A diagram showing this procedure is shown in Figure 5.3. Each partition is located in a single node, ensuring data locality, and the number of partitions (i.e. the number of tasks) is related to the computing capabilities of the environment.

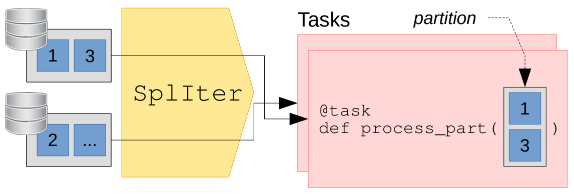


Figure 5.3: Diagram of *SplIter* interactions and task invocation

In the COMPSs & dataClay stack, the *SplIter* cooperates with dataClay in order to retrieve the location of objects and generate the partition objects (which will be also dataClay objects, linked to the same backend as the objects it represents). For Dask, we have implemented *SplIter* through the use of the different `client` methods. The *SplIter* implementation queries data location information to Dask and uses that information to both build partitions and to schedule the tasks according to the location of said partitions.

We will discuss the specific usage of *SplIter* for each application in the following Section 5.3. But before that, we want to illustrate the fundamental usage of the *SplIter* and the codebase changes that it requires. We will be using here the Histogram as a sample application. The pseudo-code for the original implementation is shown in Listing 5.1. That implementation shows an embarrassingly parallel stage, called `partial_histogram`, annotated with the `task` decorator; it also shows a merge stage –the `sum_partials`, a direct addition– annotated with the `task`. The pseudo-code used follows the general syntax of the COMPSs programming model (as shown in previous chapters), but the general idea behind this code will also be applied to Dask.

```
 1 @task
 2 def partial_histogram(block, ...):
 3     return np.histogramdd(block, ...)
 4
 5 @task
 6 def sum_partials(partials):
 7     return np.sum(partials, axis=0)
 8
 9 # Main application
10 partials = list()
11 for block in experiment._blocks:
12     partial = partial_histogram(block, ...)
13     partials.append(partial)
14 result = sum_partials(partials)
```

Listing 5.1: Sample of the original code –Histogram application

The next Listing 5.2 shows the changes required in order to make use of *SplIter*. Note that the merge (`sum_partials` function) remains unmodified; also the `partial_histogram` remains the same –albeit it is not a task now but simply a function. What has been added is the `compute_partition`, a task that processes all the blocks in a partition. The code in this inner loop (Listing 5.2 lines 12-14) is the same as the code in the original loop (Listing 5.1 lines 12-14). By design, adding the *SplIter* results in an extra loop nesting as there is now an additional iteration per partitions (lines 19-21). The number of tasks (the number of `compute_partition` invocations) is equal to the outer loop size (i.e. number of partitions).

```
 1 def partial_histogram(block, ...):
 2     return np.histogramdd(block, ...)
 3 
 4 @task
 5 def sum_partials(partials):
 6     return np.sum(partials, axis=0)
 7 
 8 @task
 9 def compute_partition(partition):
10   part_results = list()
11   for block in partition:
12     partial = partial_histogram(block, ...)
13     part_results.append(partial)
14   return np.sum(part_results, axis=0)
15 
16 # Main application
17 partials = list()
18 for partition in spliter(experiment):
19   partial = compute_partition(partition)
20   partials.append(partial)
21 result = sum_partials(partials)
```

Listing 5.2: Minimal changes on sample code to include the usage of *SplIter*

As can be seen, using *SplIter* does introduce some additional complexity, in the form of an additional loop. However, this pattern is simple enough and follows a clear semantics of the underlying abstractions: a loop for the partitions and a loop for the blocks that form the partition.

5.2.1 Tracking collection order

Up until this point we have explained the foundation of the *SplIter*. However, the mechanism that we have outlined so far loses the original ordering of the collection.

When the original collection ordering is relevant for the algorithm, we propose two different methods: `get_indexes` and `get_item_indexes`. The first one, `get_indexes`, returns the block index –so, in the example shown in Figure 5.3, it would return `[1, 3]`. There are scenarios where the application requires the global element indexing; for those scenarios the `get_item_indexes` returns the individual item indexes.

By embedding index information into the partition, the algorithm can leverage the information of global block position and global item indexes. Some of the applications presented in this chapter will make use of those methods; we will mention the details for each application discussion –i.e. for the Cascade SVM (5.3.3) and for the k-Nearest Neighbors (5.3.4).

5.2.2 Comparison with rechunk

There are several differences between the *SplIter* and the *rechunk* –what can be considered a direct competitor. The key difference is the fact that *rechunk* generates a new array: once the *rechunk* has finished, it has generated a new data structure with a different block size than the original. This contrasts with the *SplIter*, which produces logical groups of blocks (the partitions). Using the *rechunk* is simpler from the application point of view, as the result is a "standalone" array –same interface as the original dataset. But its simplicity comes with a penalty –when compared to the *SplIter*– in the form of additional transfers and data transformations.

Additionally, the *SplIter* is able to leverage the data locality by producing partitions with worker-local blocks. This additional indirection (the partition) does not preserve the order, but this is addressed as described in the previous subsection 5.2.1.

5.2.3 Implementation

In this subsection we will review the general implementation steps that are required in order to recreate the *SplIter* and we will discuss how these modifications have been introduced into COMPSs & dataClay and Dask.

First of all, the *SplIter* requires the data to be divided in blocks and distributed across nodes. The block subdivision is something that is already provided by the Dask Array and dislib Array data structures. The distribution of data across nodes is provided by Dask on one side and by dataClay in the other.

The *SplIter* implementation will query the data location of the blocks and yield the *partitions*. This procedure depends on the software stack and we will discuss them separately below. After the partitions have been assembled, we can proceed to the distributed execution step. In both frameworks (COMPSs and Dask) this will be done by invoking a task for each partition.

COMPSs & dataClay

The *SplIter* requires location information for distributed data, which is a feature provided by dataClay –the data blocks are dataClay objects. Given those persistent objects (blocks), we can use dataClay to query the location of said objects. This is the beginning of the *SplIter* implementation. With that information, the partitions are created. Those partitions are implemented as dataClay objects too –the partition is created in the same backend as its constituent blocks. This partition contains, fundamentally, the list of blocks (as node-local references, just as dataClay object references). Additionally, for the features explained in 5.2.1, the partition contains the list of indexes (index for each block), and the list of item indexes (index for each element in each block). This information is populated during partition creation.

Finally, a task that accepts the partition will be in charge of execution –task which will be invoked several times, as many as total partitions. Each task will iterate all the blocks in the partitions. This task is application-specific code. Note that *SplIter* requires no modifications in the COMPSs framework.

The partition data model (including logic and index tracking) is about 50 lines of source code. The other modification required for implementing the *SplIter* is the `spliter` function which amounts to less than 100 lines.

Dask

The Dask array is already a blocked data structure distributed across nodes –something that *SplIter* requires. Moreover, the Dask API already provides a `who_has` call which returns the location for a batch of objects in an efficient manner. As these features are already present in Dask, the *SplIter* implementation has the appropriate data structure and the query mechanism.

From this starting point, the *SplIter* implementation uses these features in order to generate the partitions. The partitions are, once again, built with the location query information. In this case, the partition references the blocks by the Dask identifier string. Each partition will contain the identifier of its objects. If indexing information is needed (as explained in 5.2.1) it is generated along the partition.

When the execution is on the worker –i.e., when the task that processes a partition is being executed– the actual object can be retrieved by using the identifier and the worker cache. More specifically, the worker cache is a dictionary of Python objects indexed by the Dask Future identifiers –the same identifiers that the partitions contain. This process (which is Dask specific) guarantees no data movements and also guarantees locality among tasks, partitions and blocks.

This implementation of the *SplIter* is built with a minimal partition structure (a 15 lines long class definition) but requires accommodating certain Dask scheduling aspects, which adds 30-40 lines of source code. The `spliter` function is only 5 lines long, but once again, 30-40 additional lines of source code are required for managing the objects within the task.

5.3 Applications

Three applications that we will be using in this chapter (Histogram, k-means and k-Nearest Neighbors) have been explained and evaluated in the previous Chapter 4. In this section, we will introduce an additional application (Cascade SVM) and review for each of them the aspects that are relevant for the *SplIter* evaluation.

The source code for the applications discussed in this chapter is available on GitHub[21].

All applications are implemented in Python. The main numerical library used in all of them is the NumPy library, which will be used either directly on the implementation or indirectly through higher-level abstractions. The datasets for all those applications are collections of n-dimensional points. The data structure that will be used to hold them are either dislib native array objects or Dask native Dask Array structures.

5.3.1 Histogram

For this application, we want to showcase the main advantages of the *SplIter* mechanism and the effect on locality –which should be a key aspect on execution times due to the memory bandwidth bottleneck of the application. This is a memory intensive application and as such we will be using *active* methods as shown previously (4.3.1). The *active* implementation has already been shown. In order to address the *SplIter* modifications we have to consider the different parts of the implementation:

- The main iteration across all the blocks –that is the main loop.
- The embarrassingly parallel task –the function `partial_histogram` which processes a single block.
- The reduction task –called `sum_partials`, which processes the partial results.

```python
class PersistentBlock(DataClayObject):
    ...  # unmodified

class Partition(DataClayObject):
    ...
    @task(...)
    @dclayMethod(...)
    def compute_partition(self):
        part_results = list()
        for block in self.blocks:
            partial = block.partial_histogram(...)
            part_results.append(partial)
        return np.sum(part_results, axis=0)

# Main application
partials = list()
for partition in spliter(experiment):
    partial = partition.compute_partition()
    partials.append(partial)
result = sum_partials(partials)
```

Listing 5.3: Implementation of Histogram application with *active* methods and *SplIter*

With this, we can look into the modifications required to include *SplIter* into the implementation (shown in Listing 5.3). The summation operation done inside this `compute_partition` is the same operation done in `sum_partials`; doing it within the `compute_partition` task guarantees data locality on this first merge operation. The `sum_partials` function is omitted given that it has no changes; the same happens to the `PersistentBlock` that suffered no modifications.

5.3.2 *k*-means

For this application, we want to follow-up on the advantages of the *SplIter* mechanism on memory intensive applications. The added complexity on the numerical procedures (the iterative nature of the algorithms, paired with a more costly reduction stage, compared to the previous application) will affect the execution times and the *SplIter* contribution.

```python
class Partition(DataClayObject):
    ...
    @task(...)
    @dclayMethod(...)
    def compute_partition(self):
        part_results = list()
        for block in self.blocks:
            partial = block.partial_sum(...)
            part_result.append(partial)
        return _merge(*subresults)

class KMeans(BaseEstimator):
    def fit(self, x, y=None):
        ...
        spl = spliter(x)
        while iterate:
            partials=list()
            for partition in spl:
                p = partition.compute_partition()
                partials.append(p)
            self._recompute_centers(partials)
            ...
```

Listing 5.4: Implementation of *k*-means application with *active* methods and *SplIter*

As discussed in Section 5.2, and similarly to the previous application, we can see the addition of a nested loop. In the inner loop (within the `compute_partition` method) we see, once again, a partial merge call (the `_merge` invocation). This function is already being used by `_recompute_centers`, and now we can see its explicit invocation within the inner loop –an invocation with guaranteed locality.

5.3.3 Cascade SVM

The next application that we will consider is a distributed SVM, or more precisely, the distributed training procedure of a support vector machine following the Cascade SVM algorithm. We have chosen this application because of its relevance as a data analytics algorithm as well as its importance in the machine learning ecosystem. The distributed implementation takes advantage of fundamental kernels (SVC) and globally it is a compute-bound algorithm. The evaluation and discussion for this application can be extrapolated to other applications with high computation requirements that start with an embarrassingly parallel stage and then have non trivial reduction procedures (e.g. mesh refinements algorithms, iterative optimization strategies, etc.).

The main microkernel of this application is an SVM itself. More specifically, the actual implementation of the `dislib` library uses the `sklearn`[65] C-Support Vector Classification (SVC). This method has a very high computation cost, with a low memory footprint. Using *active* methods has no significance, as there is little potential gains due to locality in the first stage of the application. Thus, no *active* evaluation is presented in this application; this is the reason that this application was not presented in the previous chapter.

In this chapter and for this application, we want to showcase the main advantages of the *SplIter* mechanism when applied to such a compute-bound application. This is also an application where the item ordering is relevant for the result. The implementation that we will be using (the one in the `dislib`[4] library) is based on the algorithm described by Graf *et al.*[39].

The input dataset data structure is the `dislib`'s `Array`, a blocked structure in which each block is a set of points (this structure is called `x` later on in the code snippets). Along this array there is another one (called `y` in the pseudo-code), with the same cardinality, containing the *labels* (or *categories*) of those points –the *SVM* is a supervised classifier. The Cascade SVM is an iterative algorithm, with each iteration starting by an embarrassingly parallel stage where a SVM is run in each block. The merge also consists on an SVM.

Let us discuss the original source code of the `CascadeSVM` shown on Listing 5.5 –as implemented in the dislib library. In that snippet we can see the two arrays, one with points (parameter `x`) and the other with labels (parameter `y`).

```
@task
def _train(...)
    ...

class CascadeSVM(BaseEstimator):
  def _do_iteration(self, x, y, ...):
    for blockset in zip(...):
      x_data = blockset[0]._blocks
      y_data = blockset[1]._blocks
      _tmp = _train(x_data, y_data, ...)
```

Listing 5.5: Relevant lines of the original CascadeSVM implementation (dislib codebase)

The modifications introduced in the *SplIter* implementation are shown Listing 5.6. The ordering within the collection matters, because the labels (parameter `y`) are linked to the points (parameter `x`). This is handled by using the `get_indexes` mechanism, explained in subsection 5.2.1. The main task `_train` is unmodified.

```
@task
def _train(...)
  ...

class CascadeSVM(BaseEstimator):
  def _do_iteration(self, x, y, ...):
    for partition in spliter(x._blocks):
```

```
    split_indexes = partition.get_indexes()
    x_data = partition._chunks
    y_data = [y[idx]._blocks[0]
              for idx in split_indexes]
    _tmp = _train(x_data, y_data, ...)
```

Listing 5.6: Relevant modifications for the *SplIter* implementation on the CascadeSVM application

5.3.4 *k*-Nearest Neighbors

This algorithm depends on the item ordering of the input, something that the *SplIter* has to take into account. The support comes through the `get_item_indexes` primitive offered by our *SplIter* proposal and discussed in subsection 5.2.1. The explicit code will not be shown in the snippets here (for the sake of brevity and simplicity) but the final implementation and the evaluation include it.

For this application, we want to show the full potential of the *SplIter* mechanism when applied to more complex data structures and algorithms. There will be more changes on the application –at least compared to the minimal ones shown in previous applications. However this will also unlock certain opportunities –as we will explain here and later on discuss during the evaluation, in subsection 5.5.4.

The original implementation was previously seen in Listing 4.10, and we already discussed the three relevant tasks: `compute_fit`, `_get_kneighbors`, and `_merge_kqueries`. In order to use the *SplIter* in this application we include the modifications outlined in Listing 5.7. The general flow of the application is the same, with a difference in the `compute_fit_partition`, which now applies to a whole partition.

```
class NearestNeighbors(BaseEstimator):
  def fit(self, x):
    for partition in spliter(x):
      nn = _compute_fit_partition(partition)
      self._fit_data.append(nn)

  def kneighbors(self, y):
    indices = list()
    for q_row_b in y:
      queries = list()
      for persistent_nn in self._fit_data:
        q = persistent_nn.get_kneighbors(q_row_b)
        queries.append(q)
      ind = _merge_kqueries(*queries)
    return indices
```

Listing 5.7: Implementation of NearestNeighbors with *SplIter* mechanism

In the original `compute_fit`, that task receives a block (conceptually, a set of points); on the other hand, `compute_fit_partition` processes a whole partition (which is a set of blocks, each block being a set of points, so at the end, a partition is also a set of points).

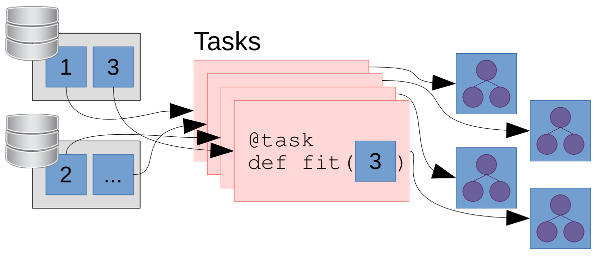


Figure 5.4: Diagram of the tree generation on the *fit* (original implementation)

This change seems minimal in the source code, but has an interesting implication: instead of generating a tree lookup data structure per input block (see Figure 5.4), we are generating a single tree lookup data structure per partition –decoupling the number of intermediate data structures from the number of blocks in the input dataset. Figure 5.5 shows the data structures resulting in the *SplIter* version of the application. Using *SplIter* in this fashion is done for two main reasons. First, it allows the implementation to exploit locality, as each tree is generated without requiring any serialization nor data transfer between nodes. Secondly, it generates more efficient lookup data structures, as having a single but bigger tree is more efficient on look-ups than having several smaller trees. Our evaluation in 5.5.4 will explore the impact of the tree sizes on number of tasks and total execution time. This efficiency increase due to different intermediate data structures show an advantages that the *SplIter* can bring us: by using partitions, the intermediate data structures can be combined and generated in a more sensible way.

Using *SplIter* is not the only way to achieve bigger trees and the evaluation will consider the *rechunk* approach as an alternative. The application developer could manage it with either hard-coding an optimal block size (which requires knowledge of the computing resources and lots of ugly hard-coding and platform-dependent code) or they could query manually the object store for the placement of objects and perform the partitions manually. The first option is an anti-pattern, and loses a lot of portability –negating the advantages of a task-based programming model. The second option is effectively reimplementing the *SplIter* ad-hoc for each application, which is less efficient and yields no advantages over having it tightly integrated into the programming model –which is exactly what we propose.

5.4 Hardware platform

The hardware configuration is the same as the one used in 4.4:

All the experiments are executed in the MareNostrum 4 HPC cluster [16]. The nodes in this cluster have the following technical specs:

- 2×Intel® Xeon® Platinum 8160L CPU @ 2.10GHz
- 96GB of DRAM (12×8GB 2667MHz DIMM)

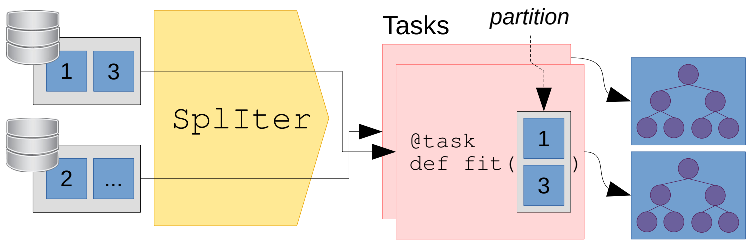


Figure 5.5: Diagram of the tree generation on the *fit* (implementation with *SplIter*)

- 100 Gb/s Intel Omni-Path (between computing nodes)
- 10 Gb Ethernet (storage and management)

Each node contains a total of 48 (2×24) cores. The experiments in this chapter will be run in multiple nodes. The largest experiments will be done on a total of 16 computing nodes.

5.5 Evaluation

In this section we will be evaluating *SplIter* performance for the different proposed applications. There will be two baseline executions, one per each framework; we will call the first one "COMPSs & dataClay" and the other one is called "Dask". These executions will be done without any mechanism to tackle granularity issues on the data –neither *SplIter* nor *rechunk*.

Besides these baselines, we will include both *SplIter* executions (one on top of COMPSs & dataClay and the other one on top of Dask). An extra additional execution "Dask + rechunk" is included in the evaluation; this configuration will issue a *rechunk* operation (see subsection 5.2.2). This configuration is used as the main competitor of the *SplIter*.

5.5.1 Histogram

The first experiments that we will discuss are performed with the Histogram application. This application was already evaluated with dataClay in the previous chapter (see 4.5.1); in this section we will be doing an evaluation using the same dataset size as in the previous chapter (880 million 5-dimensional points per node, roughly 33GB/node).

Scalability for highly fragmented datasets

We will start with a dataset divided into a large number of blocks. Our goal is to showcase the benefits of the *SplIter* in an environment where locality and data transfers are critical –i.e. during a memory intensive application execution, such as the Histogram. The block size is chosen in order to attain 48 blocks per core.

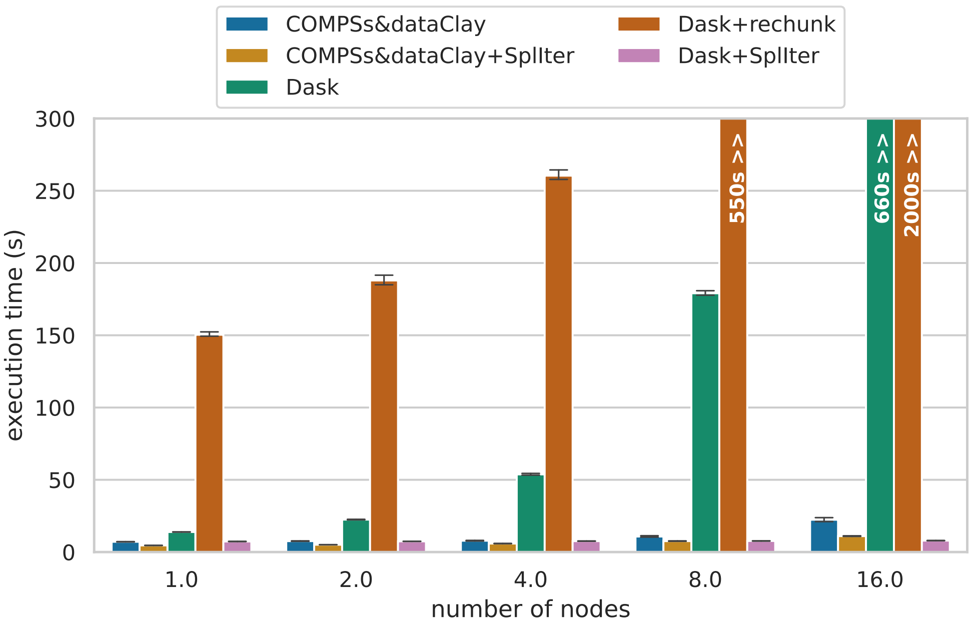


Figure 5.6: Histogram weak scaling from 1 node to 16 nodes, with 2304 blocks per node (48 per core)

The execution times are shown in Figure 5.6. The figure shows the good behavior of the *SplIter* and its satisfactory scalability. Both *SplIter* executions show similar execution times (meaning that the "quantity of work" done is similar, which means that both algorithm implementations are equivalent and the *SplIter* approach is equally effective regardless the execution framework). These executions outperform their respective baselines for any number of nodes (and the difference grows with the number of nodes). This shows us that *SplIter* is successfully reducing the scheduler and execution overhead, and the improvement increases with the number of blocks.

If we look into the scalability of the two baseline executions, we can observe that COMPSs & dataClay presents a good but not ideal scalability (i.e., the performance difference with *SplIter* grows when the number of nodes and thus tasks grows). The performance degradation is due to the scheduling and invocation costs. The scalability of the Dask baseline is much worse. This suggests that the COMPSs scheduler is able to behave better under heavy pressure.

The competitor approach, Dask + rechunk, is a mechanism that reduces the number of tasks and improves the inadequate granularity issues. However, we can see in the figure that the execution times when using *rechunk* are the worst. This happens in this kind of application due to the steep cost in terms of data transfers –a cost that shadows any improvement. The data transfers constitute a high expense in this application because the dataset size is large and the execution time is very low, meaning that data transfers become the main bottleneck instead of the computation or the scheduler overhead.

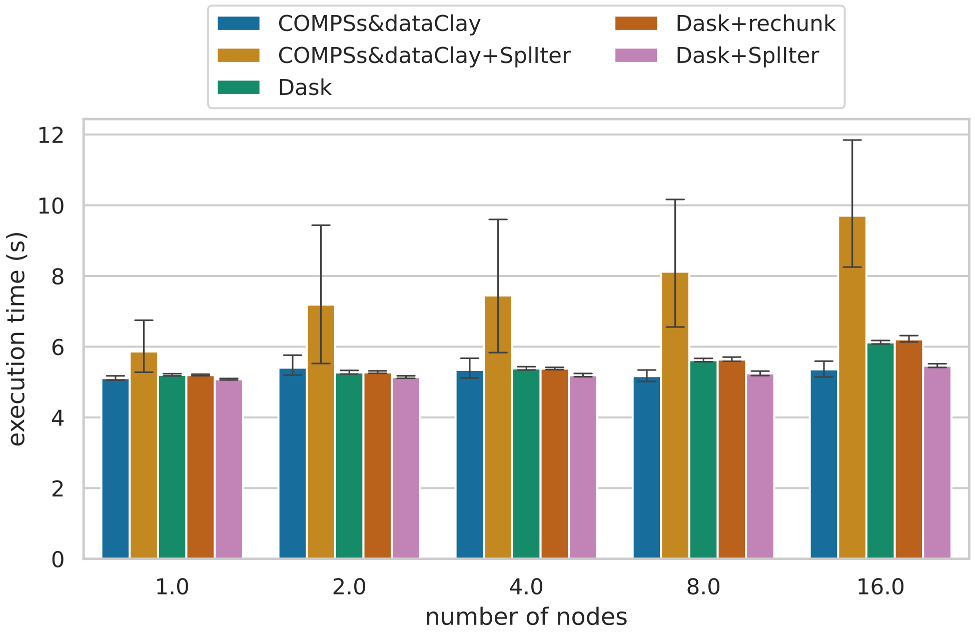


Figure 5.7: Histogram weak scaling from 1 node to 16 nodes, with 48 blocks per node (one per core)

Overhead for perfectly balanced datasets

Now we move the evaluation to the scenario were the dataset is perfectly balanced; this means that there will be a single task per core. The total dataset size remains constant (880 millions points per node) so the change from the previous experiment is the number of blocks and the block size. Now the blocks are 18432 thousand points (48 times more than the previous experiment).

This experiment presents a worst-case scenario for the *SplIter*, as a perfectly balanced dataset means that there is no room for improvement. Moreover, given that we are evaluating an application that has small absolute execution times, any penalty or additional noise will be conspicuous.

The execution times for this experiment are shown in Figure 5.7 where we can see a weak scaling from 1 to 16 nodes.

Excluding the COMPSs & dataClay + *SplIter* execution, we see that all the other ones have close execution times between them. Both the baseline and the *SplIter* are effectively performing the same Histogram operation unto the same data, and data is perfectly balanced.

In this specific scenario, the *SplIter* implementation on Dask is able to exploit data locality a bit better; the *SplIter* execution is able to exploit data locality during its kernel (just as its baseline) but the partitions are arranged divided in workers, which are able to exploit locality during the first reduction step (when partial results are merged together).

Given that executions are so fast, being able to exploit this extra of data locality is able to give some additional benefit –a surprising result, given that this is designed to be a worst case scenario for *SplIter*.

There is a slight difference among the two *SplIter*, as we can see that the *SplIter* implemented on top of Dask behaves much more consistently than the COMPSs & dataClay + *SplIter* one. Of course, we have to take into account that this execution is very fast (just a few seconds long) but still, the overhead of the *SplIter* on dataClay adds a fair amount of noise and an additional couple of seconds on average. Assembling the partition requires to retrieve location of blocks, and the overhead of that query will be different between dataClay and Dask. These results suggest that dataClay could be improved to have a faster query operation, and that would reduce the *SplIter* overhead. Dask already offered a single operation to query the location of multiple objects, which –seeing the results– is more efficient and consistent than the one we implemented on top of dataClay.

This worst-case scenario shows us that the overhead of the *SplIter* –an overhead that depends on the number of blocks– can be low. A higher number of blocks should result in a more laborious partition preparation, but the overhead can be minimal as demonstrated by the Dask execution.

Sensitivity to fragmentation

After discussing a favorable scenario for *SplIter* and an unfavorable one, we want to discuss what happens across this spectrum. This experiment will sweep the number of blocks from 1 per core (worst *SplIter* scenario, which matches the previous experiment) to 48 blocks per core (best scenario, which matches the first experiment on this application). The experiment is evaluated in 8 nodes and the dataset size is constant and the same as in previous scenarios: 7 billion points in total. The execution times are shown in Figure 5.8.

The *SplIter* behavior is quite flat, showing that it is a tool that desensitizes application execution performance from task granularity –the positive outcome that we expected. As we discussed for the worst-case scenario, the *SplIter* implementation on top of dataClay has some room for improvement (the leftmost second bar, which stands out a little bit), but it shows good and steady performance everywhere else.

Once again we see that the baseline executions degrade a little bit (starting on the left being as fast as *SplIter* and degrading progressively when the number of tasks increases. The COMPSs scheduler is much more robust against bad granularity scenarios (increasing from 5s to 10s) while the Dask scheduler suffers much more and increases its execution time from 5s to more than one hundred.

The rechunk is still a very expensive operation. When there is a single block per core, the rechunk operation is a no-op. However, as soon as the rechunk operation performs its duties (which happens at 4 blocks per core, when the rechunk is expected to redistribute data between workers) the huge amount of data that must be transferred between nodes results in hundreds of seconds of execution overhead.

We can see some slight inconsistencies between 1 and 4 blocks per core (the COMPSs & dataClay + SplIter implementation improves, while baseline executions are sitting very

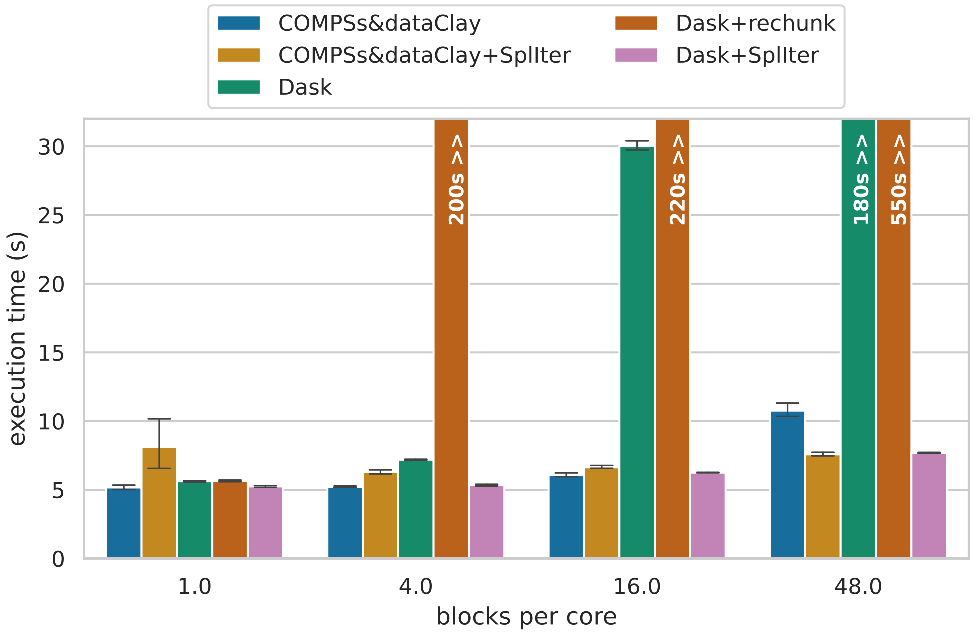


Figure 5.8: Histogram on 8 computing nodes. The X axis show variation on the total number of blocks per core. Block size changes in order for the total dataset size to remain constant.

still). There is a performance variation that contributes to this: the special behavior of the microkernel (the `numpy.histogramdd` implementation that we are using in all scenarios) on big block sizes. The numerical implementation of the numpy `histogramdd` benefits from having smaller blocks: for instance, it is 20% faster to perform 16 histograms of block sizes equal to 1152 thousand points instead of doing a single histogram of 18432 thousand points. This is taking into account reduction and independently of parallelism, just saturating all the cores and memory of a single socket. Discussing this application in this level of detail is outside the scope of this thesis, as it is related to the actual implementation on the `numpy` library and is also related to the intra-node architecture.

Insights

In this kind of memory intensive application, the *SplIter* shows a good behavior, with big improvements on favorable scenarios and low-to-no overhead for unfavorable ones. This kind of application also renders the *rechunk* approach impractical –using it results in an excess of data transfers. Also, the single-pass (as opposed to an iterative algorithm) negates the reuse potential of both *rechunk* and *SplIter* operations.

The *SplIter* has shown that it is able to address the task granularity of the data while maintaining the data locality benefits; this is not achieved by the *rechunk* for this kind of applications.

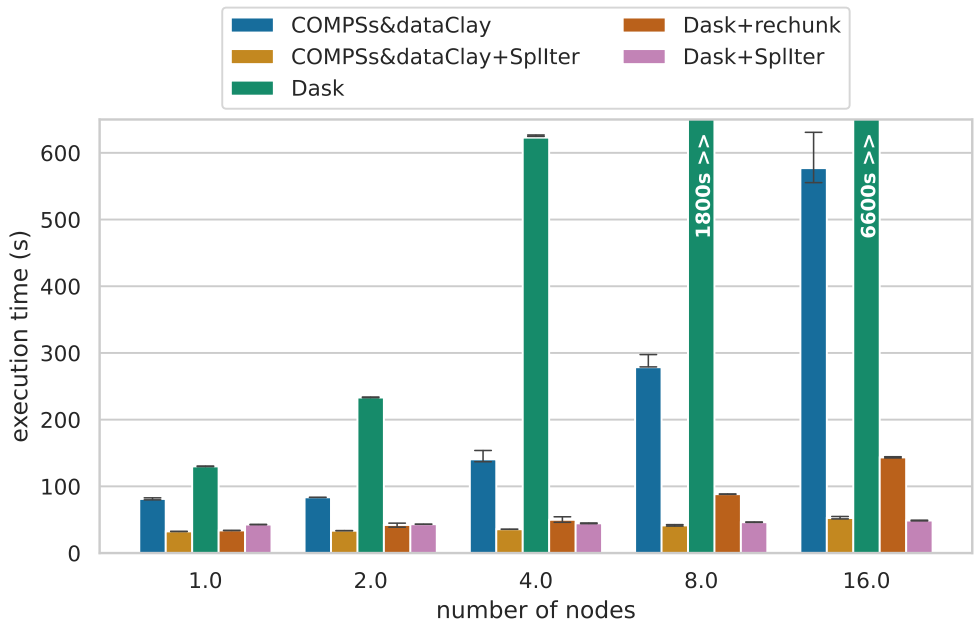


Figure 5.9: k-means weak scaling from 1 node to 16 nodes, with 2304 blocks per node (48 per core)

5.5.2 *k*-means

The next application is k-means. This application was already evaluated with dataClay in the previous chapter (see 4.5.2); in this section we will be doing an evaluation using the same dataset size as in the previous chapter (74 million points, with 20 features).

Scalability for highly fragmented datasets

Our goal for this experiment is to showcase the benefits of the *SplIter* for another quite memory intensive application. This application has an iterative approach and more complex numerical procedures (when compared to the previous one). This first experiment is done with a highly fragmented dataset in which each block contains 64 thousand points and it is divided into 2304 blocks per node (which equals to 48 blocks per core).

The iterative approach should reduce the overall impact of both the *rechunk* and the *SplIter* approaches: the burden of those mechanisms are “shared” amongst all iterations, reducing its relative penalty. Moreover, the longer execution times will make the overheads less visible. The more complex reduction will have additional data locality benefits that the *SplIter* can leverage during the reduction step.

We will be evaluating this scenario by performing a weak scaling, starting from 1 node up to 16. The experiment scales up to 36864 blocks for the 16-node execution.

The execution times are shown in Figure 5.9. Once again we can observe that *SplIter* executions have good scalability, showing very flat results and resulting in a very stable weak scaling behavior. Both implementations (COMPSs & dataClay + SplIter and Dask + SplIter) have results that are very close between them, showing that the amount of computation that they do is the same and the implementations are equivalent.

The baseline executions have a much worse scalability than the *SplIter* counterparts. We already saw in the previous application (Histogram) that the scalability for highly fragmented datasets results in a degradation of performance due to the big amount of tasks. Given that we are evaluating an iterative application (10 loops in this evaluation), the overhead is amplified 10 times. The execution times of Dask end up being one order of magnitude slower, which shows that the COMPSs scheduler is more robust to being stressed with lots of tasks; however, even in that case the COMPSs & dataClay ends up being one order of magnitude slower than the *SplIter* configuration.

An alternative to the *SplIter* would be to use the *rechunk*. This can be seen in the Dask + rechunk execution, which shows good results. Its scalability is not as good as the *SplIter* and we can see a clear uptrend in the execution times when the number of nodes (and thus, the number of blocks) increases. This overhead is due to the rechunk cost, which requires to move data between workers (a cost that increases when the number of nodes increases). However, this cost is only payed once, not for every iteration (just as the *SplIter* cost that is payed once).

Overhead for perfectly balanced datasets

The second experiment for this application will show what happens if the dataset is already perfectly balanced. Balanced means the same as in the previous experiment: the dataset is divided in such a way that there is one block per core. Once again, we want to focus on the overhead and general behaviour of the *SplIter* on a worst-case scenario, one where there is no room for improvement for any *enhanced iteration* mechanism.

This scenario will also be conducted through a weak scaling. The dataset size will be identical to the previous scenario, but each block will be bigger: 3 million points per block (48× bigger than the previous scenario). Figure 5.10 shows the execution times for this scenario.

We expected to see the pure overhead on the *SplIter* executions. That is the case for COMPSs & dataClay + SplIter execution; the execution time grows with the number of nodes (i.e. blocks). However we see a much better behavior for the Dask + SplIter execution. As seen in the Histogram application, the overhead of *SplIter* is lower in Dask. Moreover, the data locality achieved by the Dask workers during the reduction steps is higher than the one warranted by COMPSs.

In fact, the data locality achieved by the Dask + SplIter combination goes one step further. Because how the partitions are being generated (as detailed in Section 5.2), the first step of the reduction is able to exploit data locality too. This is not achieved by the baseline, and thus the Dask performance is slightly worse than the *SplIter* one –specially for a high number of nodes as the reduction cost increases.

The *rechunk* has no impact on execution times, as it is effectively a no-operation.

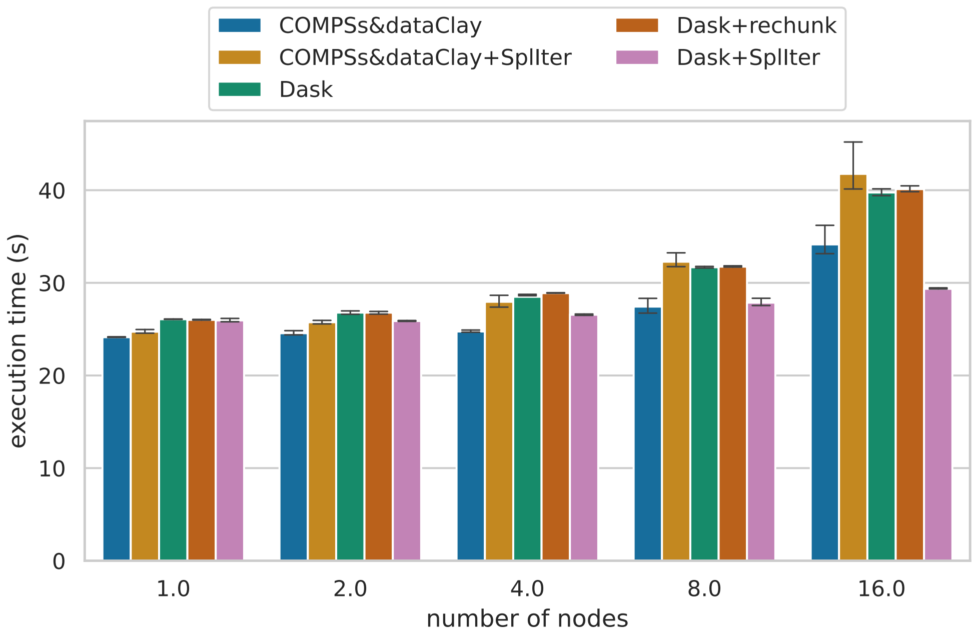


Figure 5.10: k-means weak scaling from 1 node to 16 nodes, with 48 blocks per node (one per core)

Sensitivity to fragmentation

We will follow with an experiment that should highlight the sensitivity of dataset fragmentation. The previous two experiments for this application have showcased a good case scenario and a worst case scenario for the *SplIter*. Now we will explore this spectrum and find the tipping point where the benefits of the *SplIter* are greater than its overhead.

In order to do so we will once again fix the number of nodes to 8 and change the number of blocks per core. The number of blocks per core will go from 1 (worst case scenario for the *SplIter*, as done in the previous experiment) up to 48 (which is the *highly fragmented dataset* discussed in the first experiment for this application). The results can be seen in Figure 5.11.

We have already discussed the leftmost and rightmost configurations (see the two previous experiments).

The *SplIter* executions are very stable, almost flat, showing that this mechanism reduces sensitivity to fragmentation.

The baseline executions are sensitive to fragmentation and their execution performance decreases when the number of tasks increases –i.e. when the scheduler is under pressure and there are a lot of invocations of small tasks. COMPSs scheduler is able to “resist” a bit longer (it starts degrading significantly between 16 and 48 blocks per core) while Dask starts to increase at 4 blocks per core and quickly becomes orders of magnitude slower than any other execution.

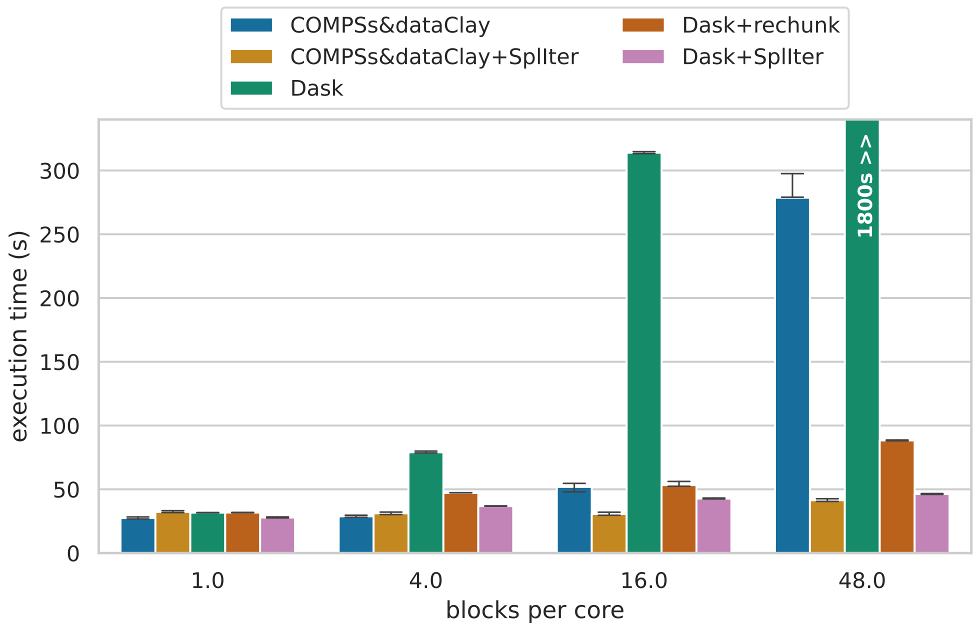


Figure 5.11: k-means on 8 computing nodes. The X axis show variation on the total number of blocks per core. Block size changes in order for the total dataset size to remain constant.

The rechunk shows an overall good behavior. It does reduce sensitivity to the dataset fragmentation, but not as effectively as *SplIter*. Its performance can be orders of magnitude better than the baseline but is still slower than the *SplIter* execution.

Insights

When handling an iterative application such as the k-means, including any mechanism to address the dataset fragmentation will result in substantial performance benefits. The overhead of *SplIter* or *rechunk* are diluted among the iterations and their benefits can be extremely large. Still, *SplIter* outperforms the *rechunk* because it is able to avoid data transfers (a cost that can grow for data intensive applications such as k-means) as well as to maximize data locality.

5.5.3 Cascade SVM

The next application is the Cascade SVM, a compute bound algorithm. The sizing of the dataset has been set in order to obtain reasonable execution times. Having a big dataset proved impractical for computation exploration as it required too many computing resources (i.e. for a statistically significant discussion). We have chosen a dataset size of 300 thousand points per computing node, which yields a good variety of execution times across all scenarios.

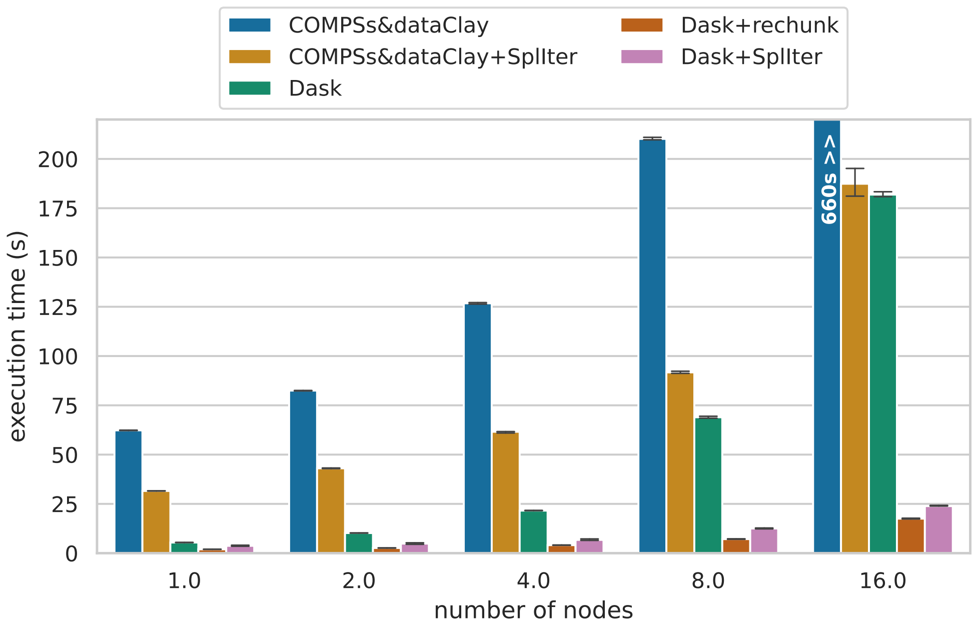


Figure 5.12: Cascade SVM weak scaling from 1 node to 16 nodes, with 384 blocks per node (8 per core)

Scalability for highly fragmented datasets

As with previous applications, we will start with a highly fragmented dataset. Our goal is to showcase the benefits of the *SplIter* for a compute-bound application. This means that the improvements due to locality will be lessened in comparison to previous applications. However, we do expect improvements due to the decrease in the number of tasks. By having less tasks we expect to increase responsiveness on the scheduler (it has less work to do) and a lower invocation overhead (due to the lower number of tasks). This is an ideal scenario for *rechunk*, as the quantity of data that needs to be moved for a rechunking operation is small, while the improvements should be substantial due to the computation being the bottleneck.

The dataset is 300 thousand points per node. The evaluation will be performed by a weak scaling up to 16 nodes. The execution is performed with a block size of 128 points. This results in 8 blocks per core, or 384 blocks per node. The execution times for this scenario are shown in Figure 5.12.

Before analyzing the behavior of the *SplIter* against the baseline and the rechunk, we should mention that there is a considerable difference between COMPSs and Dask. This difference is related to the baseline implementations and not to the *SplIter* or to the data granularity. The current execution (highly fragmented dataset) has several factors that make this comparison complex, so we will discuss this in the next configuration with a perfectly balanced dataset.

The *SplIter* executions outperform their respective baselines substantially. The improve-

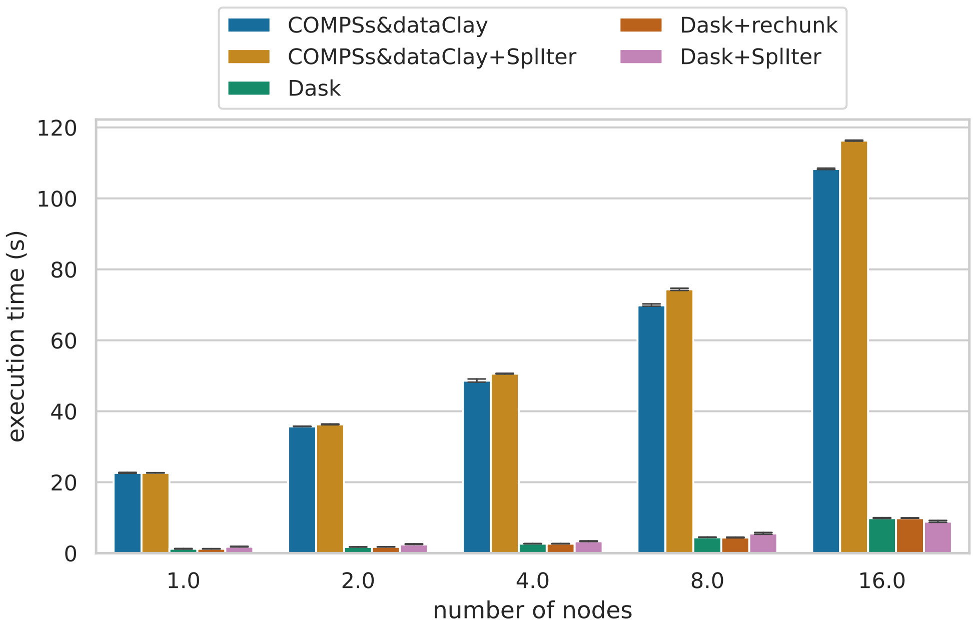


Figure 5.13: Cascade SVM weak scaling from 1 node to 16 nodes, with 48 blocks per node (one per core)

ment introduced by the *SplIter* is apparent, once again, for a highly fragmented dataset.

For this application, we see how the *rechunk* outperforms the *SplIter*. As discussed in 5.2.2, the *rechunk* materializes a new array, while the *SplIter* does not. In previous applications, the data transfer was bigger and resulted in faster execution times for *SplIter*. In this application, however, the data transfers are small enough to be almost invisible, while having the array materialized results in faster execution times.

Overhead for perfectly balanced datasets

We will now show an evaluation for a perfectly balanced dataset –with as many blocks as there are cores. In that scenario, once again, there should be no improvement for the *SplIter* or the *rechunk* –there is already a one-to-one relationship between computing resources (cores) and tasks (blocks), so there is nothing to improve in that regard. This evaluation will help us evaluate and characterize the overhead in extreme cases where no potential benefit exists from the point of view of the enhanced iteration mechanism.

This experiment will maintain the same dataset size (300 thousand points). When comparing to the previous scenario, the block size is increased from from 128 to 1024, and the number of blocks per core is reduced from 8 (the previous ratio) to 1 (i.e. perfectly balanced).

Figure 5.13 shows the execution times for this scenario.

Before discussing the *SplIter*, we can clearly see the different performance between

COMPSs and Dask. Up until now, prior application performances have been very close between COMPSs and Dask, as we have made an effort to compare equivalent implementations. In this case, however, some details of the implementation have resulted in a disparity of performance; both algorithms are effectively doing the same operations (we used the `dislib` implementation as the reference, and implemented that in Dask) but the general data management of intermediate results and the internal serializations are causing this difference. To address that we could try to improve the `dislib` implementation –which may prove difficult and is outside the scope of this thesis. Even if the current results hinder simultaneous comparisons across COMPSs and Dask, we can discuss the *SplIter* impact separately for both frameworks.

Both *SplIter* executions follow closely their respective baselines. The Dask + rechunk is almost exactly the same as Dask. The overhead on the COMPSs & dataClay executions is more visible, and we can observe how the overhead depends on the number of blocks –i.e. the difference between the baseline and the *SplIter* increases when the number of nodes increases. This overhead is smaller in Dask (a phenomenon that we have already seen in previous applications) which suggests that the mechanism to query location and build partitions is more efficient in Dask than in dataClay.

The *rechunk* is effectively a no-operation and its performance is identical to the regular Dask.

Sensitivity to fragmentation

We have shown a relatively beneficial scenario as well as a worst-case scenario from the point of view of the *SplIter* and *rechunk* mechanisms. In the following experiment we will showcase what happens across this spectrum, and specifically we will discuss how sensitive are the enhanced iteration techniques to the quantity of blocks and fragmentation of the dataset.

This experiment will fix the dataset size, fix the number of computation nodes, and vary the number of blocks. More tasks results in a more saturated scheduler and more runtime overheads –which translates to more opportunities for mechanisms that addresses granularity such as the *SplIter* or the *rechunk*.

The execution is spread across 8 computing nodes. The execution times can be seen in Figure 5.14. The leftmost group of bars show the execution times when the dataset is perfectly balanced (i.e., previous experiment); the rightmost group of bars show the other end of the spectrum (i.e., the second to last experiment, which corresponds to 8 blocks per core).

The *SplIter* executions are almost flat, showing that the *SplIter* is reducing the sensitivity to the task granularity. The execution times for 1 block per core (the worst case for the *SplIter*) is almost the same as the execution time for 8 blocks per core.

On the other hand, we see that both COMPSs & dataClay and Dask are quite sensitive to fragmented datasets. Their execution times are equal to *SplIter* for 1 block per core and grow to more than double when the fragmentation increases to 8 blocks per core. All

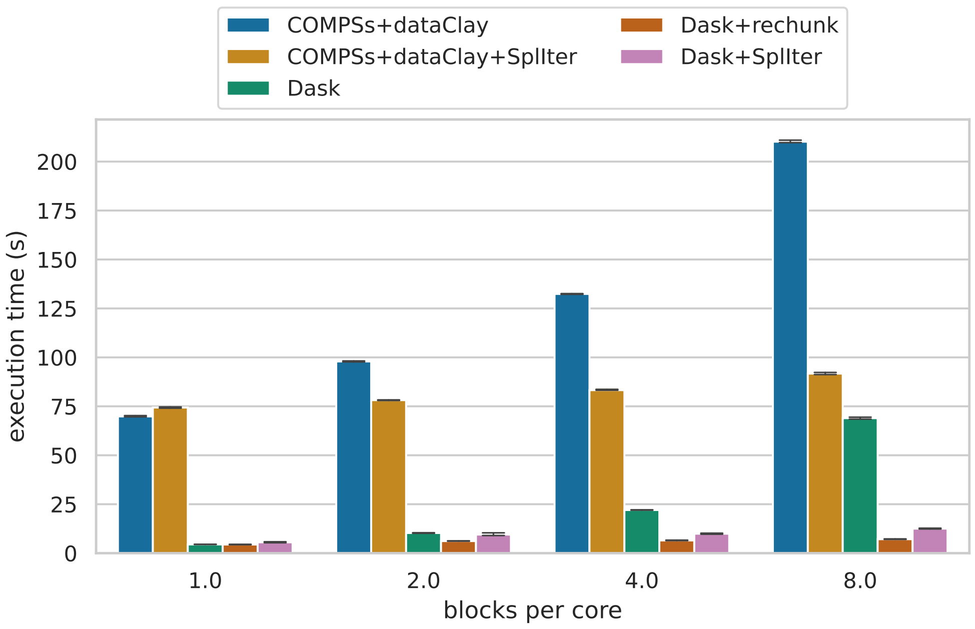


Figure 5.14: Cascade SVM on 8 computing nodes. The X axis show variation on the total number of blocks per core. Block size changes in order for the total dataset size to remain constant.

this overhead is due to the stress on the scheduler and the high number of task invocation, which become the bottleneck of execution.

This application is perfect for *rechunk*. As soon as there is more than one block per core, *rechunk* becomes the fastest configuration. The relatively small dataset results in a fast *rechunk* operation –meaning that the overhead of performing a rechunk is almost irrelevant– while having a materialized collection results in faster execution times when compared to the unmaterialized collection –the partitions yielded by the *SplIter*.

Insights

This is the first compute bound application that we have discussed and, once again, we have seen that there is the need to address dataset fragmentation. In this application, *rechunk* yields better performance than *SplIter*. The two previous applications where memory intensive and *SplIter* outperformed *rechunk*; this application, a CPU intensive one, shows that *rechunk* can beat the *SplIter* in certain scenarios. Even in these scenarios much more favorable to the *rechunk*, the difference between the two mechanisms is small.

5.5.4 *k*-Nearest Neighbors

This application (in contrast to the previous ones) behaves with more complex interactions due to its two-part algorithm. For this reason we feel the need to include here a specific discussion on the two main microkernels that are used: the *fit* and the *kneighbors* procedures.

After this characterization, we want to showcase how the *SplIter* and *rechunk* mechanisms are able to provide a substantial benefit related to the intermediate data structures that are generated (on *fit* stage) and used (on *kneighbors* stage). Our goal is to evaluate our proposal in the context of a complex application, application which also contains non-trivial data reuse. This will be shown through a general scalability series of experiments.

Finally, we will also analyze the speedup behavior when the *fit* dataset is scaled, as this will magnify the effect of the intermediate data structures and show clearly the speedup effect and full potential of the *SplIter* and *rechunk* mechanisms.

Kernel characterization

When discussing this application in 5.3.4 we discussed the impact that the size of the tree data structures will have onto the algorithm. There are a lot of nuisances, but the general intuition is that using big trees will result in less tasks, more efficiency and less overall execution times. In this first experiment we will show the impact that the tree size (which is directly related to the *fit* dataset block size) has onto the two main kernels of the application (i.e. the *fit* stage and the *kneighbors* stage. Each stage purpose and characteristics have been discussed in 4.3.3 and more specifically outlined in Figure 4.2 and Figure 4.3.

The times that we will be evaluating correspond to the numerical execution time for the payload of a single task (single block). In our scenario, this means executing the code on *sklearn* library without parallelism, data transfers nor framework overhead. As both stages are sensitive to the block size of the *fit* dataset, we will show the execution times when varying the *fit* block size. Figure 5.15 shows the execution times of the *fit* kernel and Figure 5.16 shows the execution times for the *kneighbors* kernel.

For the *fit* kernel (Figure 5.15) we can see how, when the input block increases, so does the processing time –in an almost linearly fashion. We are seeing here the cost of building the tree data structures and the cost is as expected.

The behavior of the *kneighbors* kernel (Figure 5.16) is much flatter. We already expected this because tree lookup operations do not increase linearly.

Table 5.1 explores microkernel execution time for four different block sizes. The first column shows the block size of the *fit* dataset (ranging from 3000k points for the biggest one down to 500k for the smallest). The second column shows the number of blocks in the *fit* dataset (equivalent to the number of tasks). This is a parametric value, which will depend on the size of the *fit* dataset. Decreasing the block size results in a bigger number of blocks; e.g. given a starting dataset of n blocks with a block size of 3000k points, that same dataset will yield $3n$ blocks when the block size equals 1000k points. The *fit* total time (meaning the sum of all kernel execution times, sequentially) also depends on n and varies depending on the block size. The *kneighbors* stage has its own dataset, which is why the last column on the table shows the execution time *per block*; the time shown in the table is the sum of all the *kneighbors* kernel executions (sequentially) for a single *kneighbors*

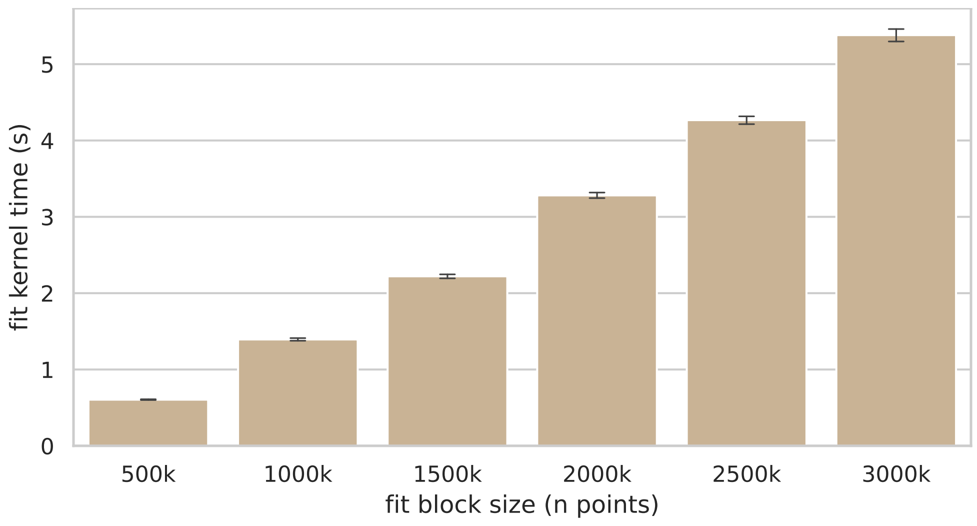


Figure 5.15: Kernel processing time for the *fit* procedure of the k-Nearest Neighbors application while increasing the *fit* block size

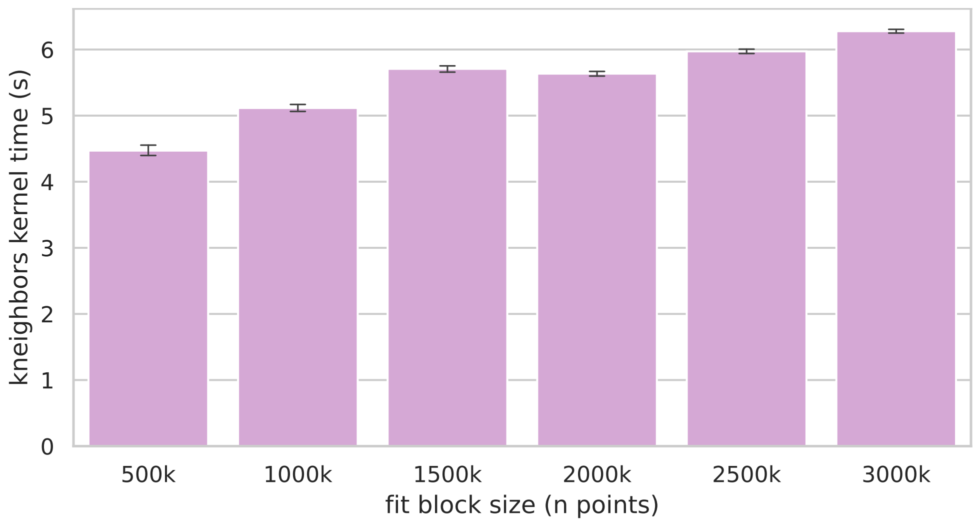


Figure 5.16: Kernel processing time for the *kneighbors* procedure of the k-Nearest Neighbors application while increasing the *fit* block size

Fit block size	**#*fit* tasks (= #blocks)**	***fit* total time**	***kneighbors* time per block**
3000k	n	$n\cdot$ 5.38s	$n\cdot$ 6.28s
1500k	$2n$	$n\cdot$ 4.44s	$n\cdot$ 11.42s
1000k	$3n$	$n\cdot$ 4.17s	$n\cdot$ 15.35s
500k	$6n$	$n\cdot$ 3.66s	$n\cdot$ 26.83s

Table 5.1: Combined execution times for k-Nearest Neighbors microkernels.

dataset block. To get the *kneighbors* total time we would need to mulitply that value per the number of blocks in the *kneighbors* dataset.

The table shows how the *kneighbors* is the longest stage and also the most sensitive to the block size (i.e. tree size). It is natural to consider beneficial configurations for it, and those are found when the *fit* block size are big –i.e. when the tree data structures are big. We must take into account this behavior, and that justifies our *SplIter* strategy for this application. Moreover, the tree data structures generated during the *fit* may be used by a lot of *kneighbors* blocks, and those trees may be used repeatedly during execution (e.g. for iterative algorithms); these characteristics dilute the cost of the *fit* stage when compared to the *kneighbors* stage.

With that in mind we will establish for the *SplIter* to build a single tree lookup data structure per location. This prior analysis corroborates the intuition that bigger *fit* block sizes increase the performance of the microkernel execution time. The last experiment for this application (see 5.5.4) will revisit this same discussion on the *kneighbors* execution time, evaluated in the complete stack environment (with COMPSs, with multiple nodes, with parallelism, and with full datasets).

General scalability

The general scalability scenario will be evaluated by using the *fit* dataset at 6 blocks per node, and 24 blocks per node for the *kneighbors* dataset. Consequently, the *kneighbors* stage will consist of a total of $6 \times 24 \times n_{nodes}$ tasks for baseline executions. Regarding the other executions, we will follow the conclusions reached in the previous experiment and use a single tree per location, i.e. per backend/worker. In our NUMA architecture computing environment this means to have a single tree per socket. This results in a total of $2 \times 24 \times n_{nodes}$ tasks for both *SplIter* and *rechunk* executions. The result of this first experiment are shown in Figure 5.17.

The *SplIter* executions are very close among them, showing that they perform the same computations and the *SplIter* performance is similar in both frameworks. The k-Nearest Neighbor cost increases when the problem size increases (i.e. when the number of nodes increases), which results in an increase in execution times in the *SplIter* executions.

The baseline executions increase faster, showing a quicker degradation of performance. In previous applications we have seen that the COMPSs framework is more robust against

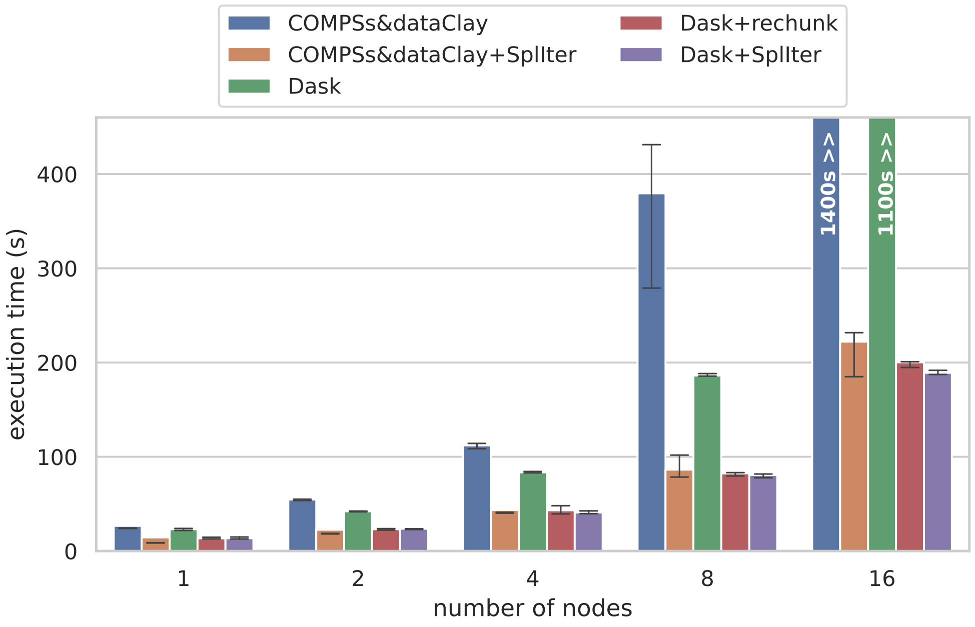


Figure 5.17: *k*-Nearest Neighbors scaling from 1 node to 16 nodes

scheduler pressure. However, the number of tasks is not as big in this application, and the data locality is more relevant –a feature that Dask is able to leverage more efficiently than COMPSs, due how Dask manages in-memory Python objects. This explains why the COMPSs & dataClay baseline is slower than Dask.

The *rechunk* execution is virtually identical to the *SplIter*; this was expected given that the final tree data structures should be equivalent (meaning equally large) in both scenarios.

Speedup for *fit* dataset scaling

In this experiment we will be scaling the training dataset (the first one, the one that is fed to the *fit* stage). The number of nodes is fixed to 8 and the size of the second dataset is also fixed to 24 blocks per worker, which is a total of 192 blocks.

Our goal is to observe the impact of both the *SplIter* and the *rechunk*, and how well they scale when the data structures generated in the *fit* stage grow. In these experiments, the non-baseline executions will generate a tree lookup data structure per location (i.e. one per backend/worker, or one per socket, which equals to a total of 16).

Figure 5.18 shows the ratio of number of blocks divided by the execution time. The plot starts at two blocks per core (the point where the baseline execution, the *rechunk* and the *SplIter* executions are all equivalent among them) and scales up to 12× which is a total of 96 blocks for the training dataset.

We see how the Dask baseline is flat, meaning that the speed at which blocks are processed depends proportionally on the training dataset size. When the training dataset

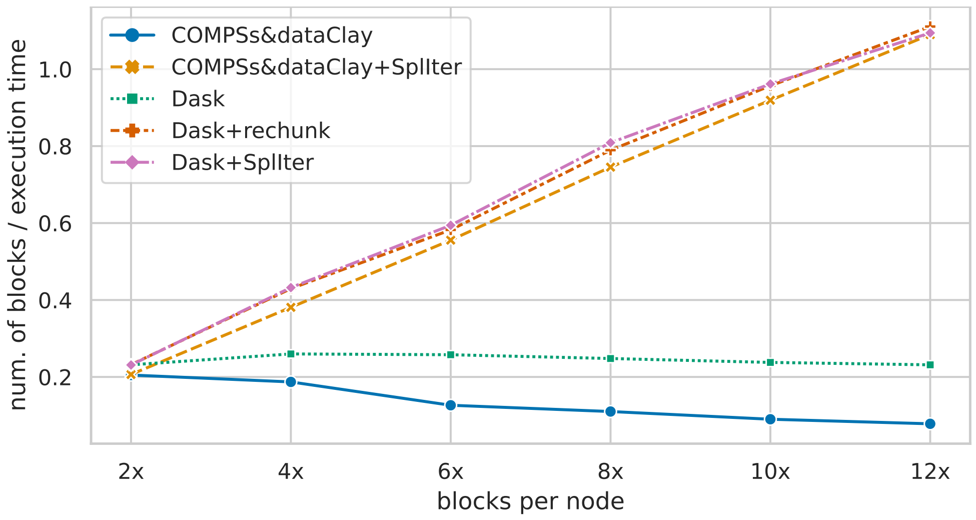


Figure 5.18: Evolution of the ratio number of blocks by execution time when increasing the number of blocks per node (higher is better).

size is doubled, so is the execution time. We can see that the ratio is quite flat across all the execution meaning that this trend is present from 2 blocks per node up to 12 blocks per node (in that last execution the execution is 6 times bigger than the first one, and thus the execution is also 6 times longer). COMPSs & dataClay execution shows a bit more degradation, but follows the same general trend. This degradation was already detected previously on 4.5.3, but what is important now is to realize that execution times is roughly proportional to the training dataset size too.

On the other hand, the ratio for all other executions increases linearly. The speed at which the *SplIter* execution processes blocks is not proportional on the training dataset size, but better –i.e., when the training dataset size is doubled, the execution time is much less than twice. The root cause is the internal lookup trees. The *kneighbors* stage is using the lookup trees generated during training; lookup operations on those trees are not $O(n)$ (linear) but $O(\log n)$. The complexity of lookup trees explain the good behavior of *SplIter* and *rechunk*: duplicating the size of the tree does not duplicate the lookup time.

To summarize this comparison: generating trees directly from the block data results in a linear behavior while consolidating blocks (either with *SplIter* or *rechunk*) results in executions that are able to perform closer to the theoretical logarithmic complexity.

Insights

This application shows how addressing the fragmentation and granularity issues of the data can give benefits that go beyond serialization and scheduler overhead. For this specific application, we started from a $O(\log n)$ theoretical complexity (the tree lookup stage) but observed it degrading into a $O(n)$ due to blocking (the baselines). Thanks to the use of either *SplIter* or *rechunk* we were able to greatly improve the performance –the execution time approaches once again the $O(\log n)$.

This improvement requires an understanding of the algorithm. At this moment, *rechunk* has no direct semantic to relate the size to the computing resources or number of nodes, so using it requires knowledge on the runtime computing resources. The *SplIter* has the semantics related to data locality and computing resources, which results in a simpler programming model interface which is able to reach the same performance benefits as the *rechunk* while avoiding data transfers.

5.6 Summary

In this chapter we have discussed and shown the *SplIter* proposal, which constitutes contribution **C2** in this thesis. At its core, the *SplIter* is able to leverage iteration optimizations and data locality with minimal impact on programmability.

In addition to designing and implementing the *SplIter*, we provide the behavior of this contribution across a variety of scientific applications, from several science domains (iterative or non-iterative applications, memory intensive applications, CPU intensive applications, machine learning, data analytics, etc.). The *SplIter* is able to reduce the performance sensitivity to the block size. Given that the application developer may not know the computing environment, decoupling the application performance from the block size is a huge benefit in terms of programmability[66] and performance portability[48, 9] from the programming model perspective.

The *SplIter* has been evaluated upon two different frameworks: COMPSs & dataClay and Dask. In general, the main idea behind the *SplIter* could be applied to any task-based programming model (more generally: to any programming model with direct access to the blocks and the iteration code structures). The evaluation has shown that the *SplIter* is able to compete and (in most situations) outperform the Dask *rechunk*.

We observed an application where *rechunk* outperformed the *SplIter* due to its very compute-intensive nature; this suggests that a new extension of the *SplIter* may consist on providing materialized partitions. A materialized partition would be a new data structure, generated with memory copies but without inter-node transfers. For iterative algorithms that have a high computation to data ratio a materialized partition should be able to achieve same execution speeds as the *rechunk* while avoiding all the inter-worker data transfers. Still, the performance difference between *rechunk* and *SplIter* was minor.

Complex applications can have multiple stages and iterations. In certain scenarios, the benefit of using *SplIter* can also be observed at an algorithmic level: the quantity and shape of intermediate results may depend on the block size and that shape may have an important performance impact –we have seen this behavior in the k-Nearest Neighbors, but having some kind of intermediate data structures is not an exclusive trait of this application. Using the *SplIter* allows the application developer to exert their (domain-specific) expertise and greatly improve the performance in a portable way, without requiring prior knowledge from distribution techniques nor any insight of the hardware infrastructure topology.

Chapter 6

Hardware Improvements: Non-Volatile Memory

There is a long-existing chasm between system memory (byte addressable, volatile, fast, scarce) and storage (composed of persistent but slow block devices, with huge latencies and a smaller bandwidth). In the State of the Art we discussed this gap and the memory hierarchy of the different technologies (Section 2.1.3, Figure 2.2).

Regarding the NVM layer, it can be used through two different approaches. First it can be used as a “fast storage device”. Alternatively, it can be used as an “extension of system memory” and perform in-place execution –thanks to the byte-addressable nature of the NVM. Being able to execute in-place, without copying data into DRAM, can have a great impact on data locality and thus performance.

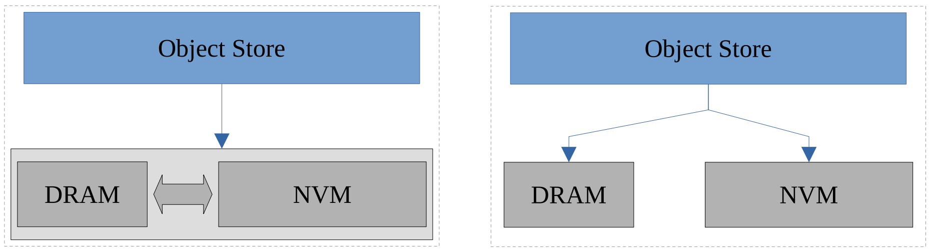
In this chapter we propose how to integrate NVM devices into dataClay, and discuss the benefits and synergies between the NVM presence and the *active* capabilities of the object store. This constitutes contribution **C3**.

6.1 Active execution and NVM

During the previous chapters we have discussed the active capabilities of dataClay. In this chapter we will explore how these active method mechanism interacts with the persistent memory space provided by NVM devices.

First of all, let’s introduce the two main motivations for using NVM devices in the active storage system:

- They have much better performance than drives, available at a fraction of the price-per-byte cost of DRAM. This allows the object store to manage big datasets –datasets that would not fit in DRAM– without incurring in the high performance penalties of storing and accessing them to/from drives.
- They are byte-addressable, which enables in-place execution (i.e., direct load/store accesses are possible and efficient, without moving data to DRAM). Being able to



(a) Memory Mode (b) App Direct mode

Figure 6.1: Configuration modes for the Optane DC NVM

execute code in-place can boost the data locality; this perfectly combines with the idea of an *active* storage system that is itself improving data locality.

The NVM devices that we will be using for the evaluation are Intel Optane DC devices, which have two main operating modes [45]: *Memory Mode* and *App Direct*.

The first operation mode (called *Memory Mode*, Figure 6.1(a)) presents a big (but volatile) system memory address space, transparently accessible by all software. This is achieved by the aggregation of the Optane DC memory into a single flat memory space. This memory space is transparent to the object store and any software being run in the system. The DRAM of the system is used as cache for this memory space. The hardware will be the one responsible of moving data back and forth between DRAM and NVM. This mode of operation does not require any software adaptation to the Optane DC devices.

Note that under Memory Mode configuration the memory space remains volatile in nature; if persistence is desired, it should be managed outside the Memory Mode bound devices.

When the system is configured in *App Direct* (Figure 6.1(b)), the system memory is unchanged and the Optane DC devices memory is persistent and independent from the main system memory. This is the canonical way of using any NVM device: an additional memory tier that can be accessed from the application/middleware. Under this configuration the object store is able to access data both in the DRAM and in the NVM, in a byte-addressable fashion. In order to use that memory space, applications can take advantage of the *Direct Access* (`dax`) feature which allows applications direct load/store access to persistent memory by memory-mapping files on a persistent memory aware file system [82, 2].

The assignment of persistent memory into one or the other operation mode is a static parameter and changing the configuration of the Optane DC devices will require a reboot of the machine. It is also possible to segregate the memory into two regions and use a different mode in each one, and thus each region will behave according to its configuration. In this article we will evaluate and discuss each mode separately, which will allow us to characterize them properly.

Both *Memory Mode* and *App Direct* will be evaluated in this chapter. Under the *Memory Mode* configuration, dataClay requires no changes given the transparent nature of the *MM*.

The design we propose is thus focused in the *App Direct* mode. However, the evaluation also includes the *Memory Mode* in order to compare and provide a comprehensive analysis on execution performance.

The evaluation will also include execution with a NVM-native non-active storage system: DAOS [1]. The DAOS storage system has a proven good performance, as shown in the IO500 ranking results [87]. This configuration is the baseline for non-active execution.

6.2 dataClay integration with NVM

As stated before, using NVM explicitly (which can be achieved under the *App Direct* configuration) requires runtime support. This means that certain code injections are required. The implementation we propose uses the PMDK [83, 82] libraries (version 1.5), and its *pynvm* Python bindings [84] of PMDK (version: 0.3.1). The numerical structures used in the applications are `numpy` arrays, represented as contiguous in-memory buffers –their canonical representation. Generally, any data structure could be used as long as its in-memory representation was known –e.g., based on contiguous memory buffers.

To understand the changes required, let's start with a very fundamental snippet of a one-dimensional Histogram using numpy library and dataClay active methods. This code is shown in Listing 6.1. Note that this sample code will work regardless of the `self.values` nature –meaning that if the `values` attribute is in NVM, the execution will work and the numpy library will perform in-place execution on the NVM.

```python
class PersistentBlock(DataClayObject):
    ...
    @dclayMethod(...)
    def histogram(self):
        return np.histogram(self.values, bins)
```

Listing 6.1: Active method for a Histogram

Following this train of thought, the core idea of our proposal is to have the `numpy` structures in the NVM. The big picture for this procedure is shown in Listing 6.2. Note that the code is simplified and lacks all kind of metadata, numpy types, data alignment, array shape, etc. It shows the two crucial steps: the memory map (line 1, storage initialization of NVM region) and the buffer feature of numpy (lines 5-7).

```python
1 reg = pmem.map_file(...)
2
3 def np_persist(np_array):
4     """Usage: a = np_persist(a)"""
5     b = np_array.tobytes()
6     reg.write(b)
7     return np.frombuffer(reg.buffer, ...)
```

Listing 6.2: Procedure to move numpy structure to NVM

The first key aspect here is the memory map. Note that this is supported through the DAX flag –the Direct Access for files feature of Linux kernel[80]. Otherwise, the Memory

Map would result in additional data copies between persistent memory and DRAM. With this first step we have `reg` which is a byte-addressable region of amorphous space.

The second key aspect is putting the numpy data into the persistent memory device. Note that by design, numpy structures are memory-contiguous and can work from arbitrary buffers. A memory map is a typical example of such a configuration, where data can be read from and written to. In the barebone example shown, the data from a regular in-DRAM numpy array is read and written into persistent memory. Then, a new numpy array is constructed, one that is backed by the persistent memory –taking advantage of its byte-addressability and unlocking the capability to perform in-place execution.

Our contribution explores taking this main idea (the union of the two key aspects just discussed) and combining it with dataClay. By performing this `np_persist` procedure –shown in the code snippet– inside the backend, dataClay can control the lifecycle of the data structure while leveraging in-place (in-NVM) execution of *active* methods.

6.3 Applications

In this chapter we will dive deeper into the memory hierarchy and the synergies between *active* methods, DRAM memory and NVM space. For this reason, we will look into a different set of scientific applications and we will evaluate them in a single node with no parallelism (i.e., no distributed execution framework). The final goal of our evaluation is to show how data locality plays an important role on the performance of applications. The applications we propose are: Histogram (single dimension), k-means, matrix addition and matrix multiplication. They are well-known kernels of more complex real-life applications. And not only they are widely used but their behavior is representative of a much larger set of applications; this will be discussed on a per-application basis in the next subsections.

The source code for the applications discussed in this chapter is available on GitHub[17].

The exact memory layout of the application data was not a performance-critical aspect on the previous evaluations. For this reason the data structures were not discussed in-depth on previous chapters (doing so would have needlessly convoluted prior *Applications* and *Evaluation* sections). However, in this chapter we will need to discuss in more detail the exact read and write patterns of the applications and their numerical routines because the memory will behave differently and data layout is relevant for understanding all this. Even for applications that have been already introduced, it is necessary to review the exact layout in order to understand the results and the evaluation, and this will be done in this section.

For the non-active evaluation (DAOS executions), the *method* will be executed in the application itself and the object store will transfer the dataset on demand –just as we would naturally do with an object store, in the absence of active features. The division of the dataset and the computation hardware remain the same in both the active and non-active evaluations.

All the evaluations will be performed with two different dataset sizes: *small dataset* (which fits in DRAM) and *big dataset* (which does not fit in DRAM). The dataset is

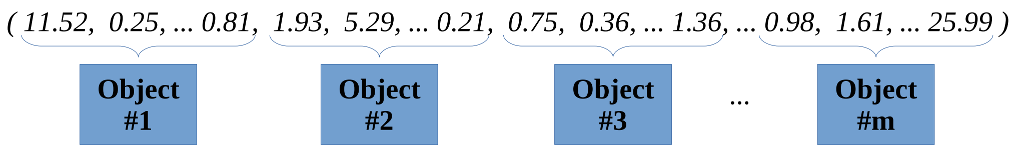


Figure 6.2: Object data structure for histogram input dataset

divided onto objects, for which we will consider two different sizes: *(a) big objects* and *(b) small objects*. The dataset size has no impact on the object size, only on the number of objects. In the latter –*small objects* scenario–, the objects are shaped to be roughly the size of the CPU cache (about a dozen megabytes) while the former –*big objects*– are an order of magnitude bigger.

6.3.1 Histogram

The first application is, once again, a Histogram. In this case, it will be a one-dimensional Histogram; this means that the input is an array of floating point numbers. The memory access pattern of the Histogram is shared amongst many applications (e.g. filtering, max/min finding, linear searches...). Our results and discussion can be extrapolated to all applications that read a big dataset once with no sizeable write operations.

The input dataset is an array of floating point values. This data structure is split into blocks, each block becoming an object in the object store –as shown in Figure 6.2. Those objects are `numpy` arrays of floating point values. The output data is a `numpy` array containing integers –the aggregates associated to the bins. This result is of small size and is returned by the *method* –it is not stored in the object store. The output size is several order of magnitudes smaller than the input, and it is constant in size.

The method computes the histogram. The return is the evaluation a partial histogram and –afterwards, outside the method– all the results are merged onto the final histogram.

6.3.2 *k*-means

The next application is one that we have already extensively discussed. The memory access patterns are identical for each iteration. The number of iterations is once again fixed and no convergence criteria is applied.

In this chapter we will focus in how the k-means clustering algorithm performs several sequential reads of the input data. This memory access patterns is shared with other scientific and machine learning applications with similar patterns (genetic algorithms, time-step simulations...). The results and discussion can be extrapolated to any of those applications that share this same memory access pattern –a big dataset that is read, whole, several times during the application execution.

The input dataset representation is an array of n-dimensional points. This data structure is split into blocks, each block becoming an object in the object store –as shown in Figure 6.3. Those objects are `numpy` matrices, each of which represents a set of n-dimensional points.

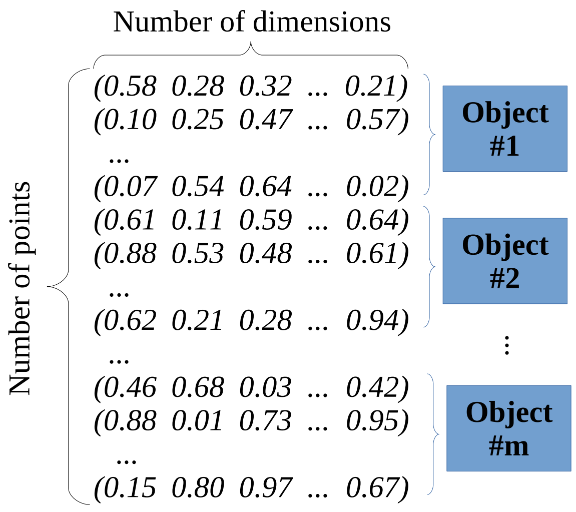


Figure 6.3: Object data structure for k-means input dataset

The output data is itself a `numpy` matrix, representing the centroids (which are n-dimensional points). This centroids structure is quite small as the number of centroids will be orders of magnitude smaller than the number of points, and thus the *method* will return the centroids data structure by value –they will not be stored in the object store.

For each iteration, the main numerical method receives the last iteration centroids as an input. This function is able to perform categorization for the points in a single object (the current block) and also evaluate a partial summation for its points given the centroids. All the partial summations can then be processed –outside the method– in order to obtain the centroids for the next iteration. Note that both centroids and partial summations are small data structures; their size is related to the number of centroids which is orders of magnitude smaller than the number of points.

6.3.3 Matrix addition

The matrix addition implementation will use randomly generated matrices (two $n \times n$ square matrices) and perform their addition (resulting in a third $n \times n$ square matrix). The complexity of the matrix addition operation is $O(n^2)$.

The data access pattern for the addition operation is predictable and deterministic: a single sweep is done to both input matrices; within this sweep the data is written to an output matrix, without reusing the input blocks. The application presents a *sequential data access pattern*, with *no object reuse*.

The memory access pattern of this application is shared with most *map* operations (from *map-reduce* programming model applications) and also with most data transformations kernels. All those applications are implemented by being applied to a big dataset, sequentially, and yielding a similarly-sized output dataset.

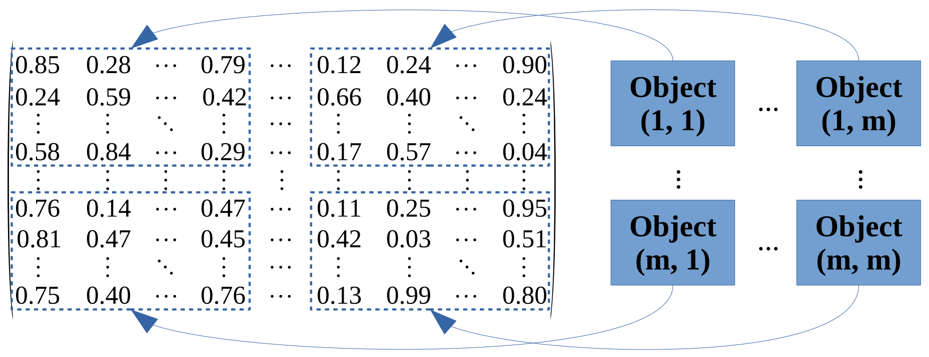


Figure 6.4: Object data structure for matrix data structures

The input dataset are two square $n \times n$ matrices. The output dataset is a single $n \times n$ matrix. Each matrix is represented as a two-dimensional array of submatrices –each submatrix is $k \times k$. Those submatrices are stored as objects in the object store –as shown in Figure 6.4. The size of the output matrix equals the size of either of the input matrices, meaning that the output size is comparable (half) the input data.

The addition of two objects (two $k \times k$ submatrices) is what the method computes. The output of that function is itself a new object (a submatrix) that may or may not be put into the object store –i.e., persisted and/or put in NVM. We will specifically discuss the placement of this output object in the evaluation.

6.3.4 Matrix multiplication

The matrix multiplication will use (just as in the matrix addition) randomly generated $n \times n$ matrices. We will consider the iterative multiplication implementation with $O(n^3)$ complexity. All previous applications have a computational complexity that matches their storage requirements. But that is not satisfied by the matrix multiplication: the computational complexity ($O(n^3)$) has a substantially larger growth than its storage ($O(n^2)$). That difference between storage and computation means that the storage access overhead becomes less and less important (performance-wise) when the dataset is increased.

The data access pattern for matrix multiplication is more complex than the matrix addition. In the discussed implementation, which uses "block matrices", the computation of the result is done by iteration of the output blocks. Each output block requires a certain quantity of multiplications and additions of input blocks. Once those multiplications and additions are finished, the output block can be stored and it won't be reused again. Note that there is reuse of the input blocks, as any block in the input matrices will be used to compute more than one block of the output matrix. This makes this application have much more data reuse than previous ones; and even though the reuse of data follows a deterministic pattern, it is not a trivial one –e.g., it is not multiple full sweeps as we could observe in the k-means. This application is a representative of *non-sequential data access pattern* with *object reuse* –as well as having a method implementation with *data reuse*.

We have chosen the matrix multiplication as an iconic and well-known numerical kernel, although we can find other matrix algorithms that will follow similar memory access pattern on input and output matrices –for example, matrix decomposition algorithms. One can also expect this pattern of non-sequential reads onto big datasets in certain applications like mesh-based simulations, data analytics, and others.

The data structure for this applications is the same as in 6.3.3 *Matrix addition*.

The numerical method is responsible of performing a *multiply-accumulate operation*. This is used with the (initially zero-initialized) output objects (submatrices) in order to perform the matrix row-by-column multiplications and additions. Using multiply-accumulate mechanisms –which are sometimes called *fused mutiply-add* or *FMA* in low-level jargon– for matrix multiplication is a well-known approach and it is appropriate for a block matrix multiplication algorithm.

6.4 Hardware platform

The hardware configuration used for the following experiments is different from the previous chapters. We require a system equipped with Optane DC Persistent Memory Modules. The technical specs of the node that we used for the experiments are the following:

- 2×Intel® Xeon® Platinum 8260L CPU @ 2.40GHz
- 192GB (12×16GB, 2666MHz) of DRAM
- 6TB (12×512GB modules) of Intel® Optane™ DC Persistent Memory

This machine contains a total of 48 (2×24) usable cores. The machine supports both the *Memory Mode* and *App Direct* execution modes; a reboot is required in order for its mode to be changed.

All the experiments will be evaluated with the same single machine, both while using dataClay (active object store) and while using DAOS (non-active object store). The executions will be done sequentially, although the numerical libraries will be taking advantage of the multiple cores through their internal multithreading capabilities. The availability of two CPU sockets allows us to isolate the application and the object store by placing them (by binding processes) to different sockets.

If we look into the hardware cost and compare the cost per gigabyte of DRAM versus NVM we can expect the latter to be lower; for typical builds, Intel reports savings of up to 30%[46]. Persistent Memory also allows a system to dramatically increase the total available memory. For starters, one can easily find 512GB Persistent Memory DIMM modules. We also have the consider the cost, as the price of a DRAM DIMM can be 5 to 10 times higher than a same-sized Optane Persistent Memory DIMM[3].

Moreover, the evolution of storage technologies has always led to increasing their performance, enlarging their capacity, and becoming more affordable. This happened with consumer hard drives and the pattern repeated with the SSD technology. It is natural to expect a similar evolution with NVM technology.

All datasets used for evaluation are tuned to be smaller than the available Persistent Memory. There is no technical limitation behind this decision, as the object store is designed to use the disk when required. However, doing so will incur in staging operations –just as any storage system would. Those disk staging operations are completely orthogonal to the active features and to the evaluation focus.

6.5 Evaluation

In order to discuss the potential of an active object store –compared to a non-active storage system– and the implications that data locality can have on the application execution performance we will be evaluating the four applications explained before (Histogram, k-means, matrix addition and matrix multiplication) under different configurations.

First and foremost, we will be focusing on the *active* behaviour of dataClay (the object store) while comparing it to a non-active object store (DAOS). The evaluation will take into account the different configuration alternatives of the Optane DC –the available NVM devices.

With all that in mind, we will be evaluating two different dataset sizes. First, a *small dataset*, where the whole data structures (both input and output) are smaller than the available DRAM. The other one, the *big dataset*, is dimensioned to be at least twice as big as the system memory, which ensures that the access to the NVM is realistic –smaller datasets may take advantage of cache through the faster DRAM, which would result in over-optimistic execution times. For each dataset size we will evaluate two different object sizes. The object size does not impact the total dataset size; big objects will result in fewer *active method* invocations but with a higher cost (due to the higher data per object).

The four different setups (three active ones with dataClay with different memory configurations, and one non-active one with DAOS) are the following:

dC DRAM [dataClay, Active] Consists on an application execution where the object store holds all the dataset in DRAM. This can only be evaluated for the *small dataset* as, by design, the *big dataset* does not fit in DRAM. The application will invoke the *active method* within dataClay. The NVM is not used in this scenario. The objective of this configuration is to evaluate the active object store with the fastest memory.

dC NVM [dataClay, Active] In this configuration, the input dataset resides in the NVM. The application will invoke the *active method* within dataClay. Said *active method* will perform in-place execution. The objective of this configuration is to evaluate the impact of the NVM performance and the potential of in-place execution on NVM.

dC MM [dataClay, Active] When the *Memory Mode* is configured, the object store holds the dataset in the flat memory space of system memory –that combines the NVM as main memory plus DRAM as a cache layer in a transparent fashion. The application

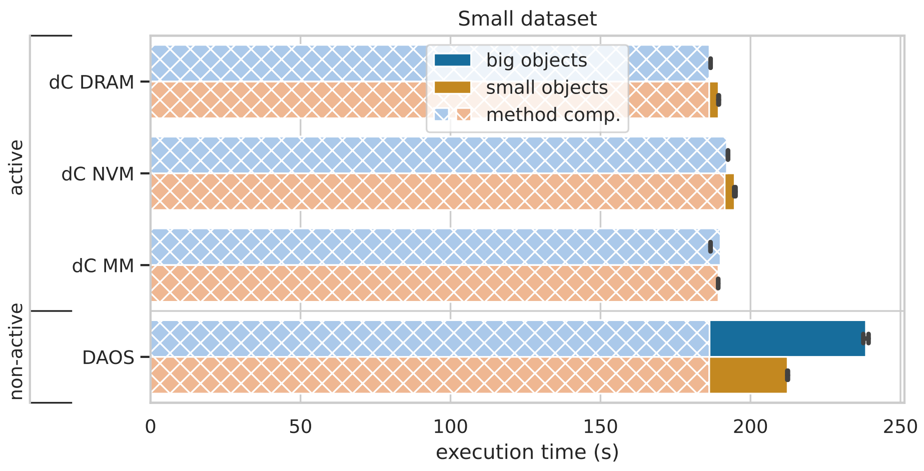


Figure 6.5: Execution times of the histogram application for a small dataset ($n = 3.2 \times 10^9$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The DRAM execution uses no additional memory tier. The NVM execution is done in-place on the persistent memory space. The MM execution uses the transparent mode of the Optane DC called *Memory Mode*.

will invoke the *active method* within dataClay. The objective of this configuration is for comparison when the NVM memory space is managed by the hardware with a single memory space –as oposed to the previous configuration where the DRAM and NVM memory spaces are separated and managed at the storage system level.

DAOS [DAOS, Non-active] The DAOS object store is used as the baseline non-active storage system. In this configuration the dataset is held in NVM by the storage stack. The application will retrieve the objects from the object store and execute the (non-active) *method* within the application space (in DRAM).

6.5.1 Histogram

The execution times of the Histogram application can be seen in the following figures: Figure 6.5 shows the execution times for the *small dataset* and Figure 6.6 shows times for the *big dataset*. We will start by looking at the total execution times (the whole bar, ignoring the lighter hatched portion for now) for each configuration –on upcoming paragraphs we will discuss about the meaning and insights given by the *method* computational portion.

All the active execution times, regardless of the dataset size or the object size, are roughly similar –within a 5% between one another. We can observe that the non-active object store execution (DAOS) is between 10% and 20% slower. This shows the overhead caused by the communication –memory transfers and data copies between the non-active object store and the application– in comparison to the computation. A similar pattern

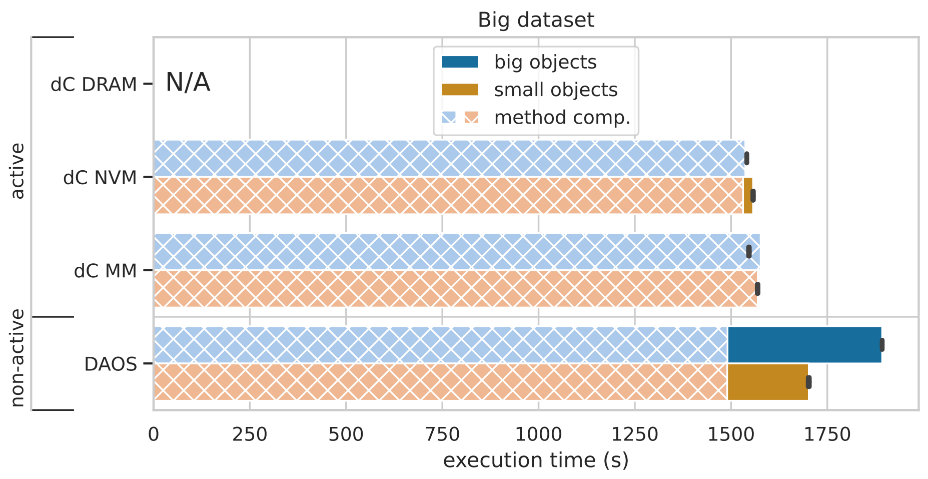


Figure 6.6: Execution times of the histogram application for a big dataset ($n = 25.6 \times 10^9$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The big dataset does not fit in DRAM. The NVM execution is done in-place on the persistent memory space, while the MM uses the *Memory Mode* transparent behaviour available in the Optane DC devices.

would be shown by any similar application with no data reuse and high computation to data ratio.

It is important to note that, when comparing the non-active DAOS executions, using big objects is slower than using small objects. This shows that the performance toll for the non-active case is not a consequence of the number of RPC calls. In this evaluation, small objects are 8 times smaller than big objects which result in 8 times more RPC. It may seem natural to expect better performance with less RPC calls. Instead, the cost of object transfers –required for non-active execution and the responsibility of the object store– increases with the object size, which results in slower execution times for bigger objects. This happens under the DAOS non-active executions for both dataset sizes.

Upon closer inspection, it is surprising to observe how close all the active configuration execution times are amongst them. In order to shed some light on this behaviour we have included an evaluation of the execution time of the *method* itself –the piece of code that is shipped to the active object store. Figure 6.7 shows the execution times of the *method* itself when this piece of code is executed either in DRAM, in the NVM, or in the *Memory Mode* Optane DC configuration. Given that the *Memory Mode* acts as a transparent cache, we have evaluated this last mode in both a hot and a cold configuration. Note how for the method execution evaluation under *Memory Mode* we are able to control the environment and consider both *hot* and *cold* execution times. For the full application execution we are considering the *hot* execution time for the small dataset, as we expect everything to fit in DRAM and thus *hot*. The *cold* execution time is used as the estimator for the big dataset;

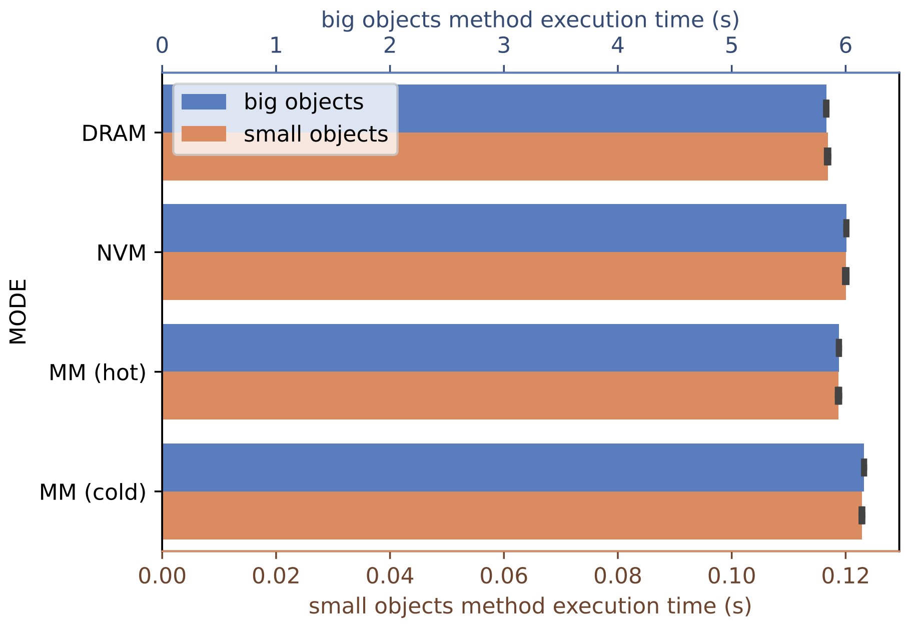


Figure 6.7: Execution times for the histogram *method*. The DRAM execution is done with no assistance of other memory tiers. The NVM execution is done in-place on the persistent memory space. The MM executions correspond to the transparent *Memory Mode* available in the Optane DC devices, which resembles a caching –hence the hot/cold executions.

it serves as a worst-case scenario, assuming that all accesses to the data miss the DRAM and go to NVM. The lighter hatched bars in Figure 6.5 and Figure 6.6 are obtained by extrapolating the execution times for the histogram method and are shown in order to give an orientation of the execution time estimated to be due to the method execution.

Insights

We have seen how the active configuration –and the data locality it implies– introduces a 10% to 20% benefit to the overall execution times. This application showcases an scenario with no data reuse and a high *method* computation index. Under those assumptions, the application has low sensitivity to the exact memory configuration, meaning similar performance for all NVM modes tested.

6.5.2 k-means

The execution times for the k-means application can be seen in Figure 6.8 (for the small dataset) and Figure 6.9 (for the big dataset). At a first glance we can see how, once again, there is a clear benefit on using an active storage system when compared to the non-active

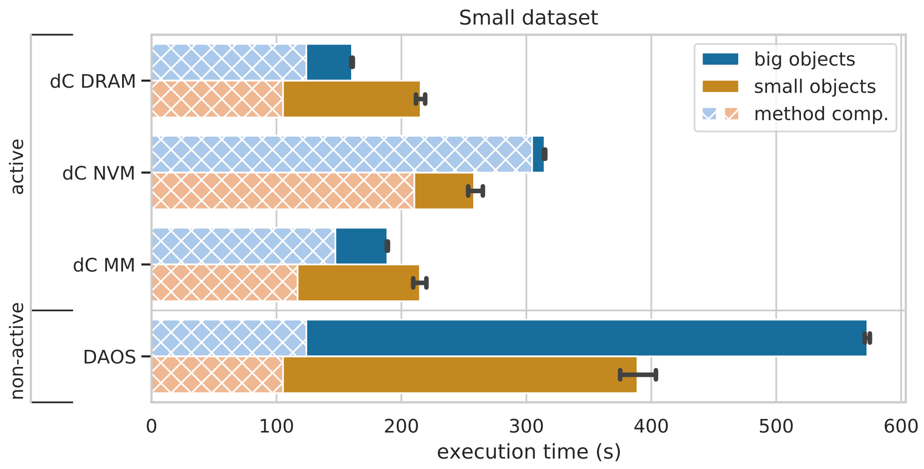


Figure 6.8: Execution times of the k-means application for a small dataset ($n = 6.4 \times 10^6$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The DRAM execution uses no additional memory tier. The NVM execution is done in-place on the persistent memory space. The MM execution uses the transparent mode of the Optane DC called *Memory Mode*.

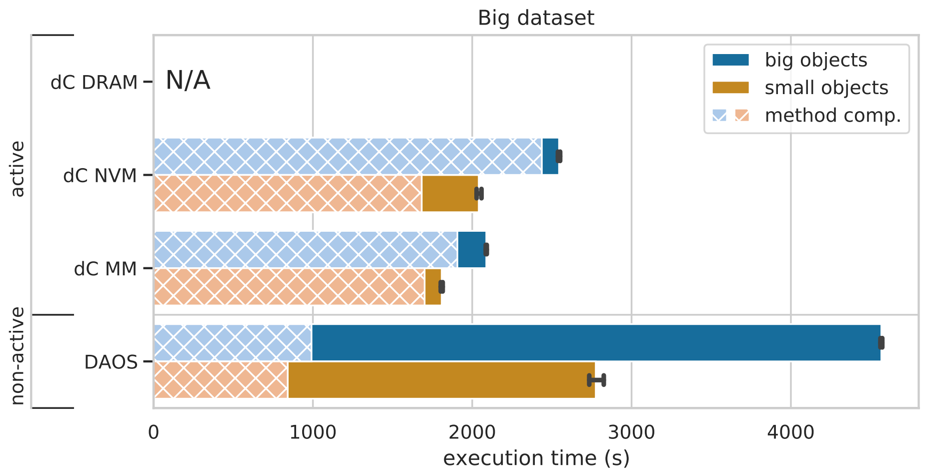


Figure 6.9: Execution times of the k-means application for a big dataset ($n = 51.2 \times 10^6$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The big dataset does not fit in DRAM. The NVM execution is done in-place on the persistent memory space, while the MM uses the *Memory Mode* transparent behaviour available in the Optane DC devices.

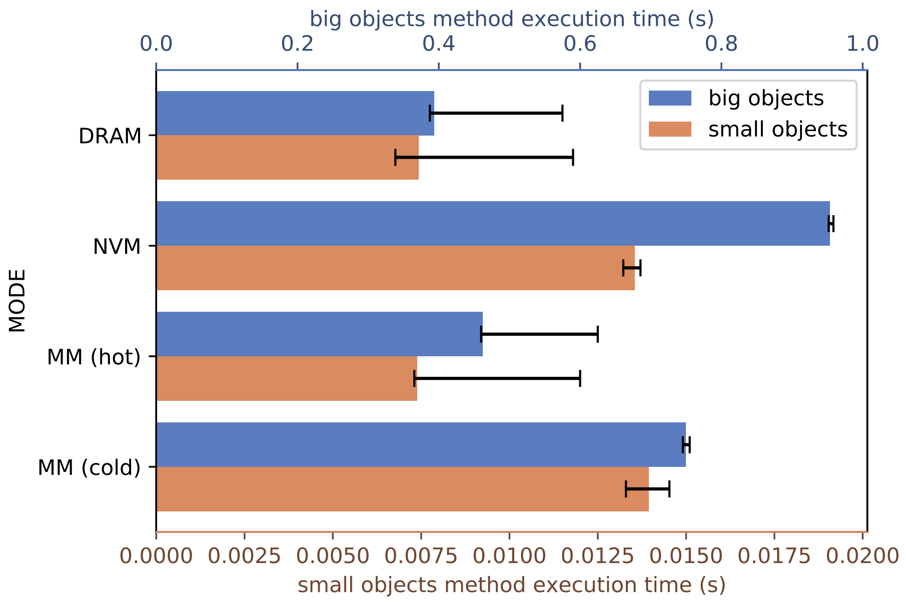


Figure 6.10: Execution times for the k-means *method*. The DRAM execution is done with no assistance of other memory tiers. The NVM execution is done in-place on the persistent memory space. The MM executions correspond to the transparent *Memory Mode* available in the Optane DC devices, which resembles a caching –hence the hot/cold executions.

object store. The non-active overhead goes from 30% –when comparing with the slowest active configuration– up to a 300%. This increased gap between the active and non-active setups is greater than on the previous application mainly due to the data reuse of the application –all data is reused for each iteration for a total of 10 iterations. This showcases how, in relative terms under this scenario, the computation time is less relevant and the time consumed by data transfers is more prominent. However, it is also apparent that this application has higher sensitivity to the object size as well as the memory configuration, as shown by the higher spread on execution times across the different configurations.

The k-means *method* being used –mainly, the pairwise distances evaluation as implemented in the `numpy` library– has a certain quirk: it reads input data several times. That means that the *method* implementation is performing multiple read operations onto the same object; that is a consequence of `numpy`'s numerical implementation.

For a better understanding of the *method* behaviour we have included the execution times in Figure 6.10. The high execution times on the NVM memory configuration –i.e., when data is computed in-place in the NVM– suggest that the data reuse in the *method* is amplifying the difference of speed between the DRAM memory tier and the NVM one. The implementation sensitivity to memory speed also explains the high variability amongst DRAM-based executions –*DRAM* and *MM (hot)*.

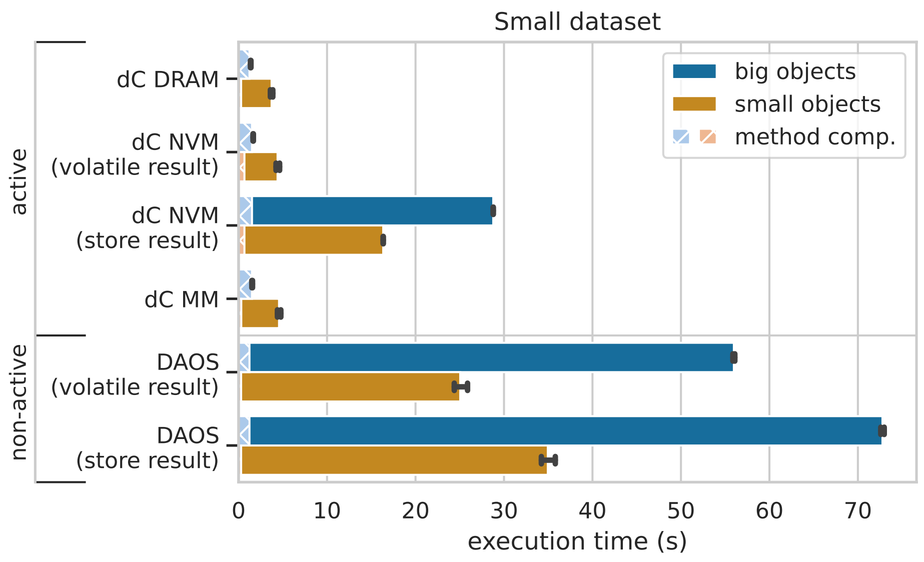


Figure 6.11: Execution times of the matrix addition application for a small dataset ($n = 42000$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The DRAM execution uses no additional memory tier. The NVM execution is done in-place on the persistent memory space. The MM execution uses the transparent mode of the Optane DC called *Memory Mode.* The result may be held in DRAM (*volatile result* executions) or stored in the object store (*store result* executions).

Insights

This application shows how algorithms that iterate over the input dataset multiple times –or, more generally, that have certain reuse along their execution– will obtain great benefits from being executed in an active environment. This is the consequence of the repeated access to the data, which magnifies the gains introduced by the data locality of the *active* object store. This application also reminds us the relevance of the numerical low-level implementation; the multiple read operations done at the numpy library level has a huge impact on execution times. Further improvements would require low-level knowledge on the numerical library implementation and the development of specific strategies for the application –something that only makes sense for heavily used microkernels which warrants the resource investment.

6.5.3 Matrix addition

The next application used for the evaluation is the matrix addition. The execution times for the application when run with a small dataset and a big dataset can be seen respectively in Figure 6.11 and Figure 6.12.

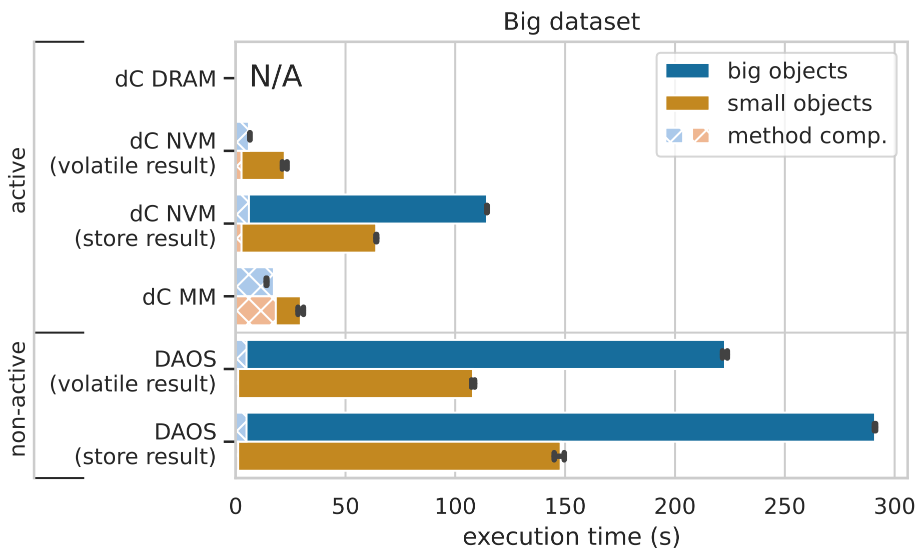


Figure 6.12: Execution times of the matrix addition application for a big dataset ($n = 84000$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The big dataset does not fit in DRAM. The NVM execution is done in-place on the persistent memory space, while the MM uses the *Memory Mode* transparent behaviour available in the Optane DC devices. The result may be held in DRAM (*volatile result* executions) or stored in the object store (*store result* executions).

This is the first application that has a sizeable output, so a distinction is being made on both *DAOS* and *dC NVM*: the result may be stored in the object store (*store result*) or kept as volatile (in the DRAM). This distinction is not applicable to neither the DRAM executions nor the MM ones, as neither of those modes have different addressable memory spaces. Also note that having the output result in DRAM is only possible if the resulting data structure is smaller than the available DRAM.

The decisions on where to store the result will be driven by the amount and availability of system memory –if there is no shortage of memory, then the object store could always choose the fastest memory available, but there will typically be restrictions in-place: memory footprint may be restricted or there may exist certain prioritization.

We will be doing a static decision for the volatile vs store. More complex applications would require additional thoughts related to where to place result data structures. The decision can be entirely developer driven, it can be decided by the software stack, or it can be somewhat in-between by including annotations and reacting to them with runtime information.

The active configurations have a much better performance than the non-active ones, specially when the result is stored in DRAM (*volatile result*) –that active configuration is 15 times faster than the best non-active one. Given that the matrix addition is a memory-bound application, this result could be expected: avoiding memory transfers is greatly beneficial to the performance, something that the data locality exhibited by the active object store is able to achieve. Moreover, having the result in DRAM while accessing the input dataset through the NVM is a great way to optimize the available bandwidth: the execution is performing all read operations from the NVM while performing write operations to DRAM, avoiding the slower write operations on the NVM and avoiding contention in both memories. That is why this execution mode (*dC NVM (volatile result)*) has the best performance.

As done with previous applications, we include the *method* execution times in Figure 6.13. These results illustrate the high cost of write operations onto the NVM for the following reason: the first three execution modes (*DRAM*, *NVM*, and *MM (hot)*) are very similar in performance, but they read the input submatrices from different memory spaces (*DRAM* reads them from system memory, *NVM* reads them from the NVM, and *MM (hot)* is expected to use DRAM as it is running *hot*). However, they all have in common that they write the output object to DRAM. The last mode, *MM (cold)*, shows a much worse performance due to the fact that it is writing the output object to NVM, which results in a noticeable penalty.

Insights

There are two main ideas that can be drawn from this application, ideas that can be interpolated to many memory-bound applications. First of all, data locality (and thus, an active storage system) can bring great benefits to such kind of application thanks to the reduction of data transfers and memory copies –memory operations that are the bottleneck of memory-bound applications. Secondly, one can generally expect a trade-off between

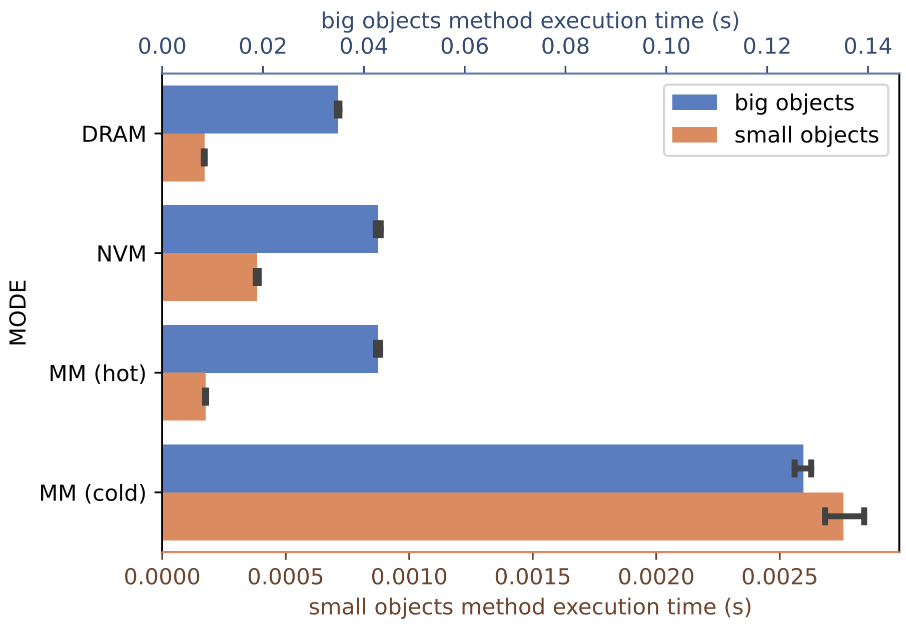


Figure 6.13: Execution times for the matrix addition *method*. The DRAM execution is done with no assistance of other memory tiers. The NVM execution is done in-place on the persistent memory space. The MM executions correspond to the transparent *Memory Mode* available in the Optane DC devices, which resembles a caching –hence the hot/cold executions.

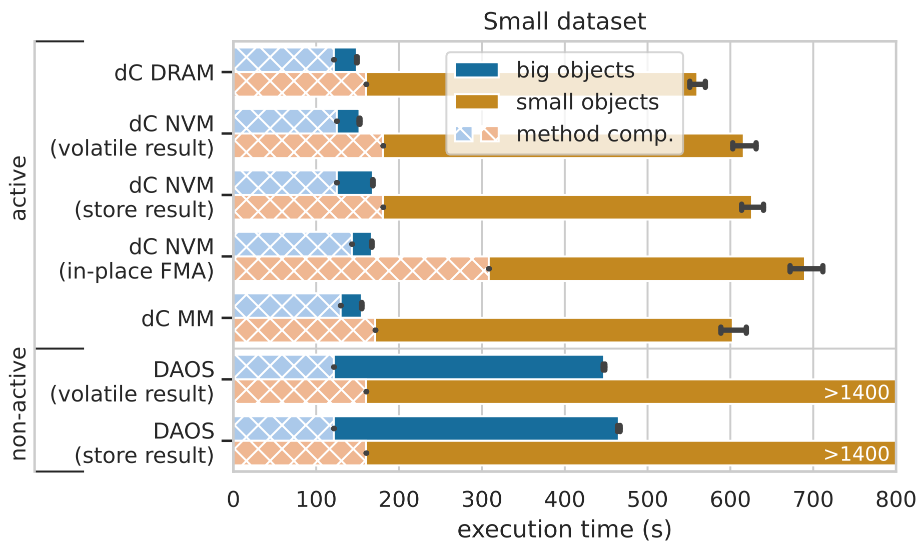


Figure 6.14: Execution times of the matrix multiplication application for a small dataset ($n = 42000$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The DRAM execution uses no additional memory tier. The NVM execution is done in-place on the persistent memory space. The MM execution uses the transparent mode of the Optane DC called *Memory Mode.* The result may be held in DRAM (*volatile result* executions) or stored in the object store (*store result* executions). The *store result* execution corresponds to an execution which evaluates the output object in DRAM and, after the *active method* invocation has finished, transfers it to the object store. The other execution, *in-place FMA*, performs all the operation in-place in the NVM, without having any input or output structure in DRAM.

the memory footprint and performance for such kind of applications. Given the NVM high write cost, it is beneficial to perform all write operations onto fastest memory (in our specific scenario, that means having the output dataset in DRAM). Just as already discussed in the previous applications, we can observe how all active scenarios outperform the non-active one. The active object store has some additional potential which, in this case, allows for a further performance improvement by leveraging both memory tiers, which sidesteps the bottleneck of write bandwidth.

6.5.4 Matrix multiplication

The different execution times can be seen in Figure 6.14 (small dataset) and Figure 6.15 (big dataset). The data structures of matrix multiplication (input dataset, output dataset, and objects) are identical to the previous application, matrix addition; however, the memory access pattern and the computation requirements are completely different.

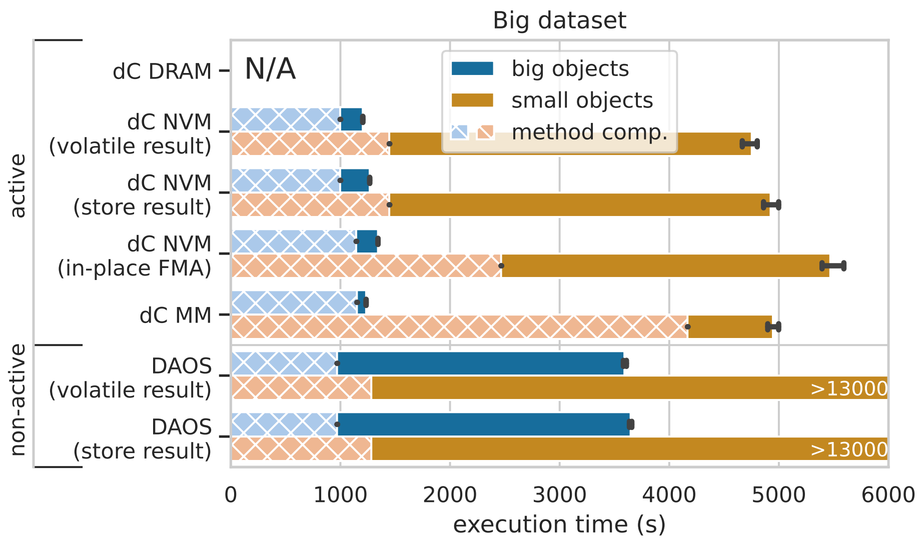


Figure 6.15: Execution times of the matrix multiplication application for a big dataset ($n = 84000$). The dataClay (dC) executions use the active object store in different configurations, while DAOS is the non-active one. The big dataset does not fit in DRAM. The NVM execution is done in-place on the persistent memory space, while the MM uses the *Memory Mode* transparent behaviour available in the Optane DC devices. The result may be held in DRAM (*volatile result* executions) or stored in the object store (*store result* executions). The *store result* execution corresponds to an execution which evaluates the output object in DRAM and, after the *active method* invocation has finished, transfers it to the object store. The other execution, *in-place FMA*, performs all the operation in-place in the NVM, without having any input or output structure in DRAM.

We can see a consistent good behavior of the active object store, specially when using big objects where active executions are 3 times faster than the non-active ones. The object reuse is something that amplifies the costs of the non-active approach, an aspect that we had already seen in the k-means application (Section 6.5.2).

There is a huge difference between executions with big objects and small objects: active executions are almost 5 times slower when using small objects than the ones using big objects. This is due to an increase on the *active method* invocation count and general memory access: having more objects implies a superlinear increase on invocations and object reuse (a behavior inherent to the matrix multiplication algorithm).

In the previous matrix addition application we had observed a big impact when having the output dataset volatile or persisted in the NVM. This difference is softened and it becomes almost inappreciable. In this case, the long computations required by the matrix multiplication decreases the gap between the different memory configurations –they all are within 15% between one another for the small objects, and only 10% for big objects. This remains true even while doing a write-heavy multiply-and-add operation in-place in the NVM (labeled *in-place FMA*); this configuration is more taxing regarding write operations on the NVM but it yields a small overhead in relative terms.

We have already discussed the programming and development aspect of the volatile vs store placement of the result (see matrix addition evaluation 6.5.3), and the same considerations apply for the in-place FMA configuration.

The fastest memory configuration is to store the output matrix in DRAM (*volatile result*). However, the decreased gap between executions implies that we can perform a huge matrix multiplication with virtually zero DRAM footprint with a bearable performance degradation, as shown by the *in-place FMA* execution times.

As in previous applications, we include the *method* execution time analysis in Figure 6.16. The *method* execution when done with big objects is almost the same (within 10%) across all configurations, something that matches the behavior of the application and is explained due to the high computation to data ratio. For small objects, where the absolute execution times are lower, the impact of the memory configuration is much more visible; in both object sizes *NVM (in-place FMA)* and *MM (cold)* are the slower configurations, but the relative difference is much more apparent when using small objects which are between 2× and 3× slower.

Insights

This application has shown how, for scenarios with high computation requirements and significant data reuse, the data locality plays a main role in the overall performance while the exact memory configuration is secondary. Even more important than the available memory and its configuration is the object size, which will impact the quantity of *active method* invocations and it can have a dramatic impact on overall performance.

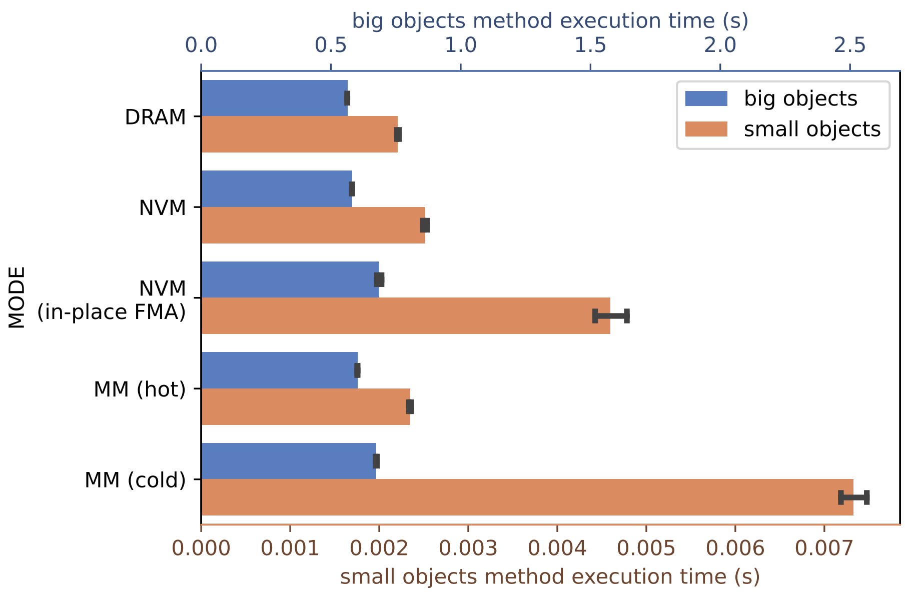


Figure 6.16: Execution times for the matrix multiplication *method*. The DRAM execution is done with no assistance of other memory tiers. The NVM execution is done in-place on the persistent memory space. The MM executions correspond to the transparent *Memory Mode* available in the Optane DC devices, which resembles a caching –hence the hot/cold executions. The *method* may perform the fused-multiply-add operation in DRAM or in-place on the NVM; this last execution corresponds to the *in-place FMA* one in the figure.

6.6 Summary

In this chapter we have shown a mechanism that leverages non-volatile memory within the object store. This software stack has demonstrated very good performance results. The evaluation has been conducted with different applications, representing a vast set of applications in HPC, data analytics, and machine learning domains.

As the evaluation has shown, the benefits obtained by this design will vary depending on the application. However, in all scenarios, we have observed that the active features provide substantial gains. We have experiments that show 10% improvement for an application with no data reuse and high computation data ratio (represented by the Histogram) and up to an order of magnitude for a memory-bound application under the most appropriate memory configuration (matrix addition with the result placed in DRAM).

On top of the aforementioned benefits, obtained through the *active* capabilities of the object store, we have also shown different memory configurations and the impact that an NVM device can have. The byte-addressable nature of this memory tier enables in-place execution; we have demonstrated how the active object store is able to execute applications with datasets bigger than system memory at near-DRAM speed.

After the evaluation we can conclude that there are two main data locality improvements that are considered: on the one hand, the *active* capabilities avoid transfers between the storage system and the application –that is regardless of the exact memory configuration or the presence of NVM devices. On the other hand, the byte-addressable memory avoids data copies between memory tiers (DRAM and NVM) within the storage system itself. This last aspect shows a great potential but can also backfire, as it is a slower memory and for instance the k-means has shown how an explicit memory copy can improve the performance by a $2\times$ factor.

As previously stated, memory-bound applications have great performance improvements but there are also gains for compute-bound applications as well. An example of such kind of application is showcased in the matrix multiplication application. Since the data locality overall impact is reduced due to the higher computation cost, compute-bound applications show lower sensitivity to memory speeds. We have seen how those kind of applications can be run almost entirely from within the NVM (i.e., huge datasets, with minimal memory footprint) while performance degradation is as little as 10% in our experiments –and still more than $2\times$ faster than non-active executions.

Having support for the persistent memory from the object store is a great benefit for programmability. Not only we are able to showcase the benefits unlocked by using NVM within the object store, but that can be managed and encapsulated by the objects and interfaces on dataClay. We have to keep in mind that the alternative is either to manually manage the *App Direct* mode or, alternatively, rely on a sub-par automatic cache provided by the *Memory Mode*. Having dataClay as means to encapsulate that provides performance portability to the application source code and maintains all the benefits, as we have shown in this chapter.

Chapter 7

Conclusions

In this thesis we have unfolded a software architecture based on an active object store that can be used in the HPC ecosystem; we have shown that this proposed software stack is able to yield good results and good performance in a variety of scenarios. Moreover, we show how this main software infrastructure can be enriched, on the active object store part, with several improvements to further increase its performance and unleash its potential.

The software stack in this thesis builds upon dataClay, an active object store heavily rooted in the object oriented paradigm. Most scientific applications are already using mainstream libraries which already embrace object encapsulation for their bindings and interfaces; moreover, the semantics of object oriented paradigm result in cleaner code, clear intentions and easier maintenance on the application developer side.

Conditioning dataClay for HPC along the integration of COMPSs and dataClay (as discussed in Chapter 4) constitutes the first contribution of this thesis **C1**: *Design the integration between a Python object oriented storage system and a distributed task-based programming model.* The evaluation of this integration shows the potential across several scientific applications. This is the tip of the iceberg regarding **RQ1**: *How can object stores be more efficiently used within HPC environments?*, as most of this thesis show more contributions that improve the object store; these contributions show that object stores can be very suitable for HPC environments.

The *active* mechanism is, at its core, a software improvement. It is an example of what kind of improvement can be integrated into the object store. This software improvement is an item of the answer of **RQ2**: *How can software improvements be supported by object stores in order to improve performance?.*

A novel improvement that we propose (also related to answering **RQ2**) is the *SplIter* mechanism, a way to enhance iteration, which is contribution **C2**: *Propose the programming interface required to improve the iteration and execution across distributed datasets through object store cooperation.* The design of this approach is explained in Chapter 5. This proposal is able to break the dependency between data granularity and task granularity; this reduces execution performance sensitivity to the block size of the data, which is specially relevant in distributed environments and data-intensive applications.

The object store can also take advantage of new hardware improvements. For instance, in Chapter 6 we propose a mechanism that allows dataClay to use its *active* mechanism in order to execute code directly unto non-volatile memory. This is a way to support a hardware improvement from the object store –answering **RQ3**: *How can hardware improvements be supported by object stores in order to improve performance?*– and constitues contribution **C3**: *Propose the mechanism to support non-volatile memory from an active object store.*

Quantifying the improvement expected by maximizing data locality (the answer to **RQ4**: *How much of an improvement can we expect when maximizing data locality during the execution of HPC applications?*) is difficult to define precisely. We have seen –along the evaluations from Chapters 4, 5, and 6– that increasing data locality can have a sizeable impact in performance and execution times. With these different evaluations in mind, we can conclude that data-intensive applications with fragmented datasets are the scenarios where we will see bigger improvements (orders of magnitude better performance). On the other side of the spectrum, compute-intensive applications tend to be less affected by the data locality –which is obvious if we think of it in terms of Amdahl's law, given that this kind of configurations will be performing computation most of the application execution time regardless of the software stack details.

7.1 Future work

This thesis establishes the groundwork for using object stores in HPC environments in a suitable manner. With our contributions as the starting point, there are several lines of future work that are worth exploring.

7.1.1 Flexible typing

There are certain schema requirements on dataClay, which are implemented through typing. This was presented in section 4.1.4. That section already hints at certain aspects of future work; said future work has already been incorporated into dataClay starting from version 3 onwards.

There has been an explosion of typing semantics on Python, both from the point of view of code development (resulting in cleaner more maintainable code) and from the point of view of runtime (allowing new automatic validation routines with unnoticeable development overhead). Evaluating, from the point of view of dataClay, the different options and establishing the comprehensive list of supported features and annotations is something that is required in order to truly support Python 3.x typing mechanisms.

7.1.2 Comparison and synergies with COMPSs cache

Recent COMPSs versions (3.x) have an stable feature which is the object cache. This cache exists at the worker nodes and is used in order to hold objects (parameters and returns). By having that object cached, a subsequent task is able to reuse an object avoiding the additional deserialization that would be needed without the cache.

The merits that this cache provides has certain overlap with the benefits that one can obtain by using persistent objects and cache. Namely, they both improve data locality by avoiding data transfers and by reducing the need of data serialization/deserialization.

However, there are no efforts nor evaluation (yet) towards integrating the COMPSs cache with dataClay. There might be synergies to be found by integrating the COMPSs cache semantics with the Storage API, or they may be direct competitors. Further research is needed in order to understand the relationship between the two approaches.

7.1.3 dataClay native dislib array

There is already some work towards a tighter integration between the *dislib* library and dataClay. In fact, some work is already available in a public repository, a dislib fork[13].

This current fork has been used for the applications in chapters 4 and 5. However, there is still some work to do towards standardizing the interfaces and making it seamless for the application developer.

Having a dataClay array that is compatible with the dislib algorithms (and in general, behaves identically to the dislib array built-in data structure) would lower the entry barriers for application developers. Moreover, this dataClay array could already include the contributions of this thesis; for instance, that means that application developers could benefit from the *SplIter* mechanism without needing to change their codebase.

7.1.4 Contribute *SplIter* to Dask

In Chapter 5, we have used Dask as another distributed execution framework on which we have evaluated the *SplIter* contribution. Because this was done as a proof of concept (and to establish a baseline), all required modifications were embedded into the application code.

After the evaluation on that chapter, we concluded that *SplIter* has a significant merit and is able to improve the performance of both Dask and COMPSs & dataClay. It is possible to pack the required modifications and include them into the Dask codebase. Integrating these kind of changes into a community project as widely used as Dask requires more in-depth knowledge on Dask runtime and some polishing on the source code rough edges. This is a technical implementation process, as the design is already completed.

7.1.5 Single process "truly multithreaded" backend implementation

In this thesis, we have introduced certain quirks related to multi-threading and the Global Interpreter Lock. Those were relevant when conditioning dataClay for HPC and implementing Python backends (introduced in Chapter 4, where the GIL mechanics were explained on Section 4.1.2).

In recent developments, there have been some efforts towards relaxing the GIL constraints; the PEP 684[75] discusses the options and general roadmap required in order to have a GIL per-interpreter. The key aspect of this change is that a single Python process can have multiple Python interpreters, each with its own state. Having a GIL per-interpreter (instead of per-process) opens the door to leveraging multithreading (true multithreading) at that level, which would increase performance through parallelism in pure-Python routines.

Having support for multi-interpreters requires a considerable engineering effort, so there is a need to first understand the advantages of having multiple interpreters and the scope of those: should we have a single interpreter per user? one interpreter per dataset? pre-instantiate one interpreter per available core? dynamically create interpreter per each RPC? It is imperative to understand the overhead and bottlenecks of the interpreters and the benefits that can be expected with a "more multi-threaded" backend –although those benefits will depend on the application implementation and the libraries used in each application.

An alternative approach is outlined in PEP 703[40], which proposes build options to make the GIL entirely optional. This is synonymous to having a thread-safe interpreter. This is something to keep an eye on, but its roadmap is a more long-term one: 2024 for the first CPython version with the `--disable-gil` –which implies having two different ABIs– and GIL disabled by default on 2030 –which may seem optimistic. Given that dataClay does not use this ABI directly, having support for this no-GIL CPython interpreter seems trivial; however, it remains an open question whether mainstream 3rd party libraries (such as the numerical libraries that HPC applications use constantly) would be interested in supporting this new ABI.

7.1.6 Replicas and scatter mechanisms for execution performance

The concept of doing replicas is already present on both dataClay and COMPSs. The current Storage API has the mechanism for allowing the execution runtime to ask for replicas unto the storage layer.

However, we can look at Dask[8] and see their `scatter` mechanism, a function that distributes data in round-robin fashion. This is an easy solution to balance data and it can be combined with its `broadcast` mode in which data is not distributed but replicated unto several workers.

Having support for a similar feature from COMPSs would require more analysis in both the COMPSs side and the dataClay side (dependencies are known by COMPSs, while data distribution and replicas are carried out by dataClay). Offering this to the application developer would need new semantics at the programming model level and special care on mutable objects –a consideration that Dask does not require given that data is always considered to be immutable, something that is not generally true under the object-oriented programming paradigm.

7.1.7 Work stealing

The *SplIter* contribution **C2** has been approached from a data-centric point of view; more specifically, it maximizes locality and minimizes data transfers.

This approach has shown promising results, as discussed in Chapter 5. However, looking it from a computational point of view, the *SplIter* (and the concept of partitions related to it) could be used in order to efficiently balance computation costs across clusters.

The concept of work stealing is born from the idea of transferring blocks of work from one partition to another during runtime. This requires coordination between the partitions (i.e. the active object store) and the computing framework. Ideally, this work stealing happens between nodes that are saturated (givers) and nodes that have already finished (takers).

The idea to perform work stealing is specially appealing when there is any kind of unbalance. This may exist from the algorithm point of view –e.g. in a scenario where there is a simulation grid and certain areas are more expensive to compute than other ones– or from the computation point of view –e.g. certain nodes are less powerful or are more busy than other ones.

Performing adaptative scheduling of data is not a new concept (as discussed in the State of the Art, section 2.2.3) but doing it on top of the *SplIter* unlocks additional potential with a low entry cost given that the logic partitioning is already done and proven to be effective by this thesis.

7.1.8 Multi-tier and dynamic object placement

Initially, dataClay was designed focusing in two clear memory tiers: memory and disk. Objects are held in memory when possible (and methods are executed in-memory) while disk is used for both persistence and spill. This ensures that objects are persisted across restarts and also that the system does not easily run out of memory.

In Chapter 6 we discussed the static placement into NVM. This can be seen as a new tier. Still, there is future work on deciding placement dynamically; this can be complex depending on which kind of inputs are used for the decision-making –runtime heuristic, static information, user hints, etc.

In addition to placing the object dynamically, we could expand the number of tiers to be used, both intranode and internode. For instance, a deep multi-tier environment could be composed of (sorted by latency) memory, NVM, NVMe, local SSD, network file system, cloud storage. There is work on managing those different memory tiers (discussed in the State of the Art, section 2.2.5) but dataClay offers a unique storage system that can be used as foundation for further work on this topic.

7.2 Results and dissemination

7.2.1 Publications

The following peer-reviewed publications are directly related to contributions of this thesis:

- Jonathan Martí, Anna Queralt, Daniel Gasull, **Alex Barcelo**, Juan José Costa, and Toni Cortes. “dataClay: A distributed data store for effective inter-player data sharing.” Journal of Systems and Software 131 (2017): 129-145.
  Impact Factor: 2.28

This is the most detailed early article describing dataClay, with a heavy focus on data-sharing. The integration of dataClay and COMPSs is discussed and the performance evaluation is performed within an HPC cluster. Those first evaluations are in Java. The Python design and implementation presented in the article is contribution **C1**, contribution that this thesis explains in much more detail in Chapter 4

- **Alex Barcelo**, Anna Queralt, and Toni Cortes. "Enhancing iteration performance on distributed task-based workflows." Future Generation Computer Systems 149 (2023): 359-375.
  Impact Factor: 7.5

  This article presents the *SplIter*, the programming interface for improving iteration and execution across distributed datasets and shows its evaluation (contribution **C2**). This corresponds to the Chapter 5.

- **Alex Barcelo**, Anna Queralt, and Toni Cortes. "Revisiting active object stores: Bringing data locality to the limit with NVM." Future Generation Computer Systems 129 (2022): 425-439.
  Impact Factor: 7.5

  This article presents the mechanism to support non-volatile memory and evaluates and characterizes its performance benefits (contribution **C3**). This corresponds to the Chapter 6.

7.2.2 Source code and implementation details

All source code is available under open source licenses. There are different aspects that have been made publicly available for further research and/or as to provide implementation details:

- `pyclay` repository: `https://github.com/bsc-dom/pyclay/`

  The implementation of dataClay under Python (and adaptation for the PyCOMPSs task-based programming model) is available in this GitHub repository under BSD 3-clause license. This is the contribution **C1** (full architecture in addition to implementation). The code available in that repository combines a certain engineering effort (performed by us, the Distributed Objects Management group in BSC) with design decisions discussed in Chapter 4.

- *SplIter* application repository:
  `https://github.com/bsc-dom/split-miniapps/`

  The different applications used in Chapter 5 are available in this GitHub repository, and made available as public domain. There is a specific branch for the Dask implementation as well as the `main` one for dataClay & COMPSs. The code contain implementation details for the *SplIter* proposal (part of contribution **C2**) as well as all the application code necessary to replicate the evaluation.

- NumPy Persistence to Non-Volatile Memory
  `https://github.com/bsc-dom/npp2nvm`

  A Python package that encapsulates the procedure of persisting numpy arrays to Optane DC (NVM). This code is published under the Apache 2.0 License and can be used standalone (without dataClay). It can be used within dataClay, and that configuration is the one used during the evaluation of Chapter 6. The npP2NVM package is part of the implementation effort of contribution **C3**.

- Optane DC application repository:
  `https://github.com/bsc-dom/optanedc-miniapps`

  The different applications used in Chapter 6 are available in this GitHub repository, and made available as public domain. This source code can be used in order to replicate the evaluation shown there; it can also be used as reference implementation for contribution **C3**.

7.2.3 Usage in national and European projects

Different parts and contributions of this thesis have been used and are part of the following projects:

- BigStorage
- EXPERTISE
- NextGenIO
- mF2C
- CLASS
- ELASTIC
- eFlows4HPC
- ADMIRE
- ICOS

The implementation details in each case can be found either in the project's public repository (open source) and/or they are already integrated in the repositories mentioned in the previous subsection.

7.3 Funding and acknowledgements

This work has been partially funded by the European Commission's Horizon 2020 Framework program and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558 and by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR (PCI2021-121957), project eFlows4HPC. The authors also acknowledge financial support by the Spanish Government (contract PID2019-107255 GB and TIN2015-65316-P) and by Generalitat de Catalunya (contract 2017-SGR-01414). Anna Queralt is a Serra Húnter Fellow.

This work has been partially supported by Intel® under the BSC-Intel collaboration on 3D XPoint™ Technology.

Bibliography

[1] DAOS administration guide. https://daos-stack.github.io/admin/intro/. Accessed: June 2020.

[2] NVM Programming Model (NPM) v1.2. Tech. rep., Storage Networking Industry Association, June 2017.

[3] Alcorn, P. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html. [Last Update: 2019-04].

[4] Álvarez, J., Solà, S., Álvarez, P., Castro-Ginard, A., and Badia, R. M. dislib: Large Scale High Performance Machine Learning in Python. In *Proceedings of the 15th International Conference on eScience* (2019), pp. 96–105.

[5] Amazon. Amazon Simple Storage Service. https://aws.amazon.com/s3/. Accessed: 2020.

[6] Anaconda, Inc. Dask.distributed. https://distributed.dask.org/. Accessed: 2023.

[7] Anaconda, Inc. and contributors. Best Practices – Dask Documentation. https://docs.dask.org/en/stable/array-best-practices.html. Accessed: 2023.

[8] Anaconda, Inc. and contributors. Dask documentation. https://docs.dask.org/en/stable/. Accessed: 2022.

[9] Aumage, O., Carpenter, P., and Benkner, S. Task-Based Performance Portability in HPC: Maximising long-term investments in a fast evolving, complex and heterogeneous HPC landscape. Tech. rep., European Technology Platform for High Performance Computing (ETP4HPC), 2021.

[10] authors, C., and contributors. Ceph Documentation. https://docs.ceph.com/. Accessed: January 2023.

[11] Badia, R. M., Conejero, J., Diaz, C., Ejarque, J., Lezzi, D., Lordan, F., Ramon-Cortes, C., and Sirvent, R. COMP Superscalar, an interoperable programming framework. *SoftwareX 3* (2015), 32–36.

[12] Banicescu, I., and Velusamy, V. Performance of scheduling scientific applications with adaptive weighted factoring. In *Parallel and Distributed Processing Symposium, International* (2001), vol. 3, IEEE Computer Society, pp. 791–792.

[13] Barcelo, A. dislib (fork). `https://github.com/alexbarcelo/dislib`, 2023.

[14] Braam, P. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[15] Breitenfeld, M. S., Fortner, N., Henderson, J., Soumagne, J., Chaarawi, M., Lombardi, J., and Koziol, Q. DAOS for extreme-scale systems in scientific applications. *arXiv preprint arXiv:1712.00423* (2017).

[16] BSC. MareNostrum. `https://www.bsc.es/marenostrum/marenostrum`. Accessed: 2022.

[17] BSC – Distributed Objects Management group. Applications used for evaluation of NVM. `https://github.com/bsc-dom/optanedc-miniapps/`, 2020.

[18] BSC – Distributed Objects Management group. dataClay Java codebase. `https://github.com/bsc-dom/javaclay/`, 2022.

[19] BSC – Distributed Objects Management group. dataClay Python codebase. `https://github.com/bsc-dom/pyclay/`, 2022.

[20] BSC – Distributed Objects Management group. dataClay Python package. `https://pypi.org/project/dataClay/`, 2022.

[21] BSC – Distributed Objects Management group. Applications used for evaluation of the COMPSs dataClay integration as well as SplIter. `https://github.com/bsc-dom/split-miniapps/`, 2023.

[22] BSC – Workflows and Distributed Computing group. COMPSs documentation. `https://compss-doc.readthedocs.io/`. Accessed: 2022.

[23] BSC – Workflows and Distributed Computing group. dislib source code repository. `https://github.com/bsc-wdc/dislib/`, 2023.

[24] Cariño, R. L., and Banicescu, I. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing 44*, 1 (2008), 41–63.

[25] Chan, E. Apache Cassandra for analytics: A performance and storage analysis. `https://www.oreilly.com/content/apache-cassandra-for-analytics-a-performance-and-storage-analysis/`. [Published: February 10, 2016].

[26] Conejero, J., Corella, S., Badia, R. M., and Labarta, J. Task-based programming in COMPSs to converge from HPC to big data. *The International Journal of High Performance Computing Applications 32*, 1 (2018), 45–60.

[27] Dask core developers. Dask-ML. `https://ml.dask.org/`. Accessed: 2023.

[28] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Symposium on Operating Systems Design and Implementation* (2004).

[29] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

[30] Docan, C., Parashar, M., Cummings, J., and Klasky, S. Moving the Code to the Data - Dynamic Code Deployment Using ActiveSpaces. In *2011 IEEE International Parallel & Distributed Processing Symposium* (2011), IEEE, pp. 758–769.

[31] Dong, B., Byna, S., Wu, K., Johansen, H., Johnson, J. N., Keen, N., et al. Data elevator: Low-contention data movement in hierarchical storage system. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)* (2016), IEEE, pp. 152–161.

[32] Fisk, N. *Mastering Ceph.* Packt Publishing Ltd, 2017.

[33] Fix, E., and Hodges, J. L. Nonparametric discrimination: consistency properties. *Randolph Field, Texas, Project* (1951), 21–49.

[34] Gagliardi, F., Moreto, M., Olivieri, M., and Valero, M. The international race towards exascale in europe. *CCF Transactions on High Performance Computing 1*, 1 (2019), 3–13.

[35] Gatziu, S., and Dittrich, K. R. SAMOS: An active object-oriented database system. *IEEE Data Eng. Bull. 15*, 1-4 (1992), 23–26.

[36] Google. Google python style guide. GitHub repository, 424ad34 commit, 2012. `https://github.com/google/styleguide/blob/424ad34/pyguide.html`.

[37] Google. Google python style guide. GitHub repository, gh-pages branch, 2022. `https://github.com/google/styleguide/blob/gh-pages/pyguide.md`.

[38] Google LLC. Protocol buffers. `https://protobuf.dev/`. Accessed: 2023.

[39] Graf, H., Cosatto, E., Bottou, L., Dourdanovic, I., and Vapnik, V. Parallel Support Vector Machines: The Cascade SVM. *Advances in neural information processing systems 17* (2004).

[40] Gross, S. Making the Global Interpreter Lock Optional in CPython. Tech. Rep. 703, 2023. `https://peps.python.org/pep-0703/`.

[41] gRPC Authors. gRPC. `https://grpc.io/`. Accessed: 2023.

[42] Hastings, L. Removing Python's GIL: The Gilectomy. PyCon, 2016.

[43] Herodotou, H., and Kakoulli, E. Automating distributed tiered storage management in cluster computing. *Proceedings of the VLDB Endowment 13*, 1 (2019), 43–56.

[44] Hummel, S. F., Schonberg, E., and Flynn, L. E. Factoring: A method for scheduling parallel loops. *Communications of the ACM 35*, 8 (1992), 90–101.

[45] Ilkbahar, A. Intel® Optane™ DC Persistent Memory Operating Modes Explained. `https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/`, 2018. [Accessed: 2020-02].

[46] Intel Corporation. Intel® Optane™ Persistent Memory Product Brief. `https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html`. [Accessed: 2021-09].

[47] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[48] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014), pp. 1–11.

[49] Kale, L. V., and Krishnan, S. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* (1993), pp. 91–108.

[50] Kannan, S., Gavrilovska, A., Schwan, K., Milojicic, D., and Talwar, V. Using active nvram for i/o staging. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities* (2011), pp. 15–22.

[51] Kannan, S., Milojicic, D., Talwar, V., Gavrilovska, A., Schwan, K., and Abbasi, H. Using active NVRAM for cloud I/O. In *2011 Sixth Open Cirrus Summit* (2011), IEEE, pp. 32–36.

[52] Keeton, K., Patterson, D. A., and Hellerstein, J. M. A Case for Intelligent Disks (IDISKs). *Acm Sigmod Record 27*, 3 (1998), 42–52.

[53] Kougkas, A., Devarajan, H., and Sun, X.-H. Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (2018), pp. 219–230.

[54] Liu, J., Koziol, Q., Butler, G. F., Fortner, N., Chaarawi, M., Tang, H., Byna, S., Lockwood, G. K., Cheema, R., Kallback-Rose, K. A., et al. Evaluation of HPC application I/O on object storage systems. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)* (2018), IEEE, pp. 24–34.

[55] Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., and Barton, E. DAOS and Friends: A Proposal for an Exascale Storage System. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 585–596.

[56] Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Alvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., and Badia, R. M. ServiceSs: an interoperable programming framework for the Cloud. *Journal of grid computing 12* (2014), 67–91.

[57] Martí, J. *dataClay: next generation object storage.* PhD thesis, Universitat Politècnica de Catalunya, 2017.

[58] Martí, J., Queralt, A., Gasull, D., Barcelo, A., Costa, J. J., and Cortes, T. dataClay: A distributed data store for effective inter-player data sharing. *Journal of Systems and Software 131* (2017), 129–145.

[59] MinIO, Inc. MinIO. `https://min.io/`. Accessed: 2021.

[60] Nair, R., Antao, S. F., Bertolli, C., Bose, P., Brunheroto, J. R., Chen, T., Cher, C.-Y., Costa, C. H., Doi, J., Evangelinos, C., et al. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development 59*, 2/3 (2015), 17–1.

[61] Nider, J., Mustard, C., Zoltan, A., and Fedorova, A. Processing in storage class memory. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (2020).

[62] OpenStack. Storlets' Documentation. `https://docs.openstack.org/storlets/latest/`. Accessed: 2020.

[63] OpenStack. Swift. `https://wiki.openstack.org/wiki/Swift`. Accessed: 2020.

[64] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[65] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and

Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[66] Perez, J. M., Badia, R. M., and Labarta, J. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE international conference on cluster computing* (2008), IEEE, pp. 142–151.

[67] Python core developers and the Python community, supported by the Python Software Foundation. CPython. `https://github.com/python/cpython`. Accessed: 2023.

[68] Python Software Foundation. The python language reference. `https://docs.python.org/3/reference/index.html`. Accessed: 2023.

[69] Python Software Foundation. The python standard library. `https://docs.python.org/3/library/index.html`. Accessed: 2023.

[70] Raschka, S., and Mirjalili, V. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd, 2019.

[71] Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), K. Huff and J. Bergstra, Eds., pp. 130 – 136.

[72] Ross, R. B., Thakur, R., et al. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux showcase and conference* (2000), pp. 391–430.

[73] Rupprecht, L., Zhang, R., Owen, B., Pietzuch, P., and Hildebrand, D. SwiftAnalytics: Optimizing Object Storage for Big Data Analytics. In *2017 IEEE International Conference on Cloud Engineering (IC2E)* (2017), IEEE, pp. 245–251.

[74] Schmuck, F. B., and Haskin, R. L. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST* (2002), vol. 2, no. 19.

[75] Snow, E. A Per-Interpreter GIL. Tech. Rep. 684, 2022. `https://peps.python.org/pep-0684/`.

[76] Soumagne, J., Kimpe, D., Zounmevo, J., Chaarawi, M., Koziol, Q., Afsahi, A., and Ross, R. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)* (2013), IEEE, pp. 1–8.

[77] Stančin, I., and Jović, A. An overview and comparison of free python libraries for data mining and big data analysis. In *2019 42nd International convention on information and communication technology, electronics and microelectronics (MIPRO)* (2019), IEEE, pp. 977–982.

[78] Tang, H., Byna, S., Tessier, F., Wang, T., Dong, B., Mu, J., Koziol, Q., Soumagne, J., Vishwanath, V., Liu, J., et al. Toward Scalable and Asynchronous Object-centric Data Management for HPC. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2018), IEEE, pp. 113–122.

[79] Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R. M., Torres, J., Cortes, T., and Labarta, J. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications 31*, 1 (2017), 66–82.

[80] The kernel development community. Direct Access for files. `https://www.kernel.org/doc/html/latest/filesystems/dax.html`. Accessed: 2023.

[81] The Netty Project. Netty. `https://netty.io/`. Accessed: 2022.

[82] The PMDK team at Intel Corporation. Persistent Memory Development Kit. `https://pmem.io/pmdk/`. Accessed: 2020.

[83] The PMDK team at Intel Corporation. Persistent Memory Development Kit. `https://github.com/pmem/pmdk`, 2020.

[84] The PMDK team at Intel Corporation. Python bindings for the PMDK. Non-volatile memory for Python. `https://github.com/pmem/pynvm`, 2020.

[85] van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., and Sato, M. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1541–1555.

[86] van Rossum, G., Lehtosalo, J., and Langa, Ł. Type hints. PEP 484, 2014. `https://www.python.org/dev/peps/pep-0484/`.

[87] Virtual Institute for I/O. IO500. `https://www.vi4io.org/io500/`, 2020.

[88] Vora, M. N. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology* (2011), vol. 1, IEEE, pp. 601–605.

[89] Warren, R., Soumagne, J., Mu, J., Tang, H., Byna, S., Dong, B., and Koziol, Q. Analysis in the data path of an object-centric data management system. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)* (2019), IEEE, pp. 73–82.

[90] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[91] Weil, S. A., Leung, A. W., Brandt, S. A., and Maltzahn, C. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07* (2007), pp. 35–44.

[92] Weiland, M., Brunst, H., Quintino, T., Johnson, N., Iffrig, O., Smart, S., Herold, C., Bonanni, A., Jackson, A., and Parsons, M. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), pp. 1–19.

[93] Winter, C., and Lownds, T. Function annotations. PEP 3107, 2006. `https://www.python.org/dev/peps/pep-3107/`.

[94] Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., and Swanson, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[95] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), pp. 15–28.

[96] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010), 95.

[97] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[98] Ziqi, F., Wu, F., Park, D., Diehl, J., Voigt, D., and Du, D. H. Hibachi: A cooperative hybrid cache with nvram and dram for storage arrays. In *33rd Symposium on Mass Storage Systems and Technologies (MSST)* (2017), pp. 1–11.