8-2023

Performance Modeling of Inline Compression With Software Caching for Reducing the Memory Footprint in PYSDC

Sansriti Ranjan
sansrir@clemson.edu

Performance Modeling of Inline Compression with Software Caching for Reducing the Memory Footprint in pySDC

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Sansriti Ranjan
August 2023

Abstract

Modern HPC applications compute and analyze massive amounts of data. The data volume is growing faster than memory capabilities and storage improvements leading to performance bottlenecks. An example of this is pySDC, a framework for solving collocation problems iteratively using parallel-in-time methods. These methods require storing and exchanging 3D volume data for each parallel point in time. If a simulation consists of M parallel-in-time stages, where the full spatial problem has to be stored for the next iteration, the memory demand for a single state variable is $M \times N_x \times N_y \times N_z$ per time-step. For an application simulation with many state variables or stages, the memory requirement is considerable.

Data compression helps alleviate the overhead in memory by reducing the size of data and keeping it in compressed format. Inline compression compresses and decompresses the application's working set as it moves in and out of main memory. Thus, it provides the system with the appearance of more main memory. Naive compressed arrays require a compression or decompression operation for each store or load and therefore hurt the performance of the application. By incorporating a software cache and storing decompressed values of the array, we limit the number of compression and decompression operations for the stores and loads, thereby improving performance overall.

In this thesis, we build a compression manager and software cache manager for the pySDC framework to reduce the memory requirements and computational overhead. The compression manager wraps around LibPressio, a C++ compression library that abstracts all compressors. We utilize blosc, a lossless compressor for our compression manager, and build a software cache manager with various cache configurations and cache policies to work in cohesion with the compression manager. We build a performance model which evaluates the compression manager and cache manager's performance on different metrics such as compression ratio and compression/decompression time. We test our framework on two different pySDC applications — e.g., Allen-Cahn and Heat-diffusion.

Results show that incorporating compression and increasing the cache size for our applications inflates the total compressed size in bytes for the arrays and therefore reduces the compression ratio, in contrast to our expectations. However, incorporating the cache and a greater cache size reduces the number of compression/decompression calls to LibPressio as well as cache evictions, significantly reducing the computational overhead for pySDC. Thus, overall, our compression and cache manager help reduce the memory footprint in pySDC.

Future work involves looking at improving the compression ratio and using lossy compression to achieve significant reduction in memory footprint.

Dedication

This work is dedicated to my parents, sister, grandparents and all my teachers and professors throughout school and college.

Acknowledgments

I would like to thank my research advisor and committee chair, Dr. Jon C Calhoun for his unwavering support and guidance throughout graduate school. His knowledge, experience and mentoring have helped me grow immensely as an engineer, researcher, developer and continue the path of learning in this field.

I would like to thank Dr. Melissa Smith and Dr. Adam Hoover for serving on my defense committee and for providing all their support and guidance throughout. Their feedback and support in times of need helped me push through.

I would like to take this opportunity to thank the Julich Supercomuting Center, specifically Dr Robert Speck and Thomas Baumann for the collaboration and support in implementing this work. I would also like to extend heartfelt thanks to the NSF grant for funding this project.

I am truly grateful to Dr Xiao, Dr Harlan Russell and Clemson University faculty and staff for providing guidance and support throughout my undergraduate and graduate education. Lastly, I would like to extend thanks to my friends and research group members for their insightful peer interactions and invaluable support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

High Performance Computing (HPC) is the ability to "process data and perform complex calculations at high speeds" [35]. HPC systems comprise compute nodes and storage nodes which are connected by a shared network infrastructure [57]. These nodes form a cluster and leverage parallel processing and clustering techniques to perform numerous tasks concurrently. HPC frameworks create conducive environments for large-scale processing and by integrating parallel programming techniques, they greatly assist in the task of accelerating computations and solving large-scale problems [28].

Scientific applications such as climatic applications, legacy code applications are compute intensive. They simulate complex natural phenomena and require large datasets to analyze and compute. In this era of big data, these applications also see an exponential growth in their results and data. Therefore, the scientific community is increasingly reliant on HPC systems and frameworks to handle their data resources and considers these systems to be of paramount importance.

While HPC has enabled significant advancements across all fields of science and engineering by allowing researchers to simulate complex phenomena that are difficult, if not impossible, in a normal laboratory setting, the computational speed far exceeds the speed of data movement. HPC applications are generating data to the size of exascale computing, making it a reality [54]. However, the current HPC storage architecture would not scale to the emerging exascale computing systems as discovered in the simulation work in [51], [56]. The ratio of I/O bandwidth to bytes of memory capacity on Titan (the 27 petaflops (PF) system at Oak Ridge Leadership Computing Facility, OLCF) is 0.0016 and the ratio on Summit (the 200 PF system at OLCF) is 0.0001. Data generation rates are increasing faster than traditional parallel file system (PFS) ingestion capabilities [26]. Thus, data storage requirement for extreme scale HPC applications creates a memory bottleneck where the data required will not all fit in main memory and random access memory (RAM). Thus, data movement and memory bottleneck limit application performance and system throughput.

Moreover, despite improvements and innovations in processor and memory speeds, memory I/O is a performance bottleneck. This is because CPU clock speeds have improved to 100MHz and beyond 1GHz to 5.6GHz today [3]. However, memory read and write speeds have not improved at the same rate. Therefore, memory access time for read and write operations creates a bottleneck in application performances. Reducing the memory and memory I/O bottlenecks is essential to HPC and scientific applications as the scientific and engineering community advances research and discoveries in various fields.

Dimensionality reduction is a method where the number of features in a dataset are removed while retaining most of the important features [24]. It transforms data from a high-dimensional space into a low-dimensional space so that the low-dimensional representation retains meaningful properties. It is used in machine learning for training datasets to reduce the number of input variables, as machine learning applications have a large number of dimensions in feature space. Reducing these input features reduces the number of dimensions of the feature space and accelerates training and performance of machine learning applications [24]. Inter-network communications such as Infiniband enable faster communication and reduce traffic in inter-node communication for large clusters.

Dimensionality reduction, effective inter-network communication are common methodologies to alleviate performance bottlenecks. However, these techniques focus on improving the runtime and bandwidth of the application and the overall I/O performance. These techniques do not focus on alleviating the memory requirements and memory bottlenecks that occur in large-scale applications. Solid State Drives (SSDs) commonly referred to as flash storage are incorporated to reduce the burden on hard disk drives (HDDs). SSDs provide faster read and write access due to no moving parts. They help with faster file access and application acceleration. Similarly, Burst buffers (BBs) which are a tier of intermediate, high-bandwidth flash-based storage devices between the compute nodes and parallel file system (PFS) are increasingly exploited in contemporary supercomputers to bridge the performance gap between compute and storage systems [26]. However, both SSDs and burst buffers require physical upgrade to the system and are expensive.

Big Data frameworks such as MapReduce [52] have witnessed rapid development to cope with the growing data challenge. MapReduce is a programming model and implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. The framework divides the data computation in terms of a map and reduce function, and the underlying runtime system automatically parallelizes the computation [12]. Spark and Hadoop are common implementations of MapReduce framework, which through their key-value (Map) pairings and reduce operations enable faster access to parallel file systems. Furthermore, while Hadoop Distributed File System (HDFS), commonly utilized in HPC as distributed file system provides scalable and fault-tolerant architecture to store and retrieve data required by MapReduce, it further runs into data access and data movement bottlenecks that hurt the read and write operations of HDFS [22]. Thus, we need an effective methodology that reduces the memory and storage requirements and alleviated the memory bottleneck in extreme scale HPC systems.

Data compression is an effective tool for reducing memory and storage requirements [**?**]. It is the art and science of representing information in a compact form using fewer bytes. In order to compute on or analyze large data sets, applications need access to large amounts of memory. Compressing large arrays stored in an application's memory does not require hardware upgrades, while enabling the appearance of more physical memory [39]. Data compression by keeping data in compressed form helps improve the capacity of the memory system. It also reduces energy and bandwidth demands [44].

Integrating compression inside an application to shrink the application's footprint is called inline compression [39]. Inline compression is a compression technique of compressing and decompressing data in its working set as the application runs and reads or writes to arrays. In Memory computing attempts to improve the performance and energy efficiency by reducing the data movement in the conventional Von Neumann architecture.

Naive inline compression requires a compression and decompression operation for every store and load (writes and reads) respectively. This hurts the performance of the application. Storing the decompressed values for retrieval limits the number of operations and improves the performance of the application. Caching is a method for reducing the impact of data movement by keeping data needed for the computation in fast memory. The cache exists between the processor and main memory and by being close to the processor, memory speed improves, which consequently improves latency and bandwidth. However, caches have to be of small sizes to ensure the energy dissipation is less since they run at fast speeds. The intuitive reason why caching works is that data accesses in scientific domains exhibit a degree of locality [15]. This locality is either temporal locality, where the data block is accessed repeatedly over a period of time or spatial locality, where the data block is accessed sequentially from memory [3] in a contiguous region of the dataset. When a data block is accessed, if it already exists in cache, it is called a *cache hit*, otherwise it is a *cache miss* [15]. For a *cache hit*, the data is fetched directly from the cache, whereas for a *cache miss* the data block has to be loaded from main memory into the cache and then accessed. Therefore, repeated access to a data item stored in cache is much faster than fetching the data from its actual storage [15]. The ratio between cache hits and total number of data accesses is called *hit ratio*, and the ratio between cache misses and total accesses is known as the *miss ratio*. Therefore, if items kept in the cache are most frequently accessed items, the cache yields a higher *hit ratio* [20].

For large-scale applications and exascale computing, an application might spend a considerable amount of time waiting on data. Hardware caches are physically implemented in the hardware and help reduce the data movement bottleneck in the memory bus. Software caches are caches managed and controlled by the operating system or specific software systems such as middleware, file systems, storage systems and databases [14]. Software caches offer more flexibility in terms of caching strategies and algorithms. The software can adapt caching behavior based on specific requirements and workload characteristics, such as access pattern of the application. Software caches can be larger in capacity compared to hardware caches because physical limitations do not constrain them. However, their size is often limited by the available memory resources. The main difference between hardware caches and software caches is that hardware caches are physically integrated into the CPU and managed by the CPU hardware itself, while software caches are implemented in the main memory and controlled by the operating system or software applications. While both types of caches aim to reduce memory access latency and improve overall system performance, software caches offer more flexibility and potentially larger capacity.

The software cache utilizes a *cache management policy* to determine what data should be cached and when it should be fetched from or stored into the cache. For every initial compression or decompression operation of an array, if it is stored in a software managed cache, every consequent compression or decompression operation on the same array in the application is reduced as long as the array exists in the cache. All compression operations are replaced with an update (write) of the array in the cache, and decompression operations are replaced with loading (reading) the array from the cache. The software cache has a capacity, and the main objective of the *cache management policy* therefore is to maximize the *hit ratio*. This ensures that the number of compression and decompression calls are minimal, which incurs minimum overhead. This policy focuses on identifying access patterns of the application and utilizes the patterns to get the maximum hit ratio. Recency and frequency are the most common signals utilized as signals by software cache management policies. Recency looks at the probability of an array that was recently accessed will likely be accessed again in the near future. Frequency looks at the likelihood that an array that has been accessed frequently will be accessed again in the near future. Generally, frequency is measured as most applications arrays' frequency changes over time [25] and a sliding window [32] or exponential decay [29] mechanism is used to measure frequency. Therefore, choosing the optimal cache management policy and cache parameters for a software cache improves the performance of inline compression. Consequently, this improvement in performance enables applications to reduce the memory bottleneck and further see an improvement in reducing performance overheads.

This research focuses on reducing the memory bottleneck of HPC applications by incorporating inline data compression, thereby improving the memory requirements of these applications. The novelty in this work lies in the software caching mechanism leveraged by inline compression to reduce the overhead incurred due to compression. We build a compression manager and a cache manager which when embedded in an application enables the application to compress the data and reduce the memory requirements of the application while leveraging the software cache for faster access to decompressed arrays and improving the performance of the application.

The main contributions of this work are as follows:

- Explore the memory intensive nature of applications and how the memory footprint becomes a bottleneck for applications.
- Built a compression manager that utilizes compression techniques to perform inline compression on arrays of an application.
- Explore software caching to alleviate the compression manager's overhead.
- Built a cache manager that reduces the overhead of the compression manager.
- Built a performance model that assesses the performance of the compression and cache manager and utilize this model as a baseline to improve upon.

Chapter 2

Background

In this section, we describe the background to understand the main ideas in this work. The main component of this work is data compression and its impact on improving the storage requirements of High Performance Computing (HPC) applications by reducing the memory footprint. Therefore, we first describe a mathematical application framework and its characteristics which lead to a memory bottleneck. We then describe how compression can be leveraged to reduce the computational burden of the application. As naive inline compression incurs additional overhead into the application, we describe the background on caching, which enables compression to reduce its overhead and improve the performance of the application. We further delve into work done by other researchers on the same problem (reducing memory bottleneck in HPC applications by incorporating compression) and how our work builds on theirs while providing additional benefits of caching.

2.1 Differential Equations

Differential Equations are a powerful tool for modeling natural processes. All natural processes are a function of time, therefore these processes can be represented with respect to time and modeled as equations [9] as shown in equation 2.1. Natural phenomena such as heat diffusion, are represented by ordinary differential equations (ODE) and partial differential equations (PDE). Complex physical and chemical processes are also represented by ODE and PDEs.

$$\frac{dy}{dt} = f(t, y) \tag{2.1}$$

2.1.1 Solving Initial Value Problems using Spectral Deferred Correction (SDC) Methods

Initial Value Problems (IVP) represent time-dependent differential equations for a time-dependent ODE. It takes the form

$$u^{'}(t) = f(u(t), t) \quad for \quad t > t_0 \tag{2.2}$$

with initial data (condition) provided

$$u(t_0) = \eta \tag{2.3}$$

Construction of efficient, stable, and high order methods for solving IVP using systems of ODEs is a mature subject. Existing methods are divided into roughly two groups: Intrinsically high-order discretization schemes (Runge-Kutta methods) and accelerating convergence low-order schemes (Deferred Correction methods) [13]. While implicit Runge-Kutta methods are excellent choice due to their stability properties, these methods are expensive to reach convergence.

The term spectral method refers to a numerical method that is capable (under suitable smoothness conditions) of converging at a rate that is faster than a polynomial in a mesh width h. A mesh is, by definition, a set of points and cells connected to form a network as shown in figure 2.1. Each small rectangle represents a cell, and the vertices are nodes. A mesh is a fundamental element of a simulation process in visualizing PDEs. It is used to solve PDEs where each cell represents an individual solution of the equation and combining these individual solutions in the network results in a solution of the entire mesh and equation. Meshes are referred to as grids as well, and meshing (generating a 2D or 3D grid) helps discretize a PDE and analyze it with simulation. Therefore, meshes enable dividing the problem domain of the PDE into multiple subdomains by discretizing the equations and solve these discretized subdomains instead of solving the entire problem at once.

In the classical spectral method, the solution to the differential equation is approximated by a function $U(x)$ that is a linear combination of a finite set of orthogonal basis functions as shown below,

$$U(x) = \sum_{j=1}^{N} c_j \phi_j(x), \tag{2.4}$$

and the coefficients are chosen to minimize an appropriate norm of the residual function. This is sometimes called a Galerkin approach. The SDC method we discuss in this section takes a different approach and is viewed as expressing $U(x)$ as shown in equation 2.1.1 but then requiring $U^{''}(x_i) = f(x_i)$ at $N-2$ grid points, along with the two boundary conditions. The differential equation will be exactly satisfied at the grid points by the function $U(x)$, although in between the grid points the ODE generally will not be satisfied. This is called collocation (collocation problem) and is sometimes called spectral collocation [30]. Each of the points the ODE is not satisfied is called a collocation node, and these collocation nodes and the dimension of the ODE system is solved for.

Deferred Corrections are defined as follows – instead of solving an equation for the solution of the ODE, solve the equation for the error and refine the solution with the error calculated. This leads to a cheap and simple scheme for solving for the error, giving a low order approximation of the error. However, by further repeating and performing more iterations, we can solve for the error and refine the solution. This allows to generate high-order solutions from a low-order base method. SDC methods are generally aimed at solving ODEs. We can utilize it to solve PDEs by rewriting the PDE as a system of ODEs using finite differences [30]. The reader is pointed to literature in [13, 30, 10, 53] for further understanding of SDC.

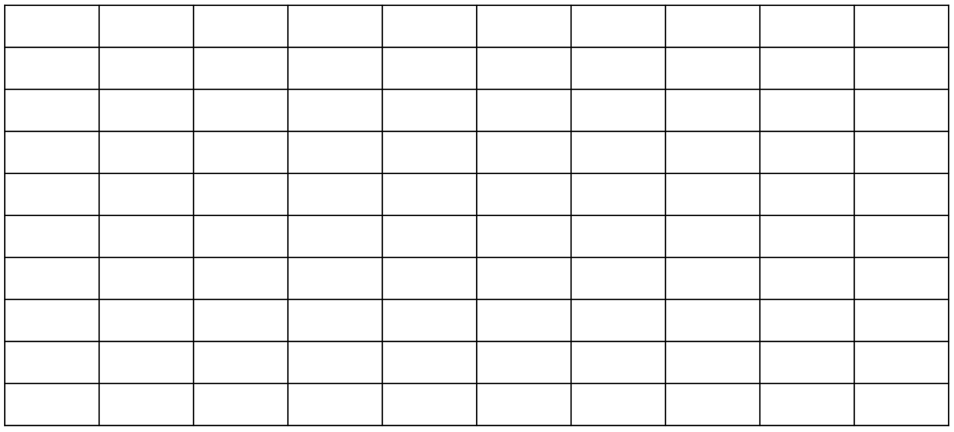


Figure 2.1: A Simple Mesh

2.1.2 Parallelizing SDC

Solving initial value problems using collocation is ideal due to the convergence and stability property [46]. Given a system of M collocation nodes and a N-dimensional system of ODEs, a system of size $MN \times MN$ has to be solved utilizing SDC. An iterative strategy here is preferred, where instead of the full system, only M smaller systems of $N \times N$ need to be solved for each iteration. This iterative structure of SDC has proven to be beneficial for algorithmic and mathematical improvements, as convergence can be accelerated by GMRES [21] and efficiency of SDC can be improved further [48]. SDC has been applied to gas dynamics and incompressible or reactive flows [8, 34, 27] as well as fast-wave slow-wave problems [40].

This iterative approach enables time-stepping, which enables efficient parallel-in-time integration [46]. Furthermore, using SDC, the "parallel full approximations scheme in space and time" (PFASST) by Emmett and Minion [16] integrates multiple time-steps simultaneously and uses SDC sweeps or iterations on a space-time hierarchy. This allows for spatial parallelization or "parallelization across the steps" of PDEs where parallelization in the temporal domain acts as a multiplier for standard parallelization techniques in space. On the other hand, "parallelization across the method" approaches parallelize the integration of each time-step individually. This leads to small-scale parallelization in the time-domain, yet parallel efficiency and applicability of these methods are more favorable. Thus, parallelizing SDC across the method and allowing the computation and update of collocation nodes concurrently is the optimal and efficient approach for solving IVPs of ODEs and PDEs.

2.2 pySDC

pySDC is a python implementation of spectral deferred correction (SDC) methods. It is a framework for simulating collocation problems faster. It allows the user to test various flavors of SDC, PFASST methods and perform parallel-in-time integration over a wide range of problems. This library is built by the scientists at Julich Supercomputing Center (JLS) [47]. For both SDC and PFASST algorithms, there are various parameters to define such as the time-step size, type, and number of collocation nodes and number of iterations for the residual (convergence condition). Moreover, the spatial problem defining the system of ODEs has to be defined and the right-hand side of the ODE has to be evaluated. pySDC as a library enables the user to choose from a multitude of options for SDC, PFASST or any other solver of collocation problems and provides as much flexibility as possible while exposing only few internals as absolutely necessary. The main website for pySDC can be found at [2] and the GitHub repository at [1]. The pySDC package is divided into seven subfolders containing various components of the framework:

- *core*: This folder contains the core components of pySDC, providing the basis for deriving custom functionality such as sweepers, problem classes, and transfer operators, but also the main structures *level* and *step*.
- *implementations*: In different subfolders, this part contains all implementations of pySDC's functionality. For example, problem classes, data types, collocation classes are defined and provided for further use. This folder should contain all implementations of relevance for multiple examples or of general interest.
- *projects*: This folder consists of all larger projects leading to publications or suitable for further demonstration. There are also specialized versions of the implementations which are used exclusively for the purpose of a single project.
- *playgrounds*: The playgrounds are a loose collection of smaller projects and code snippets.
- *tutorial*: Seven tutorials are located to provide an easier access to the theory, the code, and its functionality. The tutorials range from simple SDC runs to multi-level setups and MPI-parallel PFASST studies. Each tutorial has a specified outcome, and the whole set of introductory codes is tested each time before deployment.
- *tests*: This folder contains the tests to be run for the entire pySDC package. These consist of simple executions of tutorials and parts of the projects, as well as more direct tests of the core functionality of pySDC.
- *helpers*: Here, smaller tools and scripts are located, e.g., helping users to plot or evaluate statistics.

2.2.1 Memory Intensive Nature of SDC and pySDC applications

SDC's process of simulating collocation problems iteratively using parallel-in-time methods for arriving at the solution requires storing and exchanging of *nth* dimensional volume data for each parallel point in time. The framework is not only parallel-in-time, but also parallel-in-space. Therefore, for each time point and collocation node of the SDC problem, data is generated. If an application's simulation consists of M parallel-in-time stages (collocation nodes) and three-dimensional data, where the full spatial problem has to be stored for the next iteration, the memory demand for a single state variable is $M \times N_x \times N_y \times N_z$ per time-step. For a simulation with many state variables or stages, the memory requirement is considerable, making SDC and therefore pySDC memory-intensive and the need to reduce the memory footprint. For pySDC's memory footprint to be reduced and further improve the memory requirements, data compression is the preferred method. pySDC is an acceptable candidate for inline compression because a simulation requires the full spatial problem to be stored for multiple parallel points in time. Due to the need for each of M parallel-in-time stages to have $Nx * Ny * Nz$ data for each time-step the data required for a large simulation is massive [47].

2.3 Data Compression

Compression of scientific data is significant due to the ever-increasing volume of data generated by large-scale scientific applications. The volume of big data generated by High-Performance Computing (HPC) applications as well as application frameworks like pySDC is massive. Hardware/Hybrid Accelerated Cosmology Code (HACC) produces 40 terabytes of data in 500 snapshots with 1 trillion particles. Exascale runs estimated to be 125 trillion particles and 5 petabytes per snapshot. Data compression by representing information in a compact form enables data reduction which helps with optimal usage of storage bandwidth. There are mainly two types of data compression techniques, as described below:

- **Lossless Compression:** results in a bit wise reproducible compression/decompression operation to the original data. There is no loss of information and the data is recovered easily. Lossless compression is mainly used to compress images, sound, and text files. The amount of compression as measured by the Compression Ratio described in section 2.3.2 is low. Lossless compression is important for applications where the accuracy and preserving the context is significant, as here there is no loss in performing compression and the compressed data is equivalent to the original data.

- **Lossy Compression:** results in an output from compression/decompression that is not bitwise reproducible to the original data compression. It is used to compress large text files, audio and video files. The amount of compression as measured by the Compression Ratio described in section 2.3.2 is considerably high. If an application's accuracy is not of utmost importance and some amount of error under a threshold is allowed, lossy compression is favorable for its higher compressed size and greater reduction of data than lossless compression. It trades in accuracies for larger reductions in file sizes.

A compressor facilitates the data compression for the application. It enables the application to compress or encode data when required to be stored and decompress (decode) data when the compressed data is to be operated upon by the application.

2.3.1 Inline Compression

Data capacity and bandwidth is a constraint in High-Performance Computing (HPC) applications. A new approach to perform compression and reduce the data and memory bottleneck is performing compression and using compressed arrays throughout the application. The array is stored in a compressed array and converted to IEEE-754 decompressed arrays when required to perform computation upon [18]. This is called inline compression. Inline compression compresses and decompresses data needed by the application as it moves in and out of its working set that resides in main memory or any other block in the memory hierarchy. Therefore, after every write and store, the data is compressed and for every load, the data is decompressed. An important point to note here is that the data is always stored in its compressed state and on loading, only the decompressed values of the array are utilized, and the array still exists in its compressed state. Figure 2.2 displays how the data will be compressed and stored in main memory in the memory hierarchy.

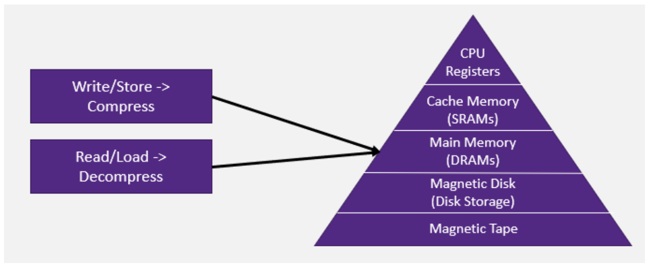


Figure 2.2: Inline Compression in Memory Hierarchy

2.3.2 Compression Metrics to Assess Performance

The compressors are assessed using the following metrics:

- **Compression Ratio** is the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression.
- **Rate**: is the average number of bits required to represent a single data sample.
- **Error**: refers to the difference between the original data and the compressed data.
- **Compressed Size**: is the number of bits required to represent the data post compression.
- **Uncompressed Size**: is the number of bits required to represent the data before compression.
- **Decompressed Size**: is the number of bits required to represent the data post decompression.
- Absolute Error: is the absolute value of the error Lossy Compressors can be further assessed through the following metrics:
- Distortion: is the difference between original data and reconstructed data after performing lossy compression. It is also referred to as fidelity and quality, and high fidelity means that the difference between the original and reconstructed data is small.

2.3.3 libpressio - Universal Compression library

libpressio [49] is a C++ library that abstracts all lossless and lossy compressors. It allows the user to set the compressor, other parameters for compression such as the error bound and the error bound method. libpressio also provides all metrics with respect to compression such as compression ratio, compressed size, uncompressed size, decompressed size in bytes as well as the compression and decompression time. The *pressio* class component of libpressio creates references to compressors, handles errors, metrics, and IO modules. *pressio* data component handles the memory management, compresses and decompresses data and *pressio* metrics measures the performance of compression and the quality of compression.

We utilize lossless compressors in our compression methodology to preserve the accuracy of applications and ensure there is no error in compressed data. We specifically utilize blosc described below with Zstd algorithm to perform lossless compression inside the application.

2.3.4 Blosc - Lossless Compressor

Blosc [7] is a high-performance computing lossless compression library that uses blocking, shuffling and compression to improve the memory access speed of data. It has been designed to transmit data to the processor cache faster than the traditional, non-compressed, direct memory fetch approach via a memcpy() OS call. Blosc reduces the size of large datasets on-disk or in-memory and accelerates memory-bound computations. It uses the blocking technique to reduce activity in the memory bus. This technique works by dividing datasets in blocks that are small enough to fit in caches of modern processors and perform compression / decompression there. It also leverages multi-threading capabilities of CPUs, in order to accelerate the compression / decompression process to a maximum.

libpressio allows incorporating Blosc's lossless compression library for lossless compression purposes. We utilize Zstandard (Zstd) compression algorithm in Blosc to preserve the fidelity of the application.

2.3.4.1 Zstandard (Zstd) – Lossless Compression Algorithm

Zstandard (Zstd) [17] is a fast lossless real-time compression algorithm, providing high compression ratios. It also offers a special mode for small data, called dictionary compression, where the compressor represents repeated data by their position in the dictionary. Zstd trades compression speed for stronger compression ratios. It is configurable by small increment. Decompression speed is preserved and remain roughly the same at all settings. We incorporate Zstd due to its lossless nature and providing fast compression speeds and high compression ratio.

While compression and decompression operations enable the application to optimally utilize memory resources, these operations incur an additional overhead. Decompression operations add to access latency as to read an array, the array is decompressed and then read from. Therefore, storing the decompressed values in a software managed cache, reduces the number of operations for future accesses to the values and further improves the latency for the arrays.

2.4 Software Caching

This section describes the background on software caching. A software cache operates at the application and system software level and provides fast access or minimal latency to operate on arrays (read or write). As naive inline compression and decompression incurs an additional overhead into an application, caching the decompressed values in a software cache limits the number of compression and decompression operations which enables the application to reduce the overhead while reducing the memory requirements, thereby improving the performance of the application.

2.4.1 Cache Parameters Affecting Performance

The configuration of a cache affects its performance. It therefore is essential to choose the optimal cache parameters that optimize the cache performance and maximize the hit ratio. The different parameters affecting the cache performance and are as follows:

1. **Cache Size**: is defined as the amount of main memory data that exists in the cache. It is the number of bytes in each data block times the number of cache block or cache line.

2. **Block Size**: is defined as the number of bytes stored in each data block of the cache.

3. **Associativity**: If a cache has multiple cache blocks k at each location, k is the associativity of the cache where each location or cache line is called a set.

Caches are of different types based on the associativity. The two popular types of caches are:

1. **Direct Mapped Cache**: In these caches, an address maps exactly to one location. To determine if a value is present, the address is hashed, yielding a location inside the cache. Provided the entry is valid and the address tag matches, the value is found in the cache. Storing a single value in each cache block enables many dispirit addresses to be stored in the cache. If two addresses map to the same location, an eviction occurs and the evicted block is replaced by a new block containing the address of the load/store [39].

2. **Set Associative Cache**: Improves over a direct mapped cache by allowing multiple cache blocks at each location (set). A k-way set associative cache has k blocks in each set. To determine if a value is present in the cache, the address is partitioned into 3 bit fields: tag, index, and offset. The index determines what set the address maps to. The tag along with a dirty bit is used to determine if the address is within any block in the set. Finally, the offset is used to locate the address location inside the identified block [39].

2.4.2 Cache Policy

An important aspect of cache design is the cache replacement policy or eviction policy when the cache is full and arrays have to be evicted. Effective caching requires careful analysis of access pattern of the application and exploiting either temporal, spatial or popularity locality. Depending on the access pattern of the application's datasets and locality of data, the optimal cache insertion and deletion (eviction) policy is determined. Two of the most commonly used cache policies are as follows:

1. Least Recently Used (LRU): Based on the cache log history, the cache line containing the data block that was least recently used is evicted and replaced with the new data block. It is based on the assumption that since this data has been accessed recently, it is likely to be accessed again in the future. It tracks individual arrays and the recency of their access. Therefore, applications with good temporal locality will benefit from this policy. LRU caches have O(1) runtime complexity for insertion, access, and deletion operations.

2. Least Frequently Used (LFU): This cache policy tracks the frequency of the array and replaces the one with the least frequency. LFU caches have the highest hit-ratio for applications with constant spatial locality and have O(log n) access time for insertion, access, and deletion operation [32]. The two main limitations of LFU however are that it requires maintaining large, complex meta-data and access frequency changes over time. While various alternatives to LFU's have been developed, utilizing time locality or recency is the most popular alternative. However, [32] has developed an O(1) run-time complexity for insertion, access, and deletion operations for LFU caches which is preferable over LRU for applications with good spatial locality.

Two other common cache eviction policies are Most Recently Used (MRU) and Most Frequently Used (MFU).

For writing to a cache, there are two main policies - **write-back** cache policy and **write-through** cache policy. In a **write-back** cache, data is written and updated in the cache every time it is rewritten to and is updated in the main memory only when the cache is evicted. In a **write-back** cache, however, data is updated in the cache and main memory every time the data block is updated (written to) [45].

2.5 Novelty

Based on literature, we build a compression and cache manager that reduces the overall memory bottleneck for extreme-scale applications. This thesis therefore attempts to answer the following research questions:

1. How does inline compression reduce the memory footprint inside large-scale HPC applications?

2. How can software caching be leveraged to reduce overhead incurred by compression operations and improve the performance of HPC applications?

3. How do we measure the storage improvement and performance improvement in the applications?

Chapter 3

Related Work

The two main pivotal ideas in this thesis are data compression and software caching to alleviate memory bottlenecks in applications. Here we discuss work in literature similar to our work in compression and software caching. This forms the basis to build our compression and cache managers and the performance model to assess them.

Previous work has explored the use of hardware compression to expand the size of hardware caches and main-memory [5, 37, 11, 33, 23]. In [43], they propose a decoupled compressed cache (DCC), which exploits the spatial locality of an application and improves the performance and energy-efficiency of cache compression. In this mechanism, the focus is on improving the effective cache capacity by compressing the cache blocks, therefore the cache misses reduce. Greater number of data blocks are accommodated, the performance improves and the energy of the system reduces. DCC increases maximum effective capacity to four times the uncompressed capacity. This is done at the hardware cache level for last-level caches and while it incurs area overheads, it proves how reducing the memory requirements helps improve the performance of applications and leads to optimum utilization of resources. Sardashti, Seznec and Wood in [41] build on this DCC and focus on lowering the overheads and improving the performance by utilizing skewed compressed caches (SCC). SCC compacts the blocks into a variable number of sub-blocks to reduce internal fragmentation and reduces the tag overhead. Sardashti, Seznec and Wood in [42] combine [43] and [41] and build a simpler and practical design to utilize the benefits of DCC and SCC.

All these works focus on compression in hardware caches. The idea of software caches to improve I/O performance has been explored in [38, 50, 44]. [31] looks at utilizing data compression and caching in communication networks. In order to minimize the latency in networks with large-scale data generated at edge nodes of routing paths, data compression and caching decisions are incorporated. They look at optimizing data compression ratios (maximizing the ratio) and cache gains (maximum hit ratio) under an energy consumption constraint. They propose an algorithm that optimizes the parameters for compression and caching to maximize the compression ratio and hit ratio. The algorithm achieved near-optimal performance for a tree-structured network and easily extends to a general network topology. While their work focuses on communication networks, incorporating compression and caching minimized the latency in the routing path of the network and proves to be beneficial in improving overall performance and optimal resource usage of applications.

Islam et al. and Bicer et al. in [22, 6] respectively explore software caching for distributed applications. Caching input data for parallel tasks is explored in [52, 4]. Gok, Capello et al. in [19] look at compressing and reducing the storage requirements in a quantum chemistry application which has a space complexity of $O(N^4)$. Due to the calculations leading to massive memory requirements, they leverage lossy compression to compress the integral's data and store it on disk to avoid recalculation and further computation. Since this application requires fast and effective compression, they develop an algorithm, Pattern Scaling for Two-electron Repulsion Integrals (PaSTRI) inside the data compression package. This compression package is utilized by the application, which analyzes the integral dataset and the patterns of the application and performs compression using the algorithm for achieving maximum compression ratio. The algorithm leverages the latent pattern features in the integral dataset and optimizes the calculation of the appropriate number of bits required for the storage of the integral. The algorithm gives a good compression ratio (16.8) while incurring minimal overhead. Our work uses a compression manager and embeds it inside the application, similar to PaSTRI. However, we utilize a software cache as well to have faster access to arrays and further reduce the overhead incurred during compression. The benefits of caching as explained in works above enable the embedded compression manager to improve the memory requirements and performance of the application.

Chapter 4

Research Methodology of the Compression and Cache Manager

This chapter focuses on the methodology and the working principle of incorporating compression in this thesis to build the compression manager and cache manager. We first describe the compression manager and the software cache manager built for reducing memory bottlenecks. We delve into the design and the properties for the two managers. For the compression manager, we describe the specific libraries and compressors utilized to perform compression. We further describe the design choices made with respect to selection of compression methodology, caching methodology and cache policy for replacement of a cache block and writing to the cache.

4.1 Compression Methodology

This section describes the compression methodology and tools incorporated to perform data compression for memory intensive applications.

4.1.1 Inline Compression

High Performance Computing (HPC) applications and scientific applications run into large scale memory requirements and memory bottlenecks. Ad-hoc compression while effective in reducing the physical storage requirements of the applications still runs into memory bottlenecks while the application is running on the system. Based on the scale and size of the application, an upgrade of physical storage is required to prevent running out of memory. Inline compression on the other hand enables improving the effective storage and memory capacity as it compresses and decompresses data accordingly while the application is running. Therefore, inline compression enables the application to reduce its memory bottleneck and improve memory performance. We utilize inline compression for the working set in main memory. We perform it in main memory specifically as the memory performance improves and there is no dependency on secondary storage while running the application and writing the data. Therefore, all compressed data blocks exist in their compressed state in main memory and to read the data block, only the decompressed values of the block are utilized while the data still exists in its compressed state in main memory.

4.2 The Compression Manager

This section describes the compression manager built to utilize compression in an application. It describes the design choices and the compression libraries it utilizes to perform compression inside an application.

4.2.1 Design of the Compression Manager

The memory intensive HPC applications need compression to alleviate the memory requirements. There are different compression libraries available which enable an application to select compressors and incorporate compression inside the application. Therefore, the application needs a pathway to be able to call upon the compression library's functions and perform compression. Our compression manager is thus a wrapper that wraps around the compression library by encapsulating its functionalities within. It acts as an interface between the memory-intensive application and the compressor (in a compression library) we utilize to reduce the memory footprint. In our experiments, we utilize libpressio as the compression library as it provides us with a variety of compression methodologies to choose from and abstracts all internal workings while providing the flexibility to tune the parameters and metrics for performance assessment. The main focus of our compression manager is to maintain the accuracy of the application and therefore we utilize lossless compression. libpressio has the functionality to call upon Blosc, a lossless compression library and we specifically utilize Blosc's Zstandard (Zstd) algorithm to perform lossless compression. The interface is shown in Figure 4.1.

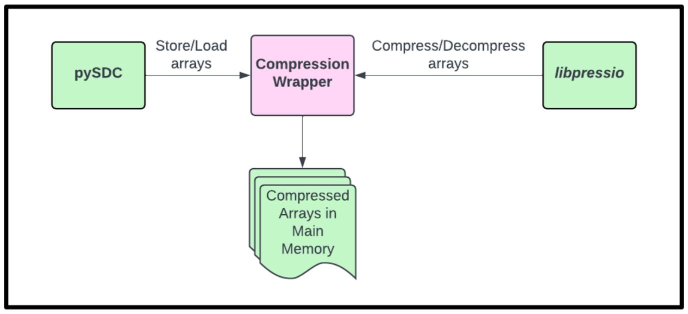


Figure 4.1: Compression Manager as the Interface

The compression manager's goals and design is displayed in Figure 4.2. The main objective of this manager is to be able to compress arrays when storing them and decompress them with minimum overhead when required for computation and operations within an application. Therefore, from an application's perspective the manager must know the array name, its data type, size and the operation being performed on the array (read from or written to). These parameters are therefore input parameters for the manager. It also takes input parameters for compression such as the type of compressor. We do not need to know the parameters for zstd as we leverage libpressio which handles the internal workings and parameters for zstd. For the software implementation, the compression manager being a wrapper for multitude of compressors within the compression library needs to be a template such that any compressor and its respective configurations are selected while also allowing the flexibility to tune parameters for optimal compression metrics. Therefore, the compression manager is implemented as a class *manager* which has the input parameters as attributes as displayed in Figure 4.2. As we incorporate a software cache to assist the compression manager's operations and reduce its overhead, the *manager* has a cache attribute as well which is a $cacheManager$ class and has its attributes pertaining to the cache. These cache parameters are tuned through this class which impact the compression manager's overhead and optimize the compression metrics further.

4.3 The Cache Manager

This section describes the design and workflow of the cache manager. The cache manager as displayed in 4.3 aids the compression manager by providing access to the decompressed values of the

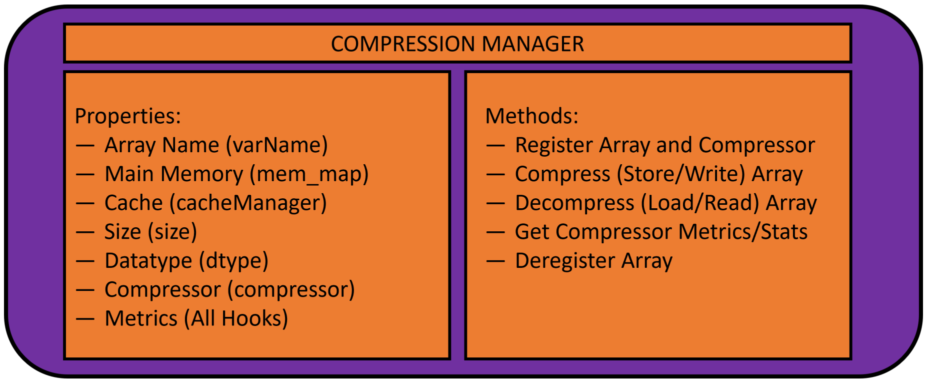


Figure 4.2: Inline Compression Manager

array for loading or updating the array for storing. This reduces the number of calls for compression and decompression thereby reducing the overhead.

4.3.1 Design of the Software Cache

The software cache stores decompressed values of the arrays and enables limited number of compression and decompression calls to libpressio and consequently the operations. It is implemented in software as a class $cacheManager$ and is an attribute of the compression manager's class $manager$. This class implementation enables it to behave as a template and test different flavors of the cache by tuning the parameters and analyzing its impact on the performance of the cache and the compression manager. The cache manager needs the array name and decompressed values of the array as an input. This is because the software cache only stores the decompressed values of the array and only requires the array name and its values. Therefore, the compression manager provides it the required inputs when it initializes the cache manager as an attribute and has to insert arrays into the cache. The $cacheManager$ has the software cache $cache$ which is a key-value pair dictionary containing all arrays $varName$ as key and the decompressed values of the respective arrays as values. The cache size impacts the number of decompressed arrays the cache can hold at a time. Therefore the $cacheManager$ has $capacity$ as an attribute which determines the number of arrays the $cache$ can store. As the cache size $capacity$ increases, the number of arrays stored increases, leading to an increase in cache hits and fewer decompression calls. With a smaller cache, the number of cache invalidates are high which reduce on increasing the cache size. To implement the cache policy described in section 4.3.2, the $cacheManager$ has dictionaries $cacheFrequency$ and $countCache$

which enable it to implement the cache write and cache eviction policy.

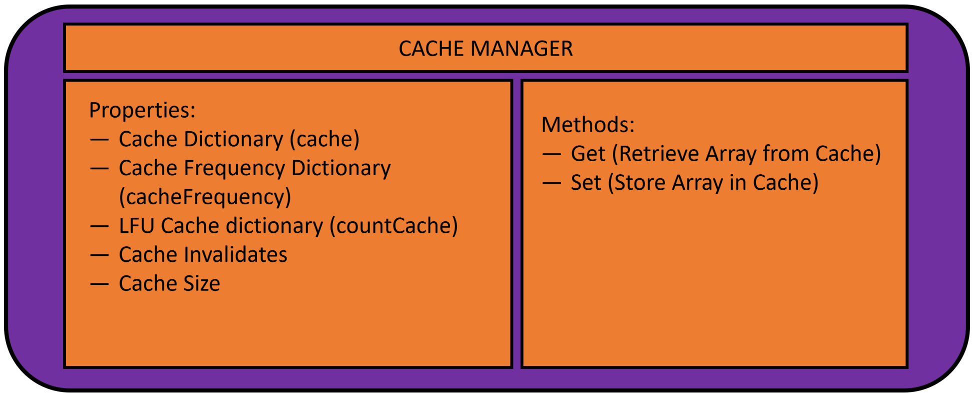

CACHE MANAGER
Properties:
— Cache Dictionary (cache)
— Cache Frequency Dictionary (cacheFrequency)
— LFU Cache dictionary (countCache)
— Cache Invalidates
— Cache Size
Methods:
— Get (Retrieve Array from Cache)
— Set (Store Array in Cache)

Figure 4.3: Cache Manager

4.3.2 Cache Policy and Implementation

Once an array is decompressed, it has to be stored into the cache. Further, if it is to be written to or updated, we have to update the cache block where the array is stored. We use a write back cache since it has low latency [45] and it is written only at one place - the cache and the I/O completion is confirmed. All large-scale applications have mixed workloads of reads and writes. Therefore, a write-back cache is the optimal and best-performing cache write policy as read and write I/O have similar response times [45].

Once the cache has reached its capacity and another array has to be inserted into the cache, a cache block is evicted to be able to insert the new array into the cache. For example, if the cache size (capcity) is 2 and two arrays A and B are to be added. A and B are first initialized and added to the cache. Once the summation operation is performed and the result is stored in a third array C, to insert C into the cache requires the eviction of one of the arrays already present in the cache as the cache is full. This requires the need for a cache eviction policy which selects the block to evict and inserts the new array block in its place. Least Frequently Used (LFU) eviction policy is implemented for its maximal hit ratio for spatial locality applications and ease of implementation. Here $cacheFrequency$ for the class $cacheManager$ stores the array and its frequency counter as a key-value pair respectively leading to an $O(1)$ access time. Another dictionary *countCache* stores the frequency as the key and all array names with that frequency as a list of values. This enables us to choose the array to evict based on recency of use of the array, when a conflict arises with multiple arrays having the same frequency counter. Based on the list, the array least recently used is the head of the list corresponding to the frequency, and this array will be evicted. Having these two dictionaries to implement the LFU cache eviction policy ensures the O(1) access time [32]. Consequently, we ensure that the eviction policy contributes the least possible overhead in terms of time.

4.4 Implementation of Compression and Cache Manager

This section describes how the compression and cache manager work cohesively inside an application to enable it to reduce its memory requirements while improving the memory performance of the application.

4.4.1 Lifecycle of a Compressed Array

A compressed array represents the array of the application in compressed form, and throughout the execution of the application is written to or read from. Therefore, to be able to perform these operations, application arrays undergo a change from their uncompressed state to compressed state to decompressed state to deletion. This entire lifecycle is described below:

- **Register array in Main Memory**: When an array is to be first written to, it is initialized as a compressed array and registered in main memory *mem_map* of the compression manager. The variable's name $varName$, compressor *compressor*, data type *dtype*, size *size* is added to the compression manager's (*manager*) main memory attribute *mem_map*. Within the registration function, the compressor is initialized as well as the variable by calling libpressio's initialize and set configuration function.

- **Write/Store array and Add to Cache**: When a variable is first registered and written to, it is compressed and added to main memory, while the variable's decompressed values are added to the cache *cache* through the $cacheManager$ attribute of the compression manager $manager$. Furthermore, after the registration for every subsequent write and store, $manager$ checks if the variable exists in cache, if it exists, it is updated. If the variable does not exist in cache, the array has to be first decompressed by calling libpressio's decompress function on the array in main memory, and then its decompressed values are added to the cache. The respective attributes and metrics for the cache manager such as cache dictionary, cache frequency dictionary for the frequency counters and a cache history log is updated. Consequently, in the compression manager, the metrics are updated by updating their respective attributes.

- **Update array in Cache**: While writing to a variable, if it exists in cache, it is an update and therefore no compression is required since this is a write-back cache. Therefore, the array's decompressed values are written to the array in the cache while the array maintains its unedited compressed state in main memory. The update time metric is updated and the cache manager's attributes and metrics are updated.

- **Read/Load array**: When an array is to be read or loaded, the manager first checks if it exists in the cache. If the array exists in the cache, it is retrieved. If the array does not exist in the cache, then it is first decompressed from main memory by calling libpressio and then added to the cache. Once added to the cache, the array is retrieved. All the metrics for the compression manager such as the number of decompression calls if decompressed, decompression time, retrieval time are updated. Similarly, the cache manager's attributes and metrics are updated.

- **Evict array from Cache**: If the array is to be added to cache (either to store or store and load) and the cache size has reached its maximum capacity, the array with the least frequency is first evicted and then the new array to be added is added to the cache. All the frequency counters for the cache and its metrics are updated, while also updating the metrics and attributes in the compression manager. Moreover, the evicted array's values are now updated in main memory and then compressed to ensure the array is up-to-date in accordance with the write-back cache mechanism. The number of cache invalidates are updated.

- **Deregister array from Main Memory**: Once the array variable is deallocated, the variable is deleted from cache if a cache exists. If it already exists in cache, it is deleted from main memory to finally deregister it as a compressed array.

4.5 Metrics and Performance Modeling

The metrics help assess and quantify the impact of incorporating compression and caching inside pySDC. They further provide insights on how the run-time and storage requirements of the

Algorithm 1 Algorithm for Compressed Mesh Arrays Lifecycle

```
Require: varName, compressor, dtype, size, varOperation
  Run application
  Initialize manager
  Initialize cacheManager
  Get varOperation
  if varOperation is Register then
      Add varName, compressor, dtype, size, stage to mem_map dictionary in manager
      Update registered variables and active registered variables in the hook‘
  else if varOperation is Write/Store varName then
      if varName is in cache then
          Update varName array’s decompressed values to the new array and update cacheFrequency, countCache
      else
          if cacheSize is less than capacity then
              Set varName, compressor, call libpressio to compress and add it to mem_map
              Add uncompressed varName array to cache and update cacheFrequency, countCache, cacheInvalidates
          else
              Evict array evictArray with the least frequency and least recency in cacheFrequency, countCache
              Update cacheFrequency, countCache and evictArray in mem_map and compress it
              Add varName array to cache and update cacheFrequency, countCache, cacheInvalidates
          end if
      end if
  else if varOperation is Read/Load then
      if varName is in cache then
          Update varName in cacheFrequency, countCache
          Return varName array
      else
          Get varName from mem_map and call libpressio to decompress varName array
          if cacheSize is less than capacity then
              Add varName array to cache and update cacheFrequency, countCache, cacheInvalidates
              Return varName array
          else
              Evict array evictArray with the least frequency and least recency in cacheFrequency, countCache
              Update cacheFrequency, countCache and evictArray in mem_map and compress it
              Add varName array to cache and update cacheFrequency, countCache, cacheInvalidates
              Return varName array
          end if
      end if
  else if varOperation is Deregister then
      Delete varName from cache, cacheFrequency, countCache
      Delete varName from mem_map
      Update number of active registered variables
  end if
```

applications are affected while incurring their respective overheads. We first delve into the metrics and then describe how these can be used to build performance models for the managers.

4.5.1 Metrics for Evaluation

The primary metrics for the compression manager and cache manager are described below.

1. **Number of Compression and Decompression Calls**: is the number of calls to libpressio to perform compression or decompression, respectively. By building our compression manager and incorporating our software cache, we have faster access to decompressed values, which should reduce the number of calls to libpressio.

2. **Time taken to Compress/Decompress**: The total time taken for the compression manager to perform compression or decompression by calling upon libpressio's encoder (compression) or decoder (decompression) function.

3. **Number of Cache Invalidates and Evictions**: Cache Invalidation is the process of deeming an array block in the cache as invalid and removing it, leading to cache eviction or updating it. Number of cache invalidates is the frequency counter of the number of times an array is deemed invalid and evicted. The number of cache invalidates equal the number of evictions in our cache manager, as we remove the array every time it is deemed invalid. We only deem an array invalid when the cache size is at its maximum capacity and an array has to be stored, so one of the array will have to be evicted.

4. **Compression Ratio**: is defined as the ratio of the size of uncompressed data in main memory in bytes to the sum total of the size of compressed data in main memory and size of decompressed data in the cache. Equation 4.1 displays the Compression Ratio where CR is the compression Ratio, $n_{UncompMem}$ is the size of uncompressed data in main memory, $n_{CompMem}$ is the size of compressed data in main memory and n_{Cache} is the size of decompressed data in cache.

$$CR = \frac{n_{UncompMem}}{n_{CompMem} + n_{Cache}} \tag{4.1}$$

4.5.2 Performance Model

The performance of the compression manager and cache manager are assessed through the metrics described above. We use these metrics to design a performance model which helps understand and quantify the impact of compression and caching on the memory-intensive applications. This performance model will be used in chapter 6 to display the overall model for the compression and cache manager and its impact on improving memory performance and storage of the application.

1. **Compression Ratio with Cache Size**: Equation 4.1 shows how compression ratio is calculated for the compression manager. It is dependent not just on the uncompressed and compressed size of arrays in main memory but also on the decompressed size of arrays in cache. Therefore, as the cache size increases, the compression ratio will reduce as the increase in cache size will add to the larger size of decompressed arrays in cache, thereby reducing the overall compression ratio.

2. **Compression/Decompression Time**: The total amount of time incurred while performing compression and decompression is an important overhead. On utilizing a cache since the number of compression and decompression calls change and the cache eviction overhead comes into the picture as well, we add the overhead time due to the cache as well in the compression and decompression times. Therefore, this model enables us to use the metrics above and know the total overhead in incorporating compression. Moreover, it helps know the optimal parameters for the compression manager as well as the cache manager for the applications. We model the total time for compression and total time for decompression as follows:

$$T_C = T_{CompMainMem} + T_{CacheUpdate} \tag{4.2}$$

   Equation 4.2 where T_C is the total time for compression and comprises the time taken to compress the array and add to main memory $T_{CompMainMem}$ and the time taken to just update (for all the writes/stores) the decompressed values of the array in cache if a cache exists and the array exists in cache $T_{CacheUpdate}$. $T_{CompMainMem}$ is the time for compression when a cache does not exist or if it does, the array is not present in the cache. This time can be further sub-divided into time taken to call libpressio to compress the array and add to main memory T_{Comp} and if a cache exists, time taken to put the uncompressed values of the array in the cache T_{Set} and if the cache is full, time taken to evict and put the uncompressed array in the cache $T_{CacheEvict}$.

$$T_{CompMainMem} = T_{Comp} + T_{Set} + T_{CacheEvict} \tag{4.3}$$

If the array is found in the cache, the time taken for compression only involves updating the uncompressed array and is therefore calculated with time $T_{CacheUpdate}$ Thus, the total time for compression can be calculated by adding all the times.

$$T_C = T_{Comp} + T_{Set} + T_{CacheEvict} + T_{CacheUpdate} \tag{4.4}$$

For total decompression time, equation 4.5 calculates it by adding the time taken to decompress the compressed array in main memory $T_{DecompMainMem}$ and the time taken to retrieve it from the cache for all the loads $T_{CacheGet}$.

$$T_D = T_{DecompMainMem} + T_{CacheGet} \tag{4.5}$$

$T_{DecompMainMem}$ can be further subdivided into time taken to call libpressio's decompress function and decompress the array in main memory T_{Decomp}, then add these decompressed values to the cache if a cache exists and return the decompressed array T_{Set} and time taken for cache evictions and retrieving the required array $T_{CacheEvict}$

$$T_{DecompMainMem} = T_{Decomp} + T_{Set} + T_{CacheEvict} \tag{4.6}$$

If the array is found in the cache, the time taken to retrieve the array is $T_{CacheGet}$. The total time can be modeled as follows for the total time in performing compression and decompression on the compressed mesh arrays.

$$T_D = T_{Decomp} + T_{Set} + T_{CacheEvict} + T_{CacheGet} \tag{4.7}$$

In chapter 6, based on the cache configurations the metrics - number of compression/decompression calls, number of cache hits and number of cache invalidates will change and therefore the respective timings will change. Thus the performance model will help visualize and quanitfy the overhead for each configuration and choose the optimal parameters for the applications.

$$T_{tot} = nCmp.T_C + nDcmp.T_D \tag{4.8}$$

Chapter 5

Testing Methodology for the Compression and Cache Manager

This chapter describes the applications and framework utilized for testing our compression manager and cache manager. We describe the pySDC [47] framework utilized to test our manager and the specific memory intensive applications inside pySDC used for testing. We then describe how the manager is embedded inside the pySDC framework for reducing bottlenecks in memory and the entire workflow of the manager and the applications. Finally, the performance modeling and metrics to assess the performance of the compression and cache managers is described. We test reducing the memory footprint inside pySDC as it runs into extreme-scale memory requirements.

5.1 pySDC Framework

pySDC solves the differential problem $Au = f$ using the iterative, parallel-in-time approach of Spectral Deferred Correction (SDC) and collocation. Therefore, for each of the M collocation nodes of the n-th dimensional system, all state variables in u have to be stored within a time-step for the next iteration. We compress u at all parallel-in-time stages in order to reduce per node memory requirements, to avoid running out of memory.

5.1.1 Compressed Mesh datatype

A mesh class is a class within pySDC that performs all the arithmetic operations on the SDC application's meshes. Meshes are the unit of computation in Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs). We create a class called Compressed Mesh which inherits from the mesh class and is used by the compression manager to initialize arrays, perform compression functionality and arithmetic operations.

The compressed mesh objects perform the same functionalities as a mesh object but on compressed arrays. We create this class to create a datatype that enables us to perform arithmetic operations on compressed arrays. The compressed mesh datatype overrides mesh arrays functionality for arithmetic operations. The compressed mesh arrays are derived mesh arrays. The compression manager is an attribute of a compressed mesh class. A compressed mesh object therefore has access to the compression manager and cache manager's functionality to compress when writing to an array and decompress and retrieve the array when reading from an array. It further uses its own addition, subtraction, and multiplication functions to perform these operations for the application's arrays.

5.1.2 Hooks

Since pySDC applications are run in different stages and time steps, logging the metrics at each time step while the application is running helps evaluate the metrics and performance model. Hook classes are created for each of the metrics and at the end of each time step, the respective functions for the metrics are called for evaluating the metrics. Figure 5.1 shows the different hook classes created and at the end of each time step, the *add_to_stats* function for each of the classes logs the metric and is used at the end of the application run to utilize the metrics for assessment and building the performance model.

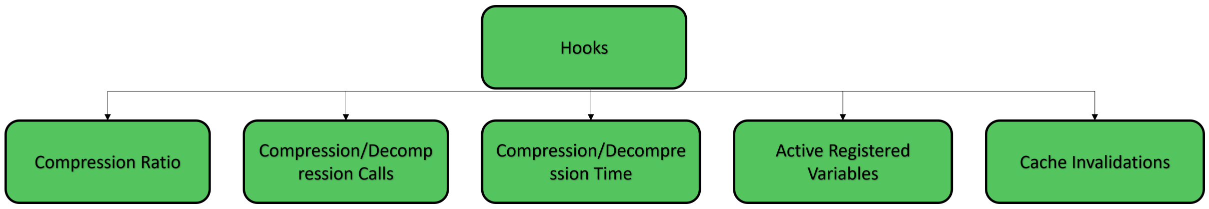

Hooks
Compression Ratio
Compression/Decompression Calls
Compression/Decompression Time
Active Registered Variables
Cache Invalidations

Figure 5.1: Hook Classes for Logging Metrics

5.2 pySDC Applications

For developing the performance model and exploring the performance impact of the compression manager and cache manager, we run pySDC for two applications:

1. Allen-Cahn equation: This application is a 2D reaction-diffusion equation of mathematical physics. The application parameters run for the experiment are shown in Table 5.1a.

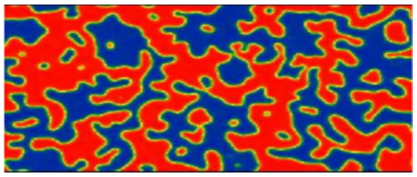


Figure 5.2: Allen-Cahn Simulation of Diffusion [36]

2. Heat Diffusion: is a physics application simulating heat traveling through a region. It is a partial differential equation and the application parameters run are as shown in Table 5.1b.

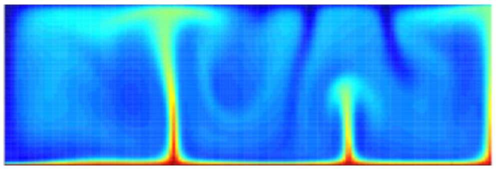


Figure 5.3: Heat-Diffusion Application Simulation [55]

Parameter	Value
Epsilon	0.4
Radius	0.25
Spectral	No
Problem Size	128×128
Residual Tolerance	$5e-07$
Maximum iterations	50
Number of Sweeps	1
Node Type	Legendre
Quad Type	Radau-Right
Collocation Nodes (M)	3

(a) Allen-Cahn Parameters

Parameter	Value
Nu	1.0
Order	4
Solver type	CG
Problem Size	$64 \times 64 \times 64$
Residual Tolerance	$1e-04$
Maximum iterations	50
Number of Sweeps	1
Node Type	Legendre
Quad Type	Radau-Right
Collocation Nodes (M)	3

(b) Heat-Diffusion Parameters

Table 5.1: Application Parameters for pySDC

These applications are chosen as these are simple ordinary differential equations (ODEs) and partial differential equations (PDEs) and their convergence is easy to simulate. These applications are great for testing with, being multidimensional enables to test the memory reduction performance of our compression manager and cache manager.

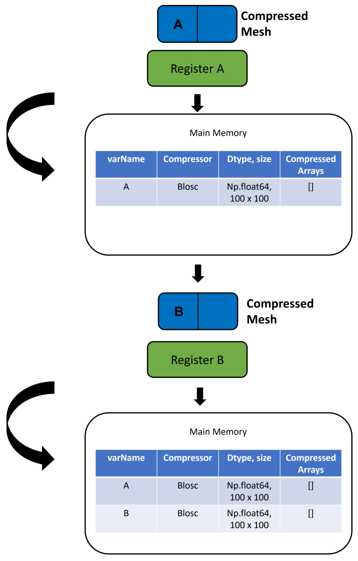
Experiments are run on Clemson's Palmetto Cluster, and we test direct-mapped caches for the cache manager with LFU eviction policy. The compression manager and cache manager's parameters are displayed in Table 5.2. The cache size ranges from 1 to 16 in powers of 2 and the baseline is no cache being utilized. The performance metrics and the performance models are compared with this baseline.

Parameter	Value
Compressor	Blosc - Zstd
Compression Type	Lossless
Cache Size	1,2,4,8,16
Associativity	Direct-Mapped
Cache Replacement Policy	LFU
Cache Writing Policy	Write-back

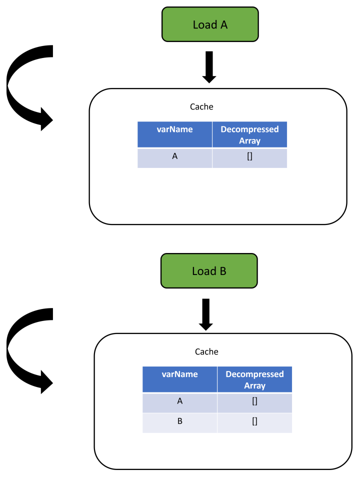
Table 5.2: Compression and Cache Manager's Parameters

5.3 Workflow of the Compression and Cache Manager for Compressed Meshes

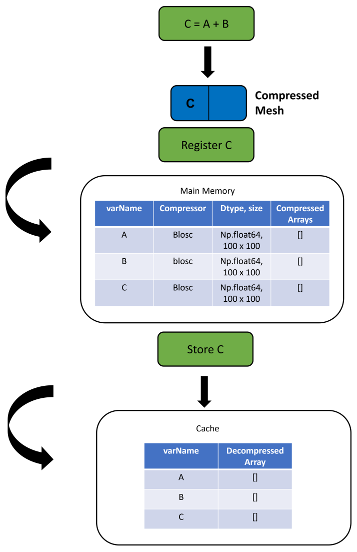
The workflow of the entire compression manager and cache manager can be explained with the help of an example of two compressed mesh arrays undergoing vector addition, as shown in figure 5.4. Initially, the two arrays of float datatype and size 100×100 are initialized as compressed arrays and registered in main memory. As they are registered and written to, in the register operation, the manager calls libpressio's encode function to compress the two meshes and store them in main memory. To perform the addition, the compressed arrays have to be first read from. Based on the lifecycle for the load operation the manager checks if the array exists in cache, since this is the first time these arrays are being read, the arrays are not found in the cache. Therefore, they have to be first retrieved from main memory and decompressed, for which the manager calls libpressio's decode function to decompress. During the manager's load operation and reading arrays, since the arrays were not in the cache, it adds the decompressed values to the cache and retrieves these for the vector addition. Once the vector addition is performed, a new array is initialized as a compressed array and registered in main memory. The manager performs compression on the array and adds


A
Compressed Mesh
Register A
Main Memory
varName | Compressor | Dtype, size | Compressed Arrays
A | Blosc | Np.float64, 100 x 100 | []
B
Compressed Mesh
Register B
Main Memory
varName | Compressor | Dtype, size | Compressed Arrays
A | Blosc | Np.float64, 100 x 100 | []
B | Blosc | Np.float64, 100 x 100 | []

(a) Registering and Storing Stage of Compressed Meshes



(b) Loading of Compressed Meshes and Performing Computation


C = A + B
C
Compressed Mesh
Register C
Main Memory
varName | Compressor | Dtype, size | Compressed Arrays
A | Blosc | Np.float64, 100 x 100 | []
B | blosc | Np.float64, 100 x 100 | []
C | Blosc | Np.float64, 100 x 100 | []
Store C
Cache
varName | Decompressed Array
A | []
B | []
C | []

(c) Final Stage of all Compressed Meshes

Figure 5.4: Workflow of the Manager – Compressed Mesh example

the compressed array to main memory. Therefore, at the end of this vector addition operation, the three compressed arrays exist in main memory in compressed state. The decompressed values of the two arrays added are in the cache, and the corresponding metrics are updated based on the state of the array.

At the end of this application, the arrays are deleted (removed) from the cache and main memory to deregister the arrays. An important point to note here is that if no cache is used, the arrays are compressed and decompressed as per the operation from main memory itself. Furthermore, the cache configuration adds overhead in terms of cache evictions and write-back mechanism. If the cache size is 1 and the cache is full, array A will first be evicted before B is added to the cache. In doing so, A's values will be updated and compressed in main memory, and this incurs considerable overhead.

Chapter 6

Experimental Results and Discussion

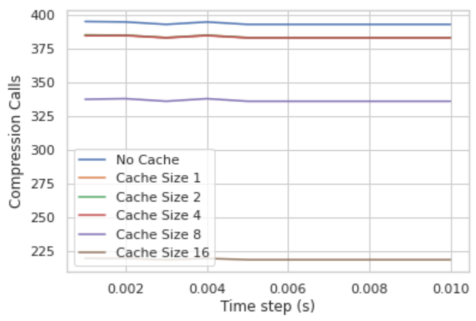
This chapter describes the results and performance modeling for the compression manager and cache manager. It further delves into explaining the performance implications of the metrics and the model.

6.1 Evaluation of Performance Metrics

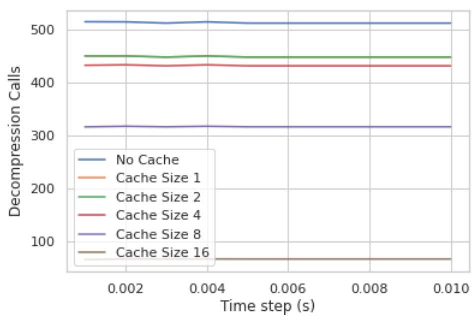
We evaluate the two scientific applications - Allen-Cahn equation and Heat-diffusion equation by running it on the Palmetto Cluster and analyze the results using data frames. The different metrics assessed for the compression manager and cache manager are described below.

6.1.1 Number of Compression/Decompression Calls Metric

The metric number of compression and decompression calls helps understand the overhead incurred by incorporating compression through our compression manager. The number of compression calls are a considerable factor, as every compression call requires the compression manager to call upon libpressio and the specific compressor (Zstandard (Zstd) in our experiments). Therefore, the number of compression and decompression calls help assess the performance of the compression manager by showing how much overhead is incurred inside the application.

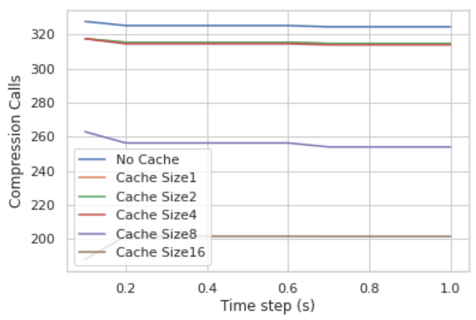


(a) Compression Calls

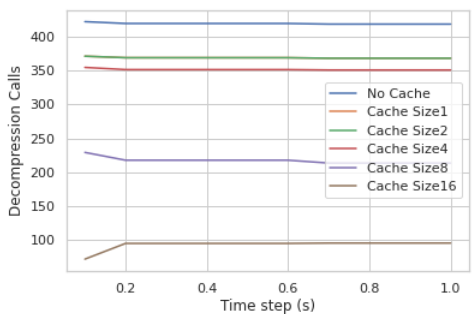


(b) Decompression Calls

Figure 6.1: Number of Compression and Decompression Calls for Allen-Cahn



(a) Compression Calls



(b) Decompression Calls

Figure 6.2: Number of Compression and Decompression Calls for Heat-Diffusion

Figures 6.1 and 6.2 show the number of compression and decompression calls for Allen-Cahn (AC) and Heat-diffusion, respectively. As the cache size increases, the number of compression and decompression calls reduce. This is because with an increase in cache size, the number of cache hits increase and the decompressed arrays can be retrieved for loading the array or updated for storing the array. Therefore, there is no need to call libpressio's functions to compress or decompress for updating or retrieving the array, respectively. The reduction in number of calls helps reduce the overhead incurred by the application in incorporating compression.

A hook class is created for logging the number of active registered arrays at each time step to understand the number of variables in main memory and how the cache manages all the arrays as described in section 5.1.2. Table 6.1 shows the number of active registered variables for both the applications for all the time steps. In both the applications, the first time step has 14 active registered arrays and all the following time steps have 15 active registered arrays. At every time step

Time Step	Allen-Cahn	Heat-Diffusion
1	14	14
2-10	15	15

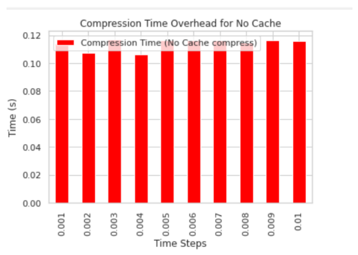
Table 6.1: Number of Active Registered Variables for all Time Steps

14 arrays are active, however in the first time step one array is registered and active throughout the entire run, it is never deleted from main memory. Therefore, it resides throughout and 15 actively registered variables exist.

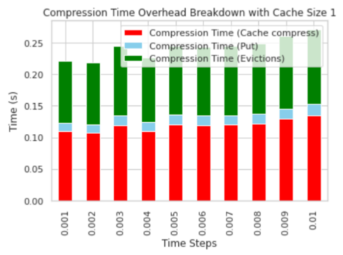
When the applications are run with the baseline which is the no cache manager, all 14 or 15 arrays have to be compressed and decompressed for every store and load respectively, leading to the maximum number of compression and decompression calls. On incorporating small cache sizes of 1, 2 and 4, we see a decrease in number of compression and decompression calls. This is because the arrays that exist in cache can be easily retrieved and updated and do not require corresponding decompression and compression calls. However, these small cache sizes have very similar number of compression calls and decompression calls and this alludes to the fact that the number of active registered variables are significantly larger than the cache size, leading to large number of cache evictions. As the cache size increases to 8, a significant reduction is observed, while with a cache size of 16 the minimum number of compression and decompression calls is observed as the entire working set fits in the cache. Once the entire working set fits in the cache, there are no more calls required to libpressio as no cache evictions occur and all the stores and loads happen directly in the cache itself.

6.1.2 Compression/Decompression Time Metric

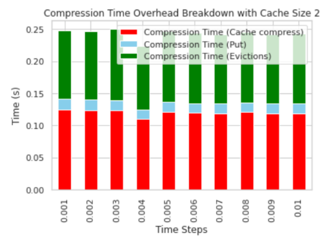
On incorporating the cache, we see the number of compression calls and decompression calls decrease, thereby reducing the overhead of the compression manager and libpressio. This should therefore translate into reducing the time taken to compress and decompress arrays. However, the compression and decompression time does not reduce for small cache sizes and incurs a significant overhead as displayed in figures 6.3, 6.4, 6.5 and 6.6 for both the applications. On increasing the cache size to 8 and 16, the overhead barrier is broken and the compression and decompression time are less than the baseline test. This is because small cache sizes lead to large number of cache evictions, and this is explained below with the help of the number of cache invalidates metric. This metric was logged through another hook class, calculated based on the number of cache evictions



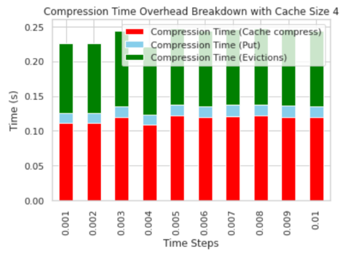
(a) No Cache



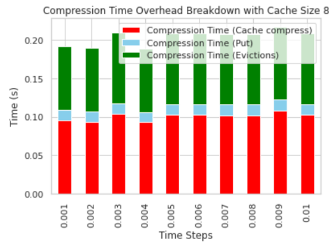
(b) Cache Size 1



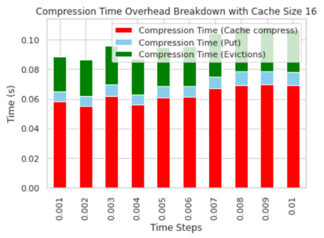
(c) Cache Size 2



(d) Cache Size 4



(e) Cache Size 8



(f) Cache Size 16

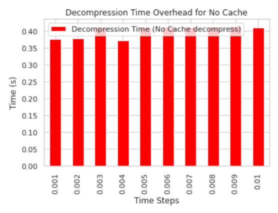
Figure 6.3: Allen-Cahn Application Compression Times

and Least Frequently Used (LFU) policy when the cache size was at its maximum. The compression time corresponds to the total compression time, $nCmp \times T_C$ and the decompression time corresponds to the total decompression time, $nDcmp \times T_D$ described in the performance model in chapter 4. This enables us to discuss the performance implications in section 6.2.

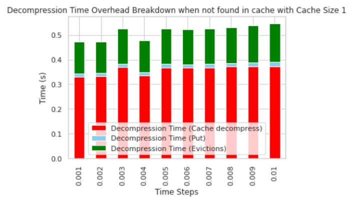
6.1.2.1 Number of Cache Invalidates Metric

Figures 6.7a and 6.7b display the number of cache invalidates for both the applications. For the baseline test, there are zero cache invalidates as there is no cache. For the same cache size, at different time steps we see an increase in cache invalidates and this is because for certain time steps, the application took higher number of iterations in that time step to converge and thus the number of cache invalidations increased. On incorporating small cache sizes of 1, 2 and 4, the cache invalidates are considerably high as the number of cache evictions are high. These high cache evictions can be attributed to the working set having comparatively higher number of arrays to be written to or read from, and using the LFU cache eviction policy leads to the least counter array data being invalidated and evicted once the cache is full.

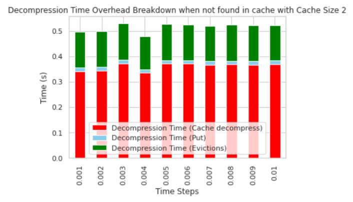
As the cache size increases to 8 and 16, the number of cache invalidates reduce as more arrays fit into the cache and cache evictions are less. When the cache size is 16, after all the arrays



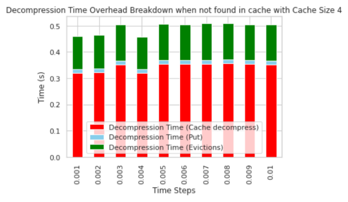
(a) No Cache



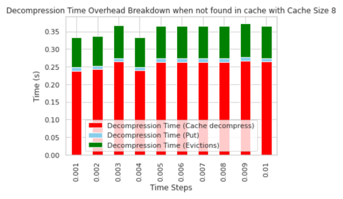
(b) Cache Size 1



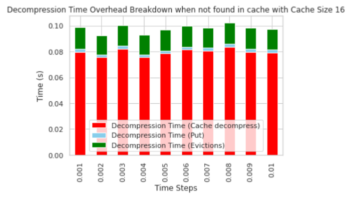
(c) Cache Size 2



(d) Cache Size 4

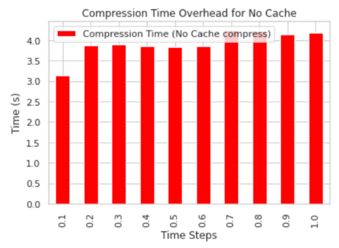


(e) Cache Size 8

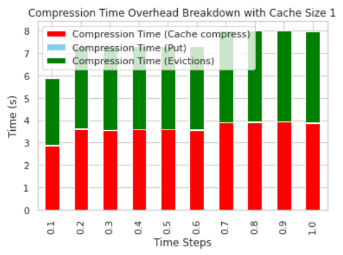


(f) Cache Size 16

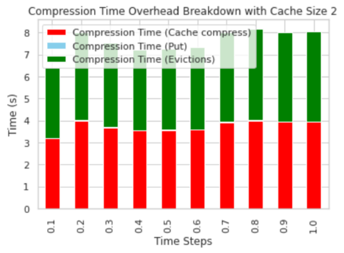
Figure 6.4: Allen-Cahn Application Decompression Times



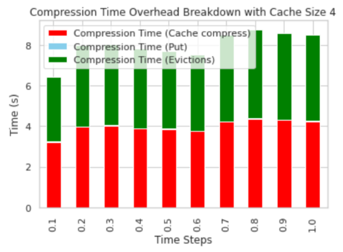
(a) No Cache



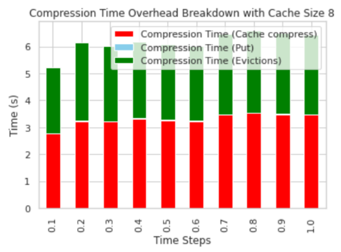
(b) Cache Size 1



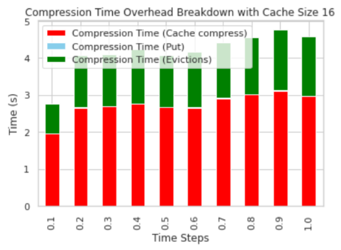
(c) Cache Size 2



(d) Cache Size 4

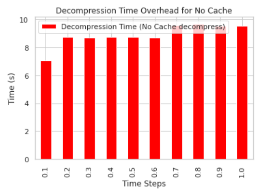


(e) Cache Size 8

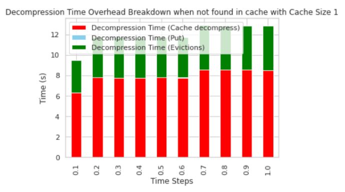


(f) Cache Size 16

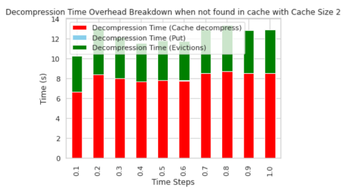
Figure 6.5: Heat Diffusion Application Compression Times



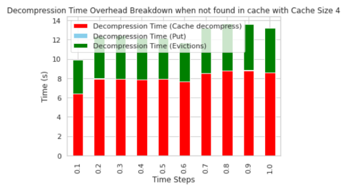
(a) No Cache



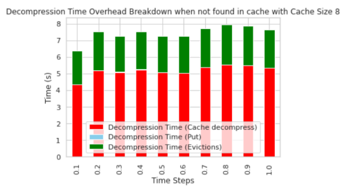
(b) Cache Size 1



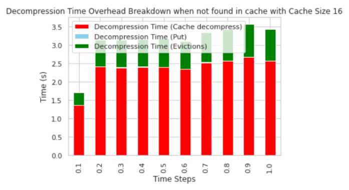
(c) Cache Size 2



(d) Cache Size 4



(e) Cache Size 8



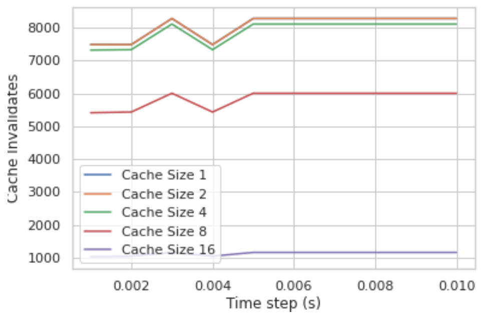
(f) Cache Size 16

Figure 6.6: Heat Diffusion Application Decompression Times

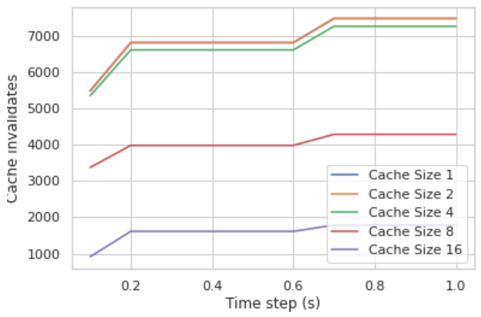
have been added to the cache, there are no evictions as the entire working set fits into the cache leading to no more evictions and cache invalidations.

Once an array is deemed invalid and evicted, it is updated and written to its most recent values in main memory as we use the write-back cache policy. However, every eviction garners writing back to main memory and calling the compression manager to compress the data and store it in a compressed mesh array. For small cache sizes, this incurs a significantly large overhead as the cache evictions are high and the arrays are constantly written back to and compressed. This number of cache invalidates lead to the corresponding number of write-backs in main memory and from figures 6.7a and 6.7b, this is significantly high. This is the main reason we see that the small cache sizes have higher times for compression and decompression than the baseline, despite the number of calls for compression and decompression being less.

Therefore, once the cache size is large to have a larger subset of working set arrays stored in the cache, the cache evictions reduce. The lesser number of cache invalidates lead to less write-backs to main memory and therefore reduce the overhead considerably due to eviction and compression for the write-back to main memory. Therefore, these cache sizes have lower times for compression and decompression and benefit the application performance by reducing the compression and decompression time.



(a) Allen-Cahn Application



(b) Heat Diffusion Application

Figure 6.7: Number of Cache Invalidates

6.1.3 Compression Ratio Metric

Table 6.2 displays the compression ratio achieved by the Zstd compressor for all the cache sizes for both the applications. The compressor is configured by libpressio and Blosc and our manager utilizes this configuration. The Compression Ratio is calculated as per equation 4.1. Therefore, increasing the cache size increases the number of decompressed arrays and the total size of decompressed arrays, eventually decreasing the compression ratio. We believe the reason for not seeing a compression ratio greater than 1 is the nature of the data (floating point) and the metadata. However, as the cache size increases, the decompressed values are readily available and the cost to compress and decompress by calling libpressio is reduced. However, the main point of incorporating compression into the applications is to reduce the memory footprint, and the compression ratio is an important indicator of the memory footprint reduced. The next steps involve working on improving the compression ratio by utilizing lossy compression techniques, which due to their nature of introducing error within a certain threshold help compress a greater amount of data and increase the compression ratio. This is discussed further in the next steps and conclusion in chapter 7.

Cache Size	Allen-Cahn CR	Heat-Diffusion CR
No Cache	0.9998	0.9999
1	0.9332	0.9333
2	0.8749	0.8749
4	0.7777	0.7777
8	0.6363	0.6666
16	0.5832	0.5599

Table 6.2: Compression Ratio for pySDC applications

6.2 Performance Implications

The metrics above help build the performance model as described in the methodology in chapter 4. The compression and decompression times are modeled based on the number of compression calls and decompression calls. These compression and decompression times are added for all the compression calls and decompression calls respectively, and the break-up of each of the times based on the performance model equations in section 4.5.2 are adhered too. One can see that the cache evictions are high for small cache sizes and therefore take up the major proportion of the time as well. As the cache size increases, the overall compression and decompression times reduce due to the reduction in cache invalidations and cache evictions. Therefore, a cache size encapsulating the entire working set helps the manager have minimum time overhead and further improves the memory performance of the application.

Chapter 7

Conclusions and Discussion

This chapter summarizes the objective and results obtained while paving the path for the next steps and future work.

7.1 Discussion of Results

High-Performance Computing (HPC) applications are memory intensive due to the extreme scale nature, and simulations required for these applications. An example is pySDC, a framework implementing spectral deferred correction (SDC) methods for ordinary differential equations and partial differential equations. These applications run into memory bottlenecks due to its iterative strategy using parallel-in-time methods, and with respect to the volume of data being stored and exchanged. Reducing the memory bottleneck enables the application to have optimal usage of memory resources and improves the memory performance of the application, which further improves the performance of the overall application. We therefore incorporate compression to reduce the memory bottleneck by building a compression manager. This compression manager abstracts all compression functionalities by utilizing compression library as libpressio and helps the application store data in compressed form. We further aid the compression manager using a cache manager that reduces the number of compression and decompression calls by keeping decompressed values of arrays for instant access. It minimizes the latency incurred by the compression manager to read and write to arrays, and therefore helps keep the time overhead to a minimum for the compression manager.

Incorporating our compression manager and cache manager enables the application being less memory intensive and improves the storage requirement as well as optimum usage of resources. The optimal parameters for the compression manager and cache manager are determined through our performance model and also vary from application to application based on the access patterns for the application. Overall, we see that a cache size of 16 or one that fits the entire working set of the application have minimum compression and decompression calls by leveraging minimum cache evictions. This considerably reduces the total time overhead as per the performance model. However, we can further improve the compression ratio by changing our compression parameters, as the cache does inflate the decompressed sizes in bytes. This is discussed in our next steps and future work below.

7.2 Future Work

For the purposes of this thesis and for preserving the accuracy of the data, we tested our compression manager with lossless compressors in our compression manager utilizing the library libpressio, specifically Blosc library for lossless compressors and the Zstd algorithm. While the results with respect to the number of compression calls, compression/decompression time and number of iterations the solution took to converge were preserved, the compression ratio and the overall application runtime were not optimal. To further improve the compression ratio and memory storage performance, introducing lossy compression is ideal. Lossy compression by introducing error and not preserving the accuracy of the data is able to compress greater amount of data, which helps get greater compression ratio and memory performance for an application.

Thus, our next steps involve introducing and running experiments with different lossy compressors such as SZ, ZFP and their configurations through libpressio. In introducing lossy compressors, parameters such as $errorBound$, $errorBoundMethod$ will be introduced and the configurations of the compressor will affect the compression metrics.

We would also look at partial inline compression for certain stages of the application in the background while the other stages are running in parallel, and therefore reduce the memory bottleneck with minimal overhead and further performance improvement of the application.

Bibliography

[1] Speck, Lunet, Baumann, Wimmer, Akramov. Compression and cache manager for pysdc. `https://github.com/Parallel-in-Time/pySDC/`. Accessed: 2023-07-31.

[2] Speck, Lunet, Baumann, Wimmer, Akramov. pysdc documentation v5.3.0. `https://parallel-in-time.org/pySDC//`. Accessed: 2023-07-31.

[3] Francesc Alted. Why modern cpus are starving and what can be done about it. *Computing in Science Engineering*, 12(2):68–71, 2010.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. {PACMan}: Coordinated memory caching for parallel jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, 2012.

[5] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 38–49, 2015.

[6] Tekin Bicer, Jian Yin, David Chiu, Gagan Agrawal, and Karen Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1205–1216, 2013.

[7] Blosc Development Team. Blosc: A blocking, shuffling and lossless compression library. `https://github.com/Blosc/c-blosc`. Accessed: 2023-07-06.

[8] Elizabeth L Bouzarth and Michael L Minion. A multirate time integrator for regularized stokeslets. *Journal of Computational Physics*, 229(11):4208–4224, 2010.

[9] Martin Braun and Martin Golubitsky. *Differential equations and their applications*, volume 2. Springer, 1983.

[10] Mathew Causley and David Seal. On the convergence of spectral deferred correction methods. *Communications in Applied Mathematics and Computational Science*, 14(1):33–64, 2019.

[11] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE transactions on very large scale integration (VLSI) systems*, 18(8):1196–1208, 2009.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Alok Dutt, Leslie Greengard, and Vladimir Rokhlin. Spectral deferred correction methods for ordinary differential equations. *BIT Numerical Mathematics*, 40:241–266, 2000.

[14] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[15] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[16] Matthew Emmett and Michael Minion. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science*, 7(1):105–132, 2012.

[17] Facebook. Zstandard - fast real-time compression algorithm. `http://facebook.github.io/zstd/#small-data/`. Accessed: 2023-07-31.

[18] Alyson Fox, James Diffenderfer, Jeffrey Hittinger, Geoffrey Sanders, and Peter Lindstrom. Stability analysis of inline zfp compression for floating-point data in iterative methods. *SIAM Journal on Scientific Computing*, 42(5):A2701–A2730, 2020.

[19] Ali Murat Gok, Sheng Di, Yuri Alexeev, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2018.

[20] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] Jingfang Huang, Jun Jia, and Michael Minion. Accelerating the convergence of spectral deferred correction methods. *Journal of Computational Physics*, 214(2):633–656, 2006.

[22] Nusrat Sharmin Islam, Xiaoyi Lu, Md. Wasi-ur Rahman, Raghunath Rajachandrasekar, and Dhabaleswar K. D K Panda. In-memory i/o and replication for hdfs with memcached: Early experiences. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 213–218, 2014.

[23] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E Haque, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[24] Jason Brownlee. Introduction to dimesionality reduction for machine learning. `https://machinelearningmastery.com/dimensionality-reduction-for-machine-learning/`. Accessed: 2023-07-31.

[25] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.

[26] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S Vazhkudai, and Misbah Mubarak. Evaluating burst buffer placement in hpc systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[27] Anita T Layton and Michael L Minion. Conservative multi-implicit spectral deferred correction methods for reacting gas dynamics. *Journal of Computational Physics*, 194(2):697–715, 2004.

[28] Craig A. Lee, Samuel D. Gasster, Antonio Plaza, Chein-I Chang, and Bormin Huang. Recent developments in high performance computing for remote sensing: A review. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3):508–527, 2011.

[29] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[30] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems.* SIAM, 2007.

[31] Jian Li, Faheem Zafari, Don Towsley, Kin K Leung, and Ananthram Swami. Joint data compression and caching: Approaching optimality with guarantees. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 229–240, 2018.

[32] Dhruv Matani, Ketan Shah, and Anirban Mitra. An o (1) algorithm for implementing the lfu cache eviction scheme. *arXiv preprint arXiv:2110.11602*, 2021.

[33] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61, 2015.

[34] Michael L Minion. Semi-implicit projection methods for incompressible flow based on spectral deferred corrections. *Applied numerical mathematics*, 48(3-4):369–387, 2004.

[35] NetApp. What is high performance computing? `https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/`. Accessed: 2023-07-25.

[36] Nils Berglund. Stochastic allen-cahn equation. `https://www.youtube.com/watch?v=mu4QeOMIv74`. Accessed: 2023-07-07.

[37] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388, 2012.

[38] Swann Perarnau, Judicael A Zounmevo, Balazs Gerofi, Kamil Iskra, and Pete Beckman. Exploring data migration for future deep-memory many-core systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 289–297. IEEE, 2016.

[39] Sansriti Ranjan, Dakota Fulp, and Jon C. Calhoun. Exploring the impacts of software cache configuration for in-line compressed arrays. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2022.

[40] Daniel Ruprecht and Robert Speck. Spectral deferred corrections with fast-wave slow-wave splitting. *SIAM Journal on Scientific Computing*, 38(4):A2535–A2557, 2016.

[41] Somayeh Sardashti, André Seznec, and David A Wood. Skewed compressed caches. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342. IEEE, 2014.

[42] Somayeh Sardashti, Andre Seznec, and David A Wood. Yet another compressed cache: A low-cost yet effective compressed cache. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(3):1–25, 2016.

[43] Somayeh Sardashti and David A Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 62–73, 2013.

[44] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. Memzip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 638–649. IEEE, 2014.

[45] Shahriar Tajbaksh. Understanding write-through, write-around and write-back caching (with python). `https://shahriar.svbtle.com/Understanding-writethrough-writearound-and-writeback-caching-with-python`. Accessed: 2023-07-07.

[46] Robert Speck. Parallelizing spectral deferred corrections across the method. *Computing and visualization in science*, 19:75–83, 2018.

[47] Robert Speck. Algorithm 997: pysdc—prototyping spectral deferred corrections. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–23, 2019.

[48] Robert Speck, Daniel Ruprecht, Matthew Emmett, Michael Minion, Matthias Bolten, and Rolf Krause. A multi-level spectral deferred correction method. *BIT Numerical Mathematics*, 55(3):843–867, 2015.

[49] Robert Underwood, Victoriana Malvoso, Jon C Calhoun, Sheng Di, and Franck Cappello. Productive and performant generic lossy data compression with libpressio. In *2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7)*, pages 1–10. IEEE, 2021.

[50] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18:15–28, 2015.

[51] Ke Wang, Abhishek Kulkarni, Michael Lang, Dorian Arnold, and Ioan Raicu. Using simulation to explore distributed key-value stores for extreme-scale system services. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[52] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on hpc systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 799–808. IEEE, 2014.

[53] Martin Weiser. Faster sdc convergence on non-equidistant grids by dirk sweeps. *BIT Numerical Mathematics*, 55(4):1219–1241, 2015.

[54] Miłosz Wieczor, Vito Genna, Juan Aranda, Rosa M Badia, Josep Lluís Gelpí, Vytautas Gapsys, Bert L de Groot, Erik Lindahl, Martí Municoy, Adam Hospital, et al. Pre-exascale hpc approaches for molecular dynamics simulations. covid-19 research: A use case. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 13(1):e1622, 2023.

[55] Wikepedia. Heat trasfer. `https://en.wikipedia.org/wiki/Heat_transfer`. Accessed: 2023-07-07.

[56] Dongfang Zhao, Da Zhang, Ke Wang, and Ioan Raicu. Exploring reliability of exascale systems through simulations. In *SpringSim (HPC)*, page 1, 2013.

[57] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 61–70, 2014.