Chunked Extendible Arrays and its Integration with the Global Array Toolkit for Parallel Image Processing

Gideon Nimako

A thesis submitted to the Faculty of Engineering and the Built Environment in fulfilment of the requirements for the degree of

Doctor of Philosophy

Supervisors:

Prof. Ekow Otoo
School of Electrical and Information Engineering
Faculty of Engineering & the Built Environment
University of the Witwatersrand

Prof. Michael Sears
School of Computer Science
Faculty of Science
University of the Witwatersrand

DECLARATION

I declare that this Thesis is my own, unaided work under the supervision of Prof. Ekow Otoo and Prof. Michael Sears. It is being submitted for the Degree of Doctor of Philosophy in Computer Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

_____

28th October, 2016

ABSTRACT

Several meetings of the Extremely Large Databases Community for large scale scientific applications have advocated the use of multidimensional arrays as the appropriate model for representing scientific databases. Scientific databases gradually grow to massive sizes of the order of terabytes and petabytes. As such, the storage of such databases requires efficient dynamic storage schemes where the array is allowed to arbitrarily extend the bounds of the dimensions. Conventional multidimensional array representations in today's programming environments do not extend or shrink their bounds without relocating elements of the data-set. In general extendibility of the bounds of the dimensions is limited to only one dimension. This thesis presents a technique for storing dense multidimensional arrays by *chunks* such that the array can be extended along any dimension without compromising the access time of an element. This is done with a computed access mapping function that maps the k-dimensional index onto a linear index of the storage locations. This concept forms the basis for the implementation of an array file of any number of dimensions, where the bounds of the array dimension can be extended arbitrarily. Such a feature currently exists in the Hierarchical Data Format version 5 (HDF5). However, extending the bound of a dimension in the HDF5 array file can be unusually expensive in time. Such extensions, in our storage scheme for dense array files, can be performed while still accessing elements of the array at orders of magnitude faster than in HDF5 or conventional array-files. We also present Parallel Chunked Extendible Dense Array (PEXTA), a new parallel I/O model for the Global Array Toolkit. PEXTA provides the necessary Application Programming Interface (API) for explicit data transfer between the memory resident global array and its secondary storage counterpart but also allows the persistent array to be extended on any dimension without compromising the access time of an element or sub-array elements. Such APIs provide a platform for high speed and parallel hyperspectral image processing without performance degradation, even when the imagery files undergo extensions.

Dedicated to

my mother, Elizabeth Aba Mensah

my father Akwasi Nimako (1950-1993) and
my guardian Alock K. Asamoah Addo.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my principal supervisor, Prof. Ekow Otoo for his technical advice, encouragement, support and the many hours of stimulating discussions. He taught me both consciously and unconsciously, how good computational science research is done. The joy and enthusiasm he has for his research was contagious and motivational for me, even during tough times in the PhD pursuit. I am thankful for the excellent example he has provided.

I am also very thankful to my co-supervisor, Prof. Michael Sears for his time in discussing various aspects of my PhD work, particularly his advice on the hyperspectral image analysis using the PGAS model. On many occasions, I simply walk to his office without an appointment and he always makes time for discussions. I am very grateful for his directions, advice and motivation.

Special thanks also go to the people within our research group (Daniel O. Kwofie and Hairong Wang). I'd like to thank them for their friendship, moral support and inspiration. I completed the work in this thesis in December, 2013 but writing up the manuscript took longer than expected. It was the encouragement and motivation of my Head of School Prof. Laetitia Rispel and my Head of Division, Prof. Tobias Chirwa that brought this work to completion. I really appreciate their efforts and leadership.

Finally, I would like to express my gratitude to my sponsors, the Center for High Performance Computing (CHPC) for their support. Without such financial support, the realization of this work would have been impossible.

List of Publications

1. E.J. Otoo, Gideon Nimako, Daniel Ohene-Kwofie, ***Single Matrix Block Shift (SMBS) Variant of Cannon's Matrix Multiplication Algorithm***, *Under Review*, ParCo, 2015, ACM.

2. E.J. Otoo, Daniel Ohene-Kwofie, Gideon Nimako, ***Concurrent Operations of O2-Tree on Shared Memory Multicore Architectures***, EAI Endorsed Transactions on Scalable Information Systems, Volume 3, 2014, DOI: 10.4108/sis.1.3.e6, URL : http://dx.doi.org/10.4108/sis.1.3.e6

3. E.J. Otoo, Gideon Nimako, Hairong Wang, ***New Approaches to Storing and Manipulating Multi-Dimensional Sparse Arrays***, SSDBM 2014, ACM, Article No 41, ISBN: 978-1-4503-2722-0, DOI: 10.1145/2618243.2618281, URL: http://dl.acm.org/citation.cfm?id=2618281

4. Daniel Ohene-Kwofie, Ekow Otoo, Gideon Nimako. ***Concurrent Operations of O_2-Tree on Shared memory Multicore Architecture***, Proceedings of the 24th Australasian Database Conference (ADC 2013), Adelaide, South Australia, Volume 137, ISBN: 978-1-921770-22-7, URL: http://dl.acm.org/citation.cfm?id=2525420

5. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***Chunked Extendible Dense Arrays for Scientific Data Storage***, Journal of Parallel Computing, Volume 39, Issue 12, December 2013, Pages 802–818, URL: http://dx.doi.org/10.1016/j.parco.2013.08.006

6. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***PEXTA: A Parallel Chunked Extendible Dense Array I/O for Global Array (GA)***, IEEE

Cluster Computing, 2013, ISBN: 978-1-4799-0898-1, DOI: 10.1109/CLUSTER.2013.6702688, URL: http://ieeexplore.ieee.org/document/6702688/

7. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***Chunked Extendible Dense Arrays for Scientific Data Storage***, ICPP 2012, Pages 38-47, IEEE, ISBN: 978-1-4673-2509-7, DOI: 10.1109/ICPPW.2012.9, URL: http://ieeexplore.ieee.org/document/6337461/

8. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***Using Chunked Extendible Array for Physical Storage of Scientific Datasets***, HPCDB'12 - Proceedings of the 2012 High-Performance Computing Meets Databases, Pages 17-20, DOI: 10.1109/SC.Companion.2012.164, ISBN: 978-0-7695-4956-9, URL: http://ieeexplore.ieee.org/document/6495946/

9. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***Fast Parallel Algorithms for Blocked Dense Matrix Multiplication on Shared Memory Architectures***, ICA3PP 12, volume 7439 of Lecture Notes in Computer Science, Pages 443-457, ISBN: 978-3-642-33077-3, Springer, DOI: 10.1007/978-3-642-33078-0_32, URL: http://link.springer.com/chapter/10.1007/978-3-642-33078-0_32

10. Gideon Nimako, E.J. Otoo, Daniel Ohene-Kwofie, ***Cache-sensitive MapReduce DGEMM algorithms for shared memory architectures***, SAICSIT 12, Pages 100-110, ISBN: 978-1-4503-1308-7, ACM, (2012), DOI: 10.1145/2389836.2389849, URL: http://dl.acm.org/citation.cfm?id=2389849

11. Daniel Ohene-Kwofie, E.J. Otoo, Gideon Nimako , O_2-***Tree: A Fast Memory Resident Index for NoSQL Data-Store***, EUC '12, Pages 50-57, ISBN: 978-1-4673-5165-2, IEEE, DOI: 10.1109/ICCSE.2012.17, URL: http://ieeexplore.ieee.org/document/6417274/

Contents

List of Figures

List of Tables

List of Abbreviations

AMEE.....Automated Morphological Endmember Extraction

API.....Application Programming Interface

AVIRIS.....Airborne Visible / Infrared Imaging Spectrometer

DRA.....Disk Resident Array

EA.....Extendible Array

GA.....Global Array

HDF5.....Hierarchical Data Format version 5

HyMap.....Hyperspectral Mapper

IXA.....Index-Array

KDEA.....K-Dimensional Extendible Array

KEA.....Karnaugh Representation of Extendible Array

MEI.....Morphological Eccentricity Index

MIMD.....Multiple Instruction, Multiple Data

MNF.....Maximum Noise Fraction

MOLAP.....Multi-dimensional On-Line Analytical Processing

MPI.....Message Passing Interface

netCDF.....Network Common Data Form

NUMA.....Non-Uniform Memory Access

OLAP.....Online Analytical Processing

PCA.....Principal Component Analysis

PCT.....Principal Component Transform

PEXTA.....Parallel Chunked Extendible Dense Array

PGAS.....Partitioned GlobalAddress Space

PPI.....Pixel Purity Index

SciDB.....Scientific Database

UPC.....Unified Parallel C

1

Introduction

The first chapter introduces Extendible Arrays and highlights the motivation of the research conducted in the thesis

Contents

Large scale scientific data are most often stored as large multidimensional arrays. Arrays are extensively used for scientific data management because they are well understood and can easily be incorporated in other data structures. In [2], the new science database system (called SciDB), that is being advocated as a common platform for almost all scientific datasets, is based on a data model organised as multidimensional array. In general k-dimensional arrays, represented in linear consecutive locations, cannot extend without reallocation of already stored elements. The extendibility is limited to only one dimension. This degrades performance of various array operations particularly in scientific and engineering applications that sometimes undergo interleaved extensions. For instance, in data processing applications of satellite imagery and remote sensing, the datasets captured along defined swathes, spectral bands, time, etc., may require incremental tiling of adjacent scenes and progressive inclusion of selected bands. In Multi-dimensional On-Line Analytical Processing (MOLAP) of data warehouses, the basic data representation is a multi-dimensional array where each dimension represents discrete valued attribute. For example, the number of items QTY sold by a company that sells many different items in different stores on any particular day, can be captured as a multi-dimensional array *QTY(ItemId, StoreId, Day). ItemId* and *StorId* denote unique identifiers for an item and a store respectively. Each cell is defined by a vector of $\langle ItemId,\ StoreId,\ Day\rangle$ and the value stored in a cell represents the quantity (QTY) sold. Extendibility of the dimension representing *ItemId* is necessary if new items are sold by the company. Similarly, the dimension representing *StoreId* will be required to extend if the company increases its number of stores. The use of coventional array to represent *QTY()* requires reorganising the allocated storage

whenever a dimension expands. Extendible arrays [3–9], on the other hand, can handle dynamic growth of the bounds of the dimensions without reorganising already allocated array elements.

1.1 Motivation

Modern programming languages provide native support for multidimensional arrays. The mapping function $\mathcal{F}()$ is normally defined so that elements are allocated either in *row-major* or *column major* order. We refer to arrays whose elements are allocated in this manner as *conventional arrays.* Conventional arrays limit their growth to only one dimension. The consequence is that, since the dimensions cannot be extended at run time, programs are forced to declare at compile time an array size that is sufficiently large for running most applications. Recent compilers allow representation of extendible array using an Iliffe vector [10]. Iliffe vectors use a one-dimensional array of references to arrays of one dimension less. For two dimensions, the alternative structure would be a vector of pointers to vectors, one for each row. Element accesses are by pointer chasing. This approach is not suitable for storing elements of arrays in a file. If we consider a mapping function that allows allocating an array of some small size initially and then incrementally expanding it at run-time to accommodate just the right size of the array, without abandoning the efficient address calculation, then such a mapping function is exactly what is needed for the storage of multidimensional array files.

Consider the maintenance of a very large array file. We desire extendibility of such files by allowing the index range of any dimension to grow so that new array elements

that lie in the extended index range can simply be appended to the file without reorganising the entire file. We present such a mapping function $\mathcal{F}^*()$ and its inverse $\mathcal{F}^{*-1}()$ for addressing elements of an array that is allowed to extend arbitrarily. The method simply carries over the mapping function for dense multidimensional extendible arrays in main memory to addressing array elements in a file stored on disk.

In conventional k-dimensional arrays, efficient computation of the mapping function is carried out with the aid of a vector that holds k multiplicative coefficients of the respective indices. This is sometimes referred to as the *dope vector*. Consequently, an N-element array in k-dimensions can be maintained in $O(N)$ storage locations and the computation of the linear address of an element, given its k-dimensional coordinate index, is done in time $O(k)$. We achieve similar efficiency in the management of extendible arrays by maintaining vectors of multiplicative coefficients each time the array expands. The computation of the linear address, corresponding to a k-dimensional index, uses these stored vectors of coefficients. The vectors of multiplicative coefficients capture the history of the expansions and are organised as the *leaf* nodes of a *Leaf-Linked Red-Black Tree* which we refer to as the O_2-Tree [11]. For a general schematic illustration of the O_2-Tree, see Figure 4.2.2 in Chapter 4. The O_2-Tree allows us to compute the mapping function and its inverse function in times comparable to that of a conventional array. Suppose a k-dimensional array has N elements and after some arbitrary extensions, dimension j maintains $\mathcal{E}_j$ expansion records. Our approach computes the linear address of the k-dimensional index in time $O(k + \sum_{j=0}^{k-1} \log(\mathcal{E}_j))$ using $O(k \sum_{j=0}^{k-1} \mathcal{E}_j)$ additional space. The worst-case additional storage occurs when, the array grows by *cubical expansions* and $\mathcal{E}_j = N^{1/k}$. The

results then reduce to a computable access function in time $O(k + \log N)$ using addition space of $O(k^2 N^{1/k})$ for an N element array. Under such cubical expansions, each expansion step adds $N^{1-1/k}$ new elements.

When arrays are allocated in files, we try to align the I/O of the array elements with the *paging schemes* of the array file. As a consequence we organise the array cells into chunks such that reading and writing of array elements are done as chunks of array elements. Accessing array chunks can then be done through file caching as well.

In this thesis, we present an efficient scheme for organising extendible arrays by *chunking*. Note that the mapping function, previously used to directly address array elements, is now used to address the array chunks. The function $\mathcal{F}^*()$ is used to compute the linear address locations of the chunks when given its k-dimensional index. The input to this function is a data structure of *axial-vectors* that hold the history of expansions of the extendible array. Instead of using a 1-dimensional axial-vector for each dimension, we use the O_2-Tree to organise the axial-vector for each dimension (see Figure 4.2.2). The O_2-Tree is an augmented Red-Black Tree in which the leaf nodes are doubly linked index blocks that store records of key-value pairs. Thus, we store the records of our axial-vectors as the leaf nodes of the O_2-Tree structure.

As high-end computing moves towards petascale and exascale, most scientific applications are adopting Message Passing Interface (MPI) [12] and Partitioned Global Address Space (PGAS) programming models to combine the advantages of shared and distributed programming models. For most parallel scientific applications that

use these programming models, the scalability limitation comes from the performance of I/O operations. The Global Array Toolkit(GA) [13, 14], which is one of the early PGAS models, introduced Disk Resident Array(DRA) [15], to extend the GA programming model to secondary storage (see Figure 1.1.1). The DRA is used to persist the elements of memory resident *global array*. It provides user-defined virtual memory by allocating arrays that are too big to fit in the global memory into disk resident arrays and transfers sections into the aggregated memory for computations. It provides a disk-based representation of arrays and the necessary functions to transfers blocks of data between the memory resident global arrays and its counterpart to disk storage. This gives programmers the needed utilities to access data on disk but modelled as arrays, rather than files. In this thesis, we illustrate how chunked extendible arrays [16, 17], can be used as an efficient I/O for the global array toolkit. The technique is to view PEXTA (Parallel Chunked Extendible Dense Array) as a large array mapped into a file whose content can be read into the global memory. A PEXTA file has two components: a principal array data file and a metadata file of the axial vectors that are also stored on disk. We term these the *principal array file* and the *metadata file* respectively. The principal-array file grows continuously by appending new data elements (in *chunks*) whenever a dimension is extended. Although the principal-array file is stored on disk, it is accessed in units of chunks through an in-memory cache. The metadata file is usually memory resident but stored on disk after every session and loaded into memory before any usage session starts.

Our PEXTA I/O can be mapped to standard scientific and imagery file formats such as NetCDF, GeoTIFF, CADRG, HDF5, etc. Such mapping enables near real-

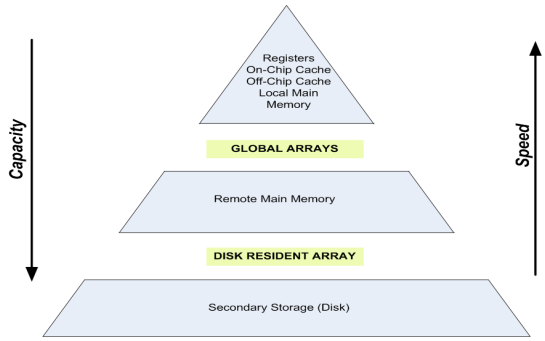


Figure 1.1.1: Global Array Memory Hierarchy

time image processing task such as *hyperspectral unmixing*. Spectral unmixing is a computational estimation of the number of endmembers, their spectral signatures and abundance fractions. This process is usually computationally intensive and as such several techniques have been developed to automate and parallelize this task [18–20] using MPI implementation. However, such techniques are not susceptible to real-time processing due to the continuous tiling of swathes and spectral bands. In this thesis, we present a model of computation and storage that achieves near real-time processing. This is done by mapping standard imagery file formats onto the PEXTA scheme. Once such mapping scheme is applied, PEXTA exposes all GA basic array and linear algebra operations for high speed parallel processing.

1.2 Main Contributions

The major contributions of this thesis is in the development and implementation of an efficient computed access scheme for chunked extendible dense array and its integration with the Global Array Toolkit for parallel image processing. The main results being reported here include:

- The implementation of a chunked computed access scheme for both memory resident and disk resident dense extendible array but with emphasis on the disk resident extendible arrays.
- An implementation of a tree-indexing structure, the O_2-Tree, for the axial-vector of extendible array.
- An analytical and experimental performance analyses of our approach compared with the conventional array technique of organising extendible arrays.
- An implementation of the PEXTA API as a drop-in replacement for the Disk Resident Array of the GA-Toolkit
- An application of PEXTA to hyperspectral image analysis

1.3 Organisation of Thesis

The remainder of this thesis is organised as follows. Chapter 2 gives the background and related works that have been done on extendible arrays. In Chapter 3, we explain the concept of In-Memory Extendible Dense Arrays. We give detailed descriptions of our approach of organising extendible dense arrays and some performance analysis. Chapter 4 discusses the chunking technique for extendible array and the tree-indexing scheme, i.e., the O_2-Tree, for the axial-vectors. In Chapter 5, we illustrate various allocation strategies at the chunk/block level and for the single elements within blocks. We evaluate the impact that Space-Filling-Curve allocations within the chunks have on the access time of our scheme. Chapter 7 describes translation functions that map extendible arrays files to standard file formats such as HDF5 and NetCDF. We

also illustrate how the chunking scheme in HDF5 differ from ours. In Chapter 8, we introduce PEXTA, a new parallel I/O model for the Global Array Toolkit. We demonstrate how our parallel I/O scheme differs from parallel HDF5 file accesses. Chapter 9 describes the application of PEXTA APIs for Hyperspectral Image Analysis. We illustrate how to implement Parallel Pixel Purity Index (PPI), N-Findr and Automated Morphological Endmember Extraction (AMEE) Endmember Extraction using the PGAS models. We summarise the work of this thesis in Chapter 10 and give some directions for future work; namely adopting our technique for Sparse Multidimensional Arrays and their application in Data Warehousing.

2

Background and Related Works

This chapter gives the background and related works that has been done on Extendible Arrays

Contents

The issue of extendible arrays dates back to the era when high level programming languages emerged. Until recently, there has not been any satisfactory technique for handling scientific operations involving extendible arrays (i.e., arrays whose sizes in various dimensions change in the course of executing an operation). However, questions concerning how to manipulate and implement arrays have received much attention. The array grammars described in [21], gave the theoretical foundation for array representation. A thorough discussion of graph-oriented view of arrays was given in [22]. This discusses a graph-oriented view of data structures in general. The idea was to exploit the structural regularities in constructing storage mappings for arrays which are usually expressed in mathematical models.

2.1 Computed Access Schemes for Arrays

In [5], two strategies for constructing readily extendible allocation schemes for arrays were proposed. The allocation scheme by shells (shells are synonymous to array chunks/blocks), assumes arrays expand in a regular manner. In that case, the array grows by addition of shells and each shell is viewed as continuous in a specified space and subsequently used to specify the shapes of the neighbourhood. The second strategy, stencil and segmentation, viewed an array as being made up of blocks or sub-arrays, rather than atomic elements, such as floating point numbers. The array in this scheme is partitioned into blocks of equal sizes. Element access is performed through a pair of functions; one giving the address of the header of the block and the other giving a displacement within the block (compute access scheme). The one-to-one infinite range map is given as the sum of the many-to-one infinite range

blocks and the many-to-one finite range displacements.

There are two issues with these two schemes. Firstly, they depend on one's ability to identify exploitable characteristics of the array to be implemented, in order to translate these peculiarities into the specification of the shells and to find ways to enumerate the shells. The problem is that each scheme devised in this manner works well only on specific classes of arrays; namely, those used in constructing the shells. Thus, the importance of these strategies lies neither in their generality nor in automating the process of array realization. It lies rather in establishing a framework in which the construction of the extendible realization is rendered systematic. Subsequent improved schemes for realizing extendible arrays have been addressed in the literature, notably in [6]. These schemes may be classified as computed access, linked allocations and hashing.

2.2 KDEA : K-Dimensional Extendible Arrays

In [3], an Index Array Scheme based on computed access was introduced. The scheme perceives the storage as consisting of a k-dimensional rectangular array that is allowed to extend by adjoining blocks of (k-1)-dimensional sub-array on any dimension. This type of array was termed *K-Dimensional Extendible Array* (KDEA). An array was kept for storing the multiplicative coefficients for indices in the KDEA and the block starting address at the instant of extending a dimension. This array was called *Index-Array* (IXA). The scheme considers the KDEA as being partitioned into blocks, each block being a sub-array of $k-1$ dimensions. If the incremental step size at each expansion is constant for each dimension, the Index-Array method achieves

$O(k)$ access cost with $O(N)$ worst case storage requirement (where N is number of elements in array). However if the step size is not constant, then the element access cost is $O(\log N)$ in the worst case scenario.

2.3 Karnaugh Representation of Extendible Arrays

In [23], the *Karnaugh Representation of Extendible Array* (KEA) which is based on mapping function [3] was introduced. In this scheme, a k-dimensional array is represented by a set of 2-dimensional arrays based on the idea of Karnaugh maps. The scheme considers extendible arrays as a combination of sub-arrays, just as in [3]. If the array is k dimensional, then the sub-array is $k-1$-dimensional, similar to the concept in [3]. Instead of axial vectors, this scheme uses three types of auxiliary tables namely, history table, coefficient table, and address table. For each dimension, three tables are kept, which take up a lot of disk or memory space as compared to a single axial vector for each dimension in [4]. The address table contains the first address of the sub-array, just as the axial vector; history table contains the construction history of sub-arrays and the coefficient table contains the coefficients of the $k-1$ dimensional sub-array.

2.4 Other Representations of Extendible Arrays

A number of research works has been done with the mapping function introduced in [3]. One such work is an index structure that keeps track of the extensions of disk resident arrays for Online Analytical Processing (OLAP) [24]. The technique in the

scheme is to build a tree-based index that will keep track of the growth of the array in any dimension and even allows adding new dimensions. The index in this case is a search tree (implemented as a B-tree) based on a compound key (D, V), where D represent the dimension and V the new range this dimension achieves after growth. Each key will point at an associated record containing some information about the extension. Extending the number of dimensions of an array is not addressed in this thesis.

2.5 Parallel Hyperspectral Image Analysis

Extraction of spectral endmembers from hyperspectral datasets is computationally complex and intensive, in particular, for very high-dimensional datasets [25]. However, the structural properties of hyperspectral datasets is amenable to parallel processing. This has led to a number of parallel implementations for endmember extraction algorithms. In [25], parallel implementation of N-FINDR and pixel purity index (PPI) endmember extraction algorithms were given. These two algorithms are susceptible to direct parallel implementation because the input dataset can be divided into multiple blocks made up of entire pixel vectors. This is known as *spatial-domain partitioning*. This approach has less data dependencies and a reduced inter-process communication. Their implementation was MPI (Message Passing Interface) based using AVIRIS Cuprite scenes.

Similar parallel implementations of hyperspectral image analysis have been proposed in [20, 26–28]. The motivation for such implementations is to speed up the processing in order to achieve near real-time analysis. Recently, there have been a number

of efforts towards GPU parallel endmember extraction implementations [29–31] for further speed-ups in processing time. However, one pending problem is the challenge of performing hyperspectral analysis whiles continuously tiling swathes/spectral bands from airborne scenes. Such datasets usually get appended to already allocated datasets. In this thesis we address this problem by providing a storage scheme that allows interleaving processing/analysis with continuous tiling.

2.6 Summary of Literature Review

There has been quite a number of efforts towards extendible arrays structures; from Compute Access Schemes to Generalized Mapping Functions. However, the problem of efficiently manipulating extendible array files still remains a challenge. Current compilers and programing environments do not have APIs and libraries that implement extendible array files. Extendibility of arrays in languages such as C/C++ is done by memory reallocation. This research contribution is in developing and implementing these APIs for both memory resident and disk resident extendible arrays using mapping functions. The implementation for the disk resident extendible array is extended to allow easy integration into other platforms and APIs such as the Global Array Toolkit. This extension enables array files of different formats (such as NetCDF, CADRG, BIL, GeoTIFF, HDF5) to be read into Global Array. Once they are read into Global Array, they will readily have the extendibility feature provided by disk resident extendible array scheme.

3

Memory Resident Extendible Dense Arrays

This chapter explains the concept of In-Memory Extendible Dense Arrays

Contents

The multidimensional dense array structure is one of the most widely used and fundamental data structure and has wide applications in engineering and mathematical sciences. Computed access techniques for mapping elements of dense arrays onto contiguous storage location have the advantages of fast element access and efficient storage utilization [3]. In this chapter, we present the basic foundation for the multidimensional dense array structure and the mapping functions that describe extendible dense arrays.

3.1 Conventional Array Mapping Function

The mapping function for conventional array is first examined. Consider a k-dimensional array $A[\mathbb{U}_0][\mathbb{U}_1][\mathbb{U}_2]...[\mathbb{U}_{k-1}]$ where $[\mathbb{U}_j], 0 \leq j \leq k-1$, denote the bounds on the index ranges of the respective dimensions and k denotes the number of dimensions. Let $L[n] = \{L\langle 0\rangle, L\langle 1\rangle, \ldots L\langle n-1\rangle\}$ where $n = \prod_{j=0}^{k-1} \mathbb{U}_j$, be a set of consecutive storage locations with the base location being $L\langle 0\rangle$. If an element of a k-dimensional array is denoted by $A\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ (alternatively by $A[i_0][i_1][i_2]...[i_{k-1}]$), then an allocation function $\mathcal{F} : \mathbb{R}^k \rightarrow \mathbb{R}$, is a mapping that assigns to each element, $A\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$, a distinct location $L\langle q\rangle$ where q is derived from the array index as $q = \mathcal{F}(i_0, i_1, i_2, ..., i_{k-1})$. The function $\mathcal{F}()$ is said to realise A. The mapping function $\mathcal{F}()$ maps the k-tuple of indices, one-to-one, onto n consecutive locations. This method of array realisation is referred to as a *computed access method* [5]. Other methods of realisation that have been used for array representations are link lists, orthogonal list and hashing techniques [6]. What makes the computed access method attractive is the advantages it exhibits and its

applications compared to other realisation methods or data structures. These are that it:

- provides easy probing of the array, which are lacking in linked list schemes.
- allows easy traversal along rows and columns of the array, which are not present in hashing schemes.

Definition 3.1.1. *A realisation of the array* $A[\mathbb{U}_0][\mathbb{U}_1]...[\mathbb{U}_{k-1}]$ *in* $\{L\langle 0\rangle, L\langle 1\rangle, ..., L\langle n-1\rangle\}$ *for* $n = \prod_{j=0}^{k-1} \mathbb{U}_j$*, is a mapping function,* $\mathcal{F} : \mathbb{U}^k \to L$*, of the elements of A, one-to-one, onto the address,* $L = \{0, 1, ..., n\}$ *with* $\mathcal{F}(0, 0, ..., 0) = 0$

Examples of this realisation of bounded arrays are the row-major (called *lexicographic* order) and column major order function. These are defined as follows:

(a) Row major realisation

Given an array element $A[i_0][i_1]...[i_{k-1}]$, the location q is computed as

$$q = \mathcal{F}(i_0, i_1, i_2, ..., i_{k-1}) = s_0 + i_0C_0 + i_1C_1 + ... + i_{k-1}C_{k-1} \quad (3.1.1)$$

where $$C_j = \prod_{r=j+1}^{k-1} \mathbb{U}_r,$$

$$0 \leq j \leq k-1 \text{ and } C_0 = 1$$

(b) Column major realisation

Given an array element $A[i_0][i_1]...[i_{k-1}]$, the location q is computed as

$$q = \mathcal{F}(i_0, i_1, i_2, ..., i_{k-1}) = s_0 + i_0C_0 + i_1C_1 + ... + i_{k-1}C_{k-1} \quad (3.1.2)$$

where $$C_j = \prod_{r=0}^{j-1} \mathbb{U}_r,$$

$$0 \leq j \leq k-1 \text{ and } C_{k-1} = 1$$

s_0 is the base location address and the convention is that an empty product is taken as 1. For any k-dimension index, the computations of the linear address using Equation 3.1.1 and Equation 3.1.2 take $O(k)$ elementary operations of multiplications and additions. The limitation imposed by $\mathcal{F}()$ is that extensions of the array can only be done on one dimension; that is dimension 0 (in the case of row major), since the evaluation of the function does not involve $\mathbb{U}_0$. This constraint imposed by $\mathcal{F}()$ is usually exploited by rearranging the order of the indices whenever computing the linear addresses of the location when an array is extended along any dimension. For example, suppose the array is extended along dimension j. Then the address calculations of the newly allocated $(k-1)$-dimensional hyperslab is done as if dimension j is constrained to be dimension 0.

3.2 Conventional Array Inverse Mapping Function

A little used inverse computation function $\mathcal{F}^{-1}$, is the inverse calculation from a linear address q to a $k-$dimensional index $\langle i_0, i_1, \ldots, i_{k-1} \rangle$. This is defined as

$$\mathcal{F}^{-1}(q) \rightarrow \langle i_0, i_1, \ldots, i_{k-1} \rangle$$

Such an inverse function is handy when computing the neighbouring cells of an array element given the linear address of a specific one. This is illustrated in Algorithm 1.

Algorithm 1: Computing the k-dimensional indices, given the linear address q.

input : q, the linear address of the k-dimensional index, and $\mathbb{U}_0, \mathbb{U}_1, \ldots, \mathbb{U}_{k-1}$, the bounding values of the respective dimensions of the array.

output : $\langle i_0, i_1, i_2, ..., i_{k-1} \rangle$, k-dimensional indexes

begin

```
for j ← k − 1 down to 1 do
    i_j ← q mod U_j ;
    q ← q/U_j ;
i_0 ← q ;
```

3.3 Extendible Dense Arrays Mapping Function

In this section we present the mapping function of extendible arrays that forms basis of the main contribution of this thesis. We discuss the mapping function for in-memory extendible dense arrays. We require a mapping function with the following properties:

- simplicity of arbitrary element access
- simplicity of traversal along a path
- simplicity of extension and
- efficiency of storage utilization.

If $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ represents a k-dimensional array, then $\mathbb{U}_j^*$ denotes the bound that has the ability to grow as opposed to a fixed bound $\mathbb{U}_j$ given in the conventional array discussed in Section 3.1. Similarly we employ the notation $\mathcal{F}()$ when referring to conventional array mapping function that allows extendibility in only one dimension and $\mathcal{F}^*()$ when referring to a mapping function that allows extendibility in any dimension.

Definition 3.3.1. *A k-dimensional extendible array is a k-dimensional array* $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ *in which the upper bounds of the various dimensions are allowed to increase indefinitely and in any order.*

The superscript star sign ('*') is used to denote the fact that the upper bounds of the index ranges are allowed to vary.

Definition 3.3.2. *An extendible array in which the dimensionality k, is also allowed to vary by taking increasing values of positive integers, is referred to as an extendible array of varying dimensions and is denoted by* $A^*[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$

We use the superscript "*" in A^* to reflect the fact that the dimensionality is allowed to vary. In this thesis however, we do not address such extendibility of the dimensionality of an array.

Definition 3.3.3. *A realisation of an extendible array* $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ *, is a mapping function,* $\mathcal{F}^* : \mathbb{R}^k \rightarrow \mathbb{R}$*, that maps elements of A, one-to-one, onto storage locations* $\{L\langle 0\rangle, L\langle 1\rangle, ..., L\langle n-1\rangle\}$ *such that with* $\mathcal{F}^*(0, 0, ..., 0) = 0$ *where* $n = \prod_{j=0}^{k-1} \mathbb{U}_j^*$*, and also varies with changing values of* U_j^**'s..*

The mapping function for extendible array uses axial-vectors to store information needed to compute the function. Let $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$ be an arbitrary 3-dimensional array. Figure 3.3.1 shows an initially allocated 3-dimensional array of dimensions $A[3][3][2]$. This in general is denoted by $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$, where, $\mathbb{U}_0^* = 3, \mathbb{U}_1^* = 3$ and $\mathbb{U}_2^* = 2$, are the upper bounds. The array was extended along dimension 1 by increasing the index range from 3 to 5; i.e. by adjoining 2 columns to the original array.

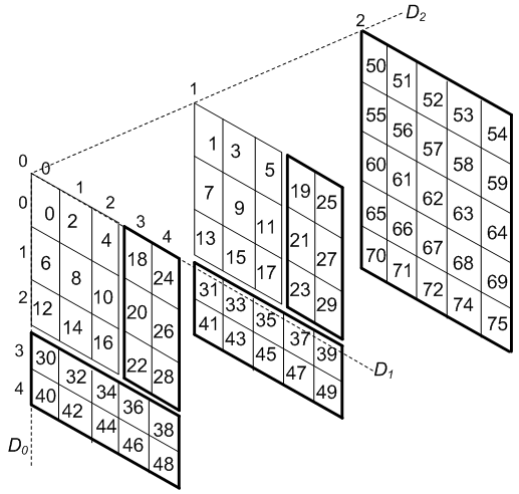


Figure 3.3.1: Storage Location for a 3-Dimensional Extendible Array

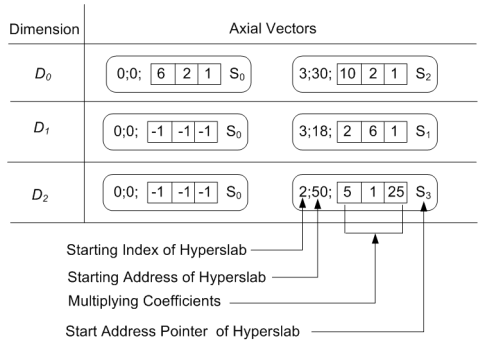


Figure 3.3.2: Axial-Vectors of Figure 3.3.1

With this extension, an element $A\langle 1,4,0\rangle$ is assigned to location 26 and $A\langle 2,4,1\rangle$, location 29. The array was thereafter extended along dimension 0 by increasing the index range from 3 to 5 and finally on dimension 2, from 2 to 3. For generalisation, suppose that in a k-dimensional extendible array $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$, dimension l is extended by λ_l, then the index range increases from $\mathbb{U}_l^*$ to $\mathbb{U}_l^* + \lambda_l$. The idea is to allocate a k-dimensional block of array elements which we refer to as a *hyperslab* or *segment*, so that addresses are computed as displacement from the location of element $A\langle 0,0,...,\mathbb{U}_l^*,...,0\rangle$. Let the location $A\langle 0,0,...,\mathbb{U}_l^*,...,0\rangle$ be denoted as $\ell_{\mathbb{Z}_l^*}$

where $\mathbb{Z}_l^* = \prod_{r=0}^{k-1} \mathbb{U}_r^*$. The desired mapping function $\mathcal{F}^*()$ that computes the address of q^* of an element $A(\langle i_0, i_1, i_2, ..., i_{k-1}\rangle)$ is given by:

$$q^* = \mathcal{F}^*(\langle i_0, i_1, i_2, ..., i_{k-1}\rangle) = \mathbb{Z}^0_{\mathbb{U}_l^*} + (i_l - \mathbb{U}_l^*)C_l^* + \sum_{\substack{j=0 \\ j\neq l}}^{k-1} i_j C_j^* \quad (3.3.1)$$

where $$C_l^* = \prod_{\substack{j=0 \\ j\neq l}}^{k-1} \mathbb{U}_j^*$$

and $$C_j^* = \prod_{\substack{r=j+1 \\ r\neq l}}^{k-1} \mathbb{U}_r^*$$

$\mathbb{Z}^0_{\mathbb{U}_l^*}$ denotes the maximum starting address of the hyperslab that is adjoined. l denotes the dimension that was extended. $\mathbb{U}_l^*$ denotes the bound of the index range before the expansion. All these values and the multiplicative coefficients needed to address elements in the adjoined hyperslab must be maintained in some data structure that will allow easy retrieval of them for the computation of an element's address within the adjoined hyperslab. The axial-vector is used for this purpose.

Entries into the axial-vector consist of a 1-dimensional vector, $v[k+3]$ of $k+3$ values and a pointer, denoted by s_j, to the allocated memory for the hyperslab. It is noteworthy that, for in-memory allocations, the entire extendible array being allocated need not be in contiguous locations. However, each hyperslab that is allocated will be in contiguous locations. The starting address of such a contiguous sequence of locations is what is stored in s_j. The values $v\langle 2\rangle, v\langle 3\rangle, ..., v\langle k+1\rangle$ are the respective multiplicative coefficients and $v\langle 0\rangle$ and $v\langle 1\rangle$ are the starting index and the starting linear address of hyperslab respectively. The value "-1" denotes null entries. The corresponding axial-vectors of Figure 3.3.1 is given in Figure 3.3.2.

To compute the address of a given k-dimensional index $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$, the hyperslab that contains the element needs to be determined first. The hyperslab whose first elements are $\langle i_0, 0, ..., 0\rangle, \langle 0, i_1, 0, ..., 0\rangle ... \langle 0, 0, .., i_{k-1}\rangle$ give the *candidate hyperslabs* that should contain the element whose index is $\langle i_0, i_1, ..., i_{k-1}\rangle$. The element of $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ always belongs to the hyperslab with the **maximum starting address** of the candidate hyperslabs. This is determined by comparing the starting addresses of the corresponding elements of the axial-vectors. Let the vector of records of dimension j be denoted by Γ_j. The starting addresses of the axial-vectors are given by $\Gamma_j\langle i_j\rangle, 0 \leq j < k$. For instance, the linear address of $A\langle 3, 3, 1\rangle$ can be determined as follow;

$$\mathbb{Z}^0_{\mathbb{U}^*_l} \quad = \quad max(\Gamma_0\langle 3\rangle.v\langle 1\rangle, \Gamma_1\langle 3\rangle.v\langle 1\rangle, \Gamma_2\langle 1\rangle.v\langle 1\rangle) \quad = \quad max(30, 18, 0) \quad = \quad 30$$

Thus $\mathbb{Z}^0_{\mathbb{U}^*_l} = 30,\ l = 0$, and $\mathbb{U}^*_l = 3$. The computation $\mathcal{F}^*(\langle 3, 3, 1\rangle) = 30 + 10 * (3 - 3) + 2 * 3 + 1 * 1 = 30 + 0 + 6 + 1 = 37$. The value 37 is the linear address relative to the starting address 0.

When extendible arrays have some structural regularity in their growth pattern, the realization functions may be simplified. The structural regularity may take the form of a:

- Uniform extendible array with linear expansion or
- Uniform extendible array with exponential expansion.

Definition 3.3.4. *Let* $\mathbb{U}^*_j,\ j = 0, 1, ..., k-1$ *denotes the respective instantaneous upper bounds, such that* $0 \leq \mathbb{U}^*_j \leq \mathbb{X}_j$, *where* $\mathbb{X}_j$*'s define the limiting maximum values of the upper bounds* $\mathbb{U}^*_j$, *of the respective dimensions. An array* $A_\theta[\mathbb{U}^*_0][\mathbb{U}^*_1][\mathbb{U}^*_2]...[\mathbb{U}^*_{k-1}]$

is said to be a uniform extendible array of linear expansion, $\iff$ *,* $\forall j$*, for which* $\mathbb{U}_j^* < \mathbb{X}_j$*, the index range is incremented by* θ *values, from* $\mathbb{U}_j^*$ *to* $\mathbb{U}_j^* + \theta$*, at each step.*

Figure A.1.1 in Appendix A.1 illustrates a schematic storage layout for a 3-dimensional uniform extendible array with linear expansion where $\theta = 2$. The extension in the figure is also termed *cyclic interruptible* extension. In cyclic interruptible extension, after each extension of the bound of a dimension D_i, the next expansion occurs on another different dimension D_{i+1} thereby interrupting the expansion on dimension D_i. This happens in a cyclic or round-robin fashion. For uniform extendible array with linear expansion, the mapping function (Equation 3.3.1) becomes:

$$q^* = \mathcal{F}^*(\langle i_0, i_1, i_2, ..., i_{k-1}\rangle) = \mathbb{Z}^0_{\mathbb{U}_l^*} + (i_l - \mathbb{U}_l^*)\frac{\mathbb{Z}^0_{\mathbb{U}_l^*}}{v\langle 0\rangle} + \sum_{\substack{j=0 \\ j\neq l}}^{k-1} i_j C_j^* \quad (3.3.2)$$

where $v\langle 0\rangle$ is the starting index of candidate hyperslap. C_j^* is the same as in Equation 3.3.1

In case of exponential expansion, the increments are by some factor e as illustrated in Figure A.1.2. The mapping function for cyclic exponential expansion is the same as in Equation 3.3.2 with the index range incremented by a factor of e, i.e., from $\mathbb{U}_j^*$ to $\mathbb{U}_j^* \times e$ at each cyclic step.

3.4 Inverse Function of Extendible Arrays

Although inverse mapping function of extendible dense arrays are not frequently used in scientific applications, in certain specific scenarios such as nearest neighbourhood search, it is useful and handy. As part of the contribution of this thesis we present

a function $\mathcal{F}^{*-1}$ similar to the one discussed in Section 3.2. This function can be computed after the array has undergone a series of arbitrary expansions. The principal cost is in identifying which dimension and consequently the axial-vector component of the dimension that contains the relevant coefficients to be applied to compute the k-dimensional indexes. Let q^* be the given linear address. We need to find a dimension l such that an axial-vector component of this dimension has a starting address of a hyperslab that is the *minimum upper-bound* of q^*. If we denote such an axial-vector as $\Gamma_l\langle z_l\rangle.v\langle 1\rangle$, then the axial-vector components can be interpreted as follows:

l: is the dimension where a vector component was found with the *minimum upper-bound* for q^*, i.e., $q^* \leq \Gamma_l\langle z_l\rangle.v\langle 1\rangle$,

z_l: The index of the axial-vector where this occurred,

$\Gamma_l\langle z_l\rangle.v\langle 1\rangle$: The actual *minimum upper-bound* found.

From the above definitions, we have the algorithm for an inverse calculation function $\mathcal{F}^{*-1}$ as Algorithm 2. Note that the elements stored in the entries $2, 3, \ldots, k+1$ of vector v of $\Gamma_l\langle z_l\rangle.v[]$, are the multiplying coefficients used in computing q^*.

To illustrate Algorithm 2, we employ the extendible array used in Section 3.3 (see Figures 3.3.1 and 3.3.2). Suppose we wish to find the 3-dimensional index of the linear address 27 (i.e.,$q^* = 27$), then the axial-vector of the hyperslab that satisfy $q^* \leq \Gamma_l\langle z_l\rangle.v\langle 1\rangle$ is $\langle 3; 30; 10, 2, 1\rangle$. Q is initialized to q^* and the initial correspoding multiplying coefficients are $C_0 = 10$, $C_1 = 2$ and $C_2 = 1$.

We update Q as $Q = 27 - 18 = 9$. The number 18 corresponds to $\Gamma_l\langle i_l\rangle.v\langle 1\rangle$ which

Algorithm 2: Computing the k-dimensional indices, given the linear address q^*

input : q^*, the linear address of the k-dimensional index
output : $\langle i_0, i_1, i_2, ..., i_{k-1} \rangle$, the k-dimensional indexes
begin

```
Determine the dimension l and the component axial-vector Γ_l⟨z_l⟩.v[], with the
  minimum upper bound on q* ;
Q ← q* ;
for j ← k − 1 down to 0 do
    C_j ← Γ_l⟨z_l⟩.v⟨j + 2⟩ ;
Q ← Q − Γ_l⟨i_l⟩.v⟨1⟩ ;
for j ← k − 1 down to 1 do
    if j ≠ l then
        i_j ← Q  mod C_{j−1} ;
        Q ← Q/C_{j−1} ;
    for r ← 0 to j-2 do
        C_r ← C_r/C_{j−1} ;
    if l > 0 then
        C_l ← C_l/C_{j−1} ;
i_0 ← Q  mod C_1 ;
i_l ← Q/C_0 ;
```

is the starting address of the hyperslap that contains the linear address. Within the second `for` loop, we start with $j = 2$ and $l = 0$, hence $i_2 = 9 \mod 2 = 1$ and Q is updated as $Q = 9/2 = 4$. C_r becomes $C_0 = 10/2 = 5$. In the second iteration, $j = 1$ and $l = 0$, hence $i_1 = 4 \mod 5 = 4$ and Q is updated as $Q = 4/5 = 1$. Quotients of fractions in the algorithm is regarded as 1. The loop is existed and $i_0 = 1 \mod 2 = 1$. The index value, i_l, which in this case is also i_0, becomes $i_0 = 1/5 = 1$. Hence $i_0 = 1$, $i_1 = 4$ and $i_2 = 1$.

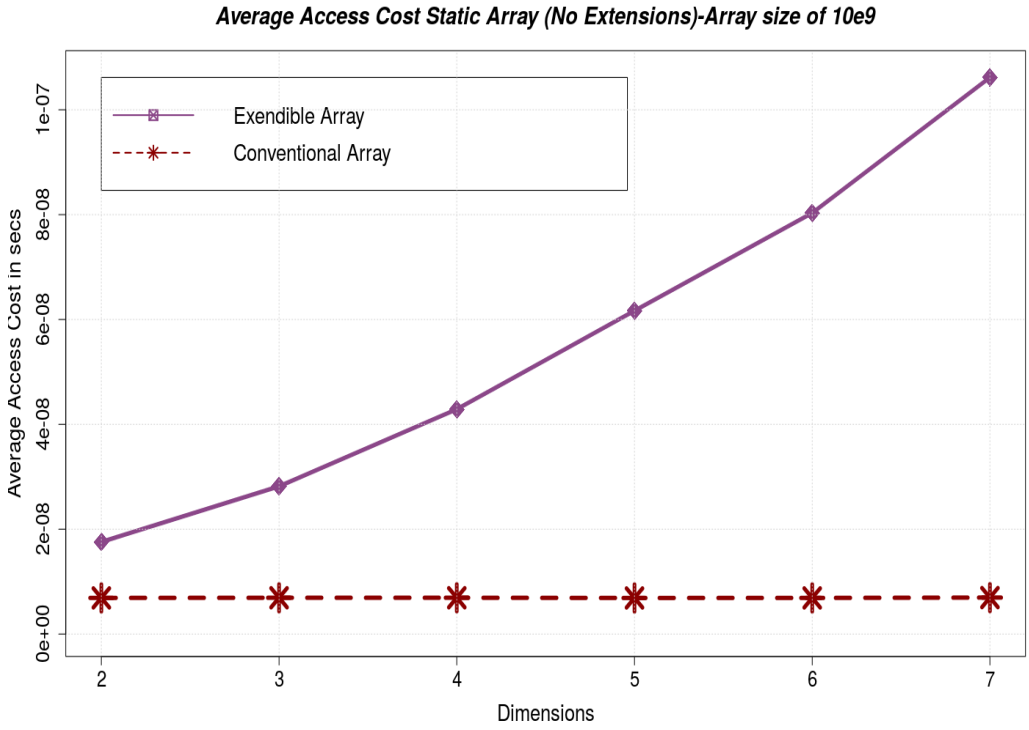


Figure 3.5.1: Average Access Cost without Extensions (in Memory)

3.5 Performance Analysis

To ascertain the performance of our extendible array compute access scheme, we conducted performance experiments. These experiments were conducted on a 64-bit Intel Core 2 Quad Q9650 processor with L2 Cache size of 2×6 MB and a frequency of 3 GHz as well as 12GB of RAM running Ubuntu Linux 12.04 LTS. We used GCC 4.6.3 compiler for the experiments enabling the following option flags; `-O2 -Wall -Wextra -ftime-report`. All array elements are double floating point numbers. The expected times for random accesses of array elements for memory resident extendible arrays (with array size of 10^9) when there are no expansions (what we called *Static Arrays*) were compared with conventional arrays for $k = 2, 3, ..., 7$.

The total cost for random element accesses, when there are random requests to extend the index range of a dimension were compared with similar operations on a conventional array, for $k = 2, 3, 4$. For each k, we performed 1000 random accesses.

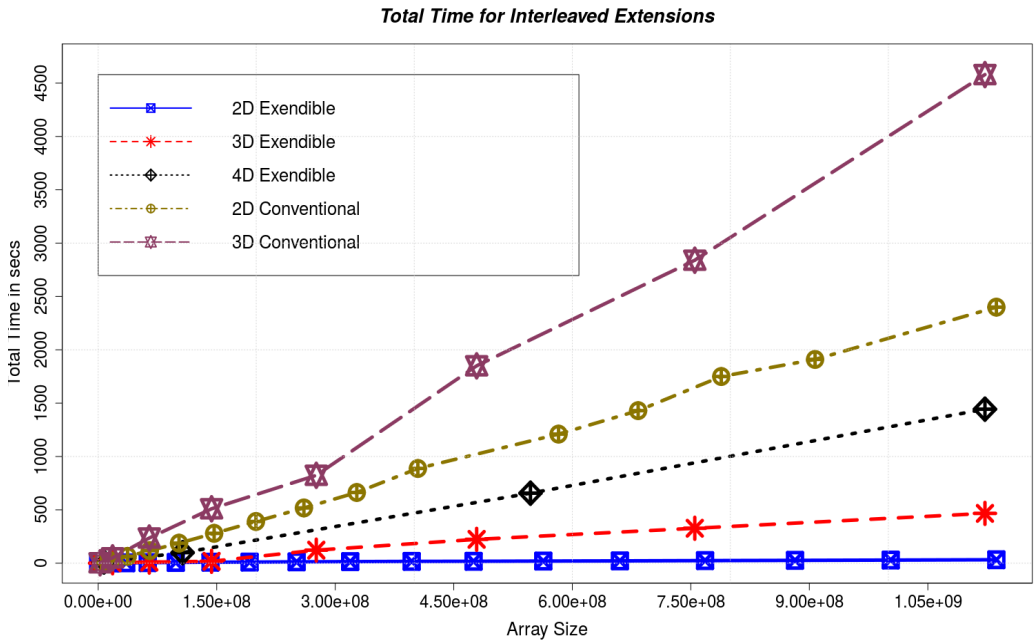


Figure 3.5.2: Total Access Cost for Interleaved Extensions in Memory

Figure 3.5.1 shows a comparison of the average access cost of an array element of our in-memory extendible array and the average access cost for the conventional array for static arrays. The graph shows that the extendible array access function varies significantly with the dimension or rank of the array. This dependency on dimension k is due to the fact that we perform k-binary search on the axial-vectors for access operations. The conventional array in this case performs better because of the traversals within axial-vectors even when there are no expansions. But with interleaved extensions as illustrated in Figure 3.5.2, the extendible array performs

considerably better than the conventional array. The high average access cost of conventional array is the result of the reorganisation of array elements each time an arbitrary dimension is extended.

4

Chunked Extendible Multidimensional Array Files

This chapter gives a description of the chunking technique for extendible array files

Contents

Although scientific application and simulation algorithms processing sessions occur in memory, the datasets these algorithms operate on reside on secondary storage. I/O has been a limiting factor for extreme scale computing particularly when there are substantial growth in the amount of data produced by these applications. In this chapter we address this challenge for extendible dense arrays that reside on secondary storage. The primary contributions of this thesis is presented in this chapter. In Section 4.1, we illustrate how allocating data elements in chunks on disk can improve I/O performance. In Section 4.2 we introduce a new data structure, the O_2-Tree [11], for indexing the axial-vectors to reduce the search-time for an array element. In Section 4.3 we conclude this chapter by providing analytical evaluation and the theoretical foundation of our allocation scheme.

4.1 Chunked Extendible Arrays

The use of the axial-vector can be expensive and depends particularly on the interruptible expansions. The number of the axial-vectors per dimension are maximum when we have *cubical extensions* (see Lemma 4.3.1). A perfect cubical extension is the scenario where for a given initial extendible array $A_\theta[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ where $\mathbb{U}_j^* = 1$ $(0 \leq j \leq k-1)$ (see Definition 3.3.4), the array undergoes cyclic interruptible linear expansion with $\theta = 1$. The explanation of linear expansion and cyclic interruptible has been given in Section 3.3. Generally, in interruptible extension/expansion, after each extension of the bound of a dimension s, the next expansion occurs on another different dimension t thereby interrupting the expansion on dimension s. Such interruptible expansion causes the addition of a new entry in the axial-vectors

each time it occurs.

Chunking the array gives some additional advantages. Firstly, it gives contiguous storage allocations for the elements of the chunks. Secondly, when arrays are allocated onto secondary storage, I/O can be made in multiples of the chunk size. Hence we introduce an approach of *chunking* the extendible array. The allocation is done in *chunks* as opposed to the single index extension described in Section 3.3. A chunk of an array, Q is denoted as $Q[\chi_0][\chi_1][\chi_2]...[\chi_{k-1}]$ where χ_j is the size, in number of indices, of dimension j. The first task in our approach is to determine the chunk-size of each dimension. In this approach, we allocate enough chunks just to contain the initial allocated array. In Figure 4.1.1, we illustrate our chunking approach with a 2-dimensional extendible array. For a chunk size of 3×3, an initial array, $A[4][5]$ will need 2 chunks on dimension D_0 and D_1 respectively. Similarly, for a chunk size of $3 \times 3 \times 3$ in Figure 4.1.2, an initial array, $A[4][4][2]$ will need 2 chunk sizes on dimension D_0 and D_1 respectively and 1 on D_2.

Given a k-dimensional extendible array $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$, and a chunked block $Q[\chi_0][\chi_1][\chi_2]...[\chi_{k-1}]$, the number of chunk indices, ρ_i for a given dimension i, is given by:

$$\rho_i = \left\lceil \frac{\mathbb{U}_i^*}{\chi_i} \right\rceil \tag{4.1.1}$$

The allocation of chunks, denoted by A_c, becomes $A_c[\rho_0][\rho_1][\rho_2]...[\rho_{k-1}]$.

The entry into the respective axial-vector is based on A_c. In Figure 4.1.1, the initial axial-vector for D_0 is $\langle 0; 0; 2, 1; S_0 \rangle$ because it is based on $A_c[2][2]$ instead of $A[4][5]$. Similarly, in Figure 4.1.2, the initial axial-vector for D_0 is $\langle 0; 0; 2, 1, 1; S_0 \rangle$ because it is based on $A_c[2][2][1]$ instead of $A[4][4][2]$. If the range of index values of a given

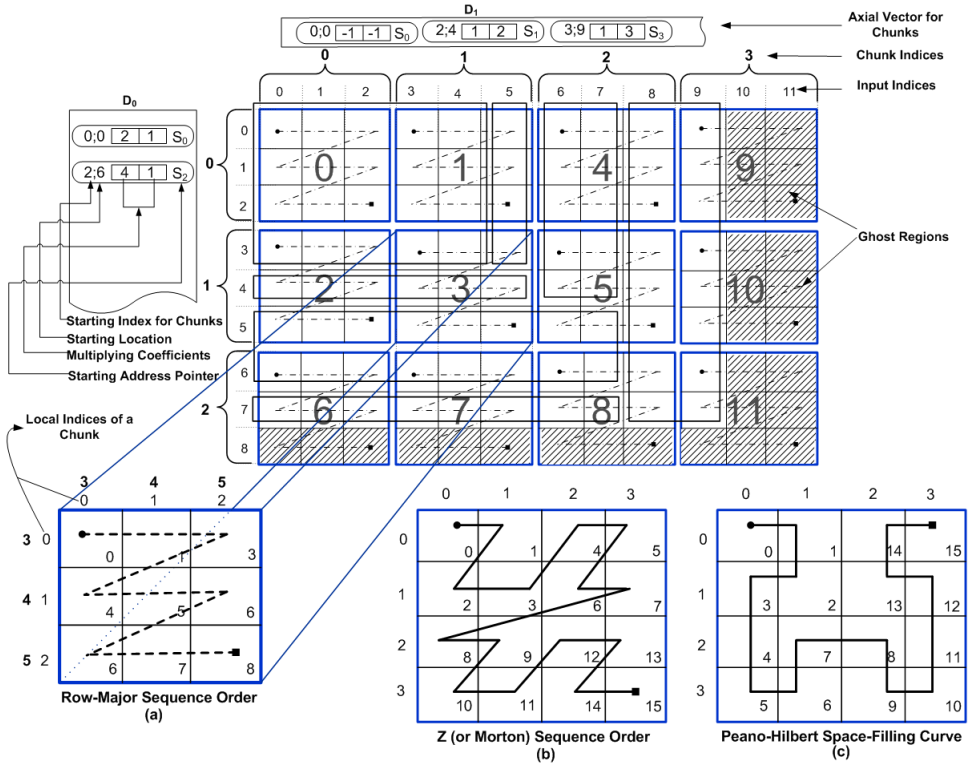


Figure 4.1.1: Chunking Extendible Dense Array-2D

dimension l of the array $A[\mathbb{U}_0^*][\mathbb{U}_1^*]\ [\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ is extended by say λ_l, then the bound of the index range, increases from $\mathbb{U}_l^*$ to $\mathbb{U}_l^* + \lambda_l$. An entry is made to the requisite axial-vector only if the condition

$$[\mathbb{U}_l^* + \lambda_l] > [\rho_l \times \chi_l] \tag{4.1.2}$$

is met. As long as the inequality 4.1.2 is not satisfied, it implies that there is enough gray region (*ghost region*) to contain λ_l (i.e. $[\rho_l \times \chi_l] \geq [\mathbb{U}_l^* + \lambda_l]$). On the other hand, if the condition 4.1.2 is satisfied, then there is a need for allocation of more

chunks. The number of additional chunks ρ_a to be allocated is given by:

$$\rho_a = \left\lceil \frac{[\mathbb{U}_l^* + \lambda_l] - [\rho_l \times \chi_l]}{\chi_l} \right\rceil \tag{4.1.3}$$

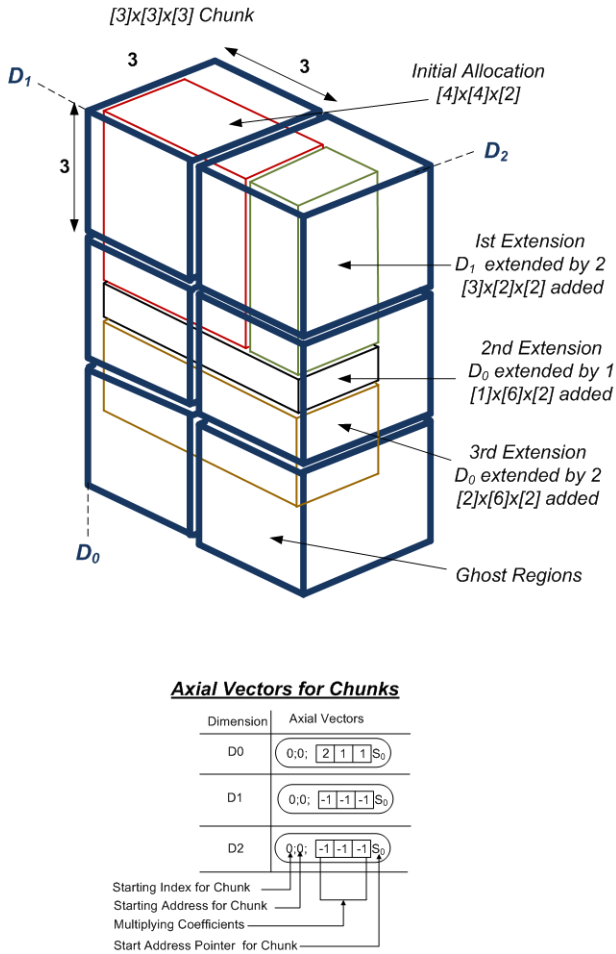


Figure 4.1.2: Chunking Extendible Dense Array-3D

Increasing the index bound on D_1 in Figure 4.1.1 by 1 (i.e. from $\mathbb{U}_l^* = 5$ to $\mathbb{U}_l^* = 6$), will not satisfy the inequality 4.1.2 since $[\rho_l \times \chi_l] \geq [\mathbb{U}_l^* + \lambda_l]$. However, with additional expansion of $\lambda_l = 2$ (i.e. from $\mathbb{U}_l^* = 6$ to $\mathbb{U}_l^* = 8$), $[\rho_l \times \chi_l] \ngeq [\mathbb{U}_l^* + \lambda_l]$. The number of chunks to be allocated on D_1 is $\lceil([6+2] - [2 \times 3])/3\rceil = 1$. The entry $\langle 2; 4; 1, 2; S_1 \rangle$

is made on the axial-vector for D_1. The initial and chunk array bounds are updated as $A[5][8]$ and $A_c[2][3]$ respectively.

Accessing an element given its k-dimensional indices requires two simple translations of indices and two mapping schemes. To access an array element $A\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$, the *input indices* $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ are translated into *chunk indices* $\langle j_0, j_1, j_2, ..., j_{k-1}\rangle$ where

$$j_i = \left\lfloor \frac{i_i}{\chi_i} \right\rfloor \tag{4.1.4}$$

Once $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ has been translated to chunk indices $\langle j_0, j_1, j_2, ..., j_{k-1}\rangle$, the starting address, q_c^* of the chunk containing $A\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ can be found by using the following:

$$q_c^* = \mathcal{F}^*(\langle j_0, j_1, j_2, ..., j_{k-1}\rangle) = \mathbb{Z}_{\rho_l}^0 + (j_l - \rho_l)C_l^* + \sum_{\substack{m=0 \\ m\neq l}}^{k-1} j_m C_m^* \tag{4.1.5}$$

where $$C_l^* = \prod_{\substack{m=0 \\ m\neq l}}^{k-1} \rho_m$$

and $$C_m^* = \prod_{\substack{r=m+1 \\ r\neq l}}^{k-1} \rho_r$$

The mapping function, $\mathcal{F}^*$ in Equation 4.1.5 is based on $A_c[\rho_0][\rho_1][\rho_2]...[\rho_{k-1}]$ instead of $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$. $\mathbb{Z}_{\rho_l}^0$ denotes the maximum starting address of candidate chunks, ρ_l is the bound of the chunk index range just before the extension on dimension l and j_l denotes the translated chunk index corresponding to dimension l. To compute the address of $A\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ within the *local chunk*, the input indices $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ need to be translated to local indices as

$\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)} \rangle$ within the chunk by taking simple modulus. Thus $i_{cm} = (i_m \mod \chi_m)$. With the base address of the local chunk as q_c^* and the local chunk indices as $\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)} \rangle$, the address of $A\langle i_0, i_1, i_2, ..., i_{k-1} \rangle$ is only a displacement within the chunk. This can be done by using a row-major sequence order (or column-major) as shown Figure 4.1.1(a). If the chunk size is 2^n where $n \geq 2$, then the Z-sequence order (also called the Morton sequence order) or Peano-Hilbert scan order can be used as illustrated in Figures 4.1.1(b) and 4.1.1(c).

4.2 Axial Vectors as Memory Resident O_2-Tree

Searching through the axial-vectors in the implementation of the memory resident extendible array in Section 3.5 was done by a simple binary search. Alternatively an interpolation search could be used. A new approach to maintaining these axial-vectors in memory is with the use of O_2-Tree [11]. An O_2-Tree is an augmented Red-Black Tree with data records stored only at the leaf nodes. We store the axial vectors of expansions as records of the leaf nodes. For chunked extendible dense array files, we maintain a metadata file F_m that corresponds to the leaf nodes of the O_2-Tree. These records in F_m will be used to reconstruct the memory resident O_2-Tree. Every expansion of a dimension results in an update of corresponding O_2-Tree and consequently the metadata file as well. If an application has already constructed the memory-resident O_2-Tree from the metadata file, then the linear address of an element of the corresponding array file, given its k-dimensional coordinates, will be computed using the same mapping function $\mathcal{F}^*()$, presented in Section 4.1. As illustrated in Figure 4.2.1, the O_2-Tree stores complete records only at the leaf nodes,

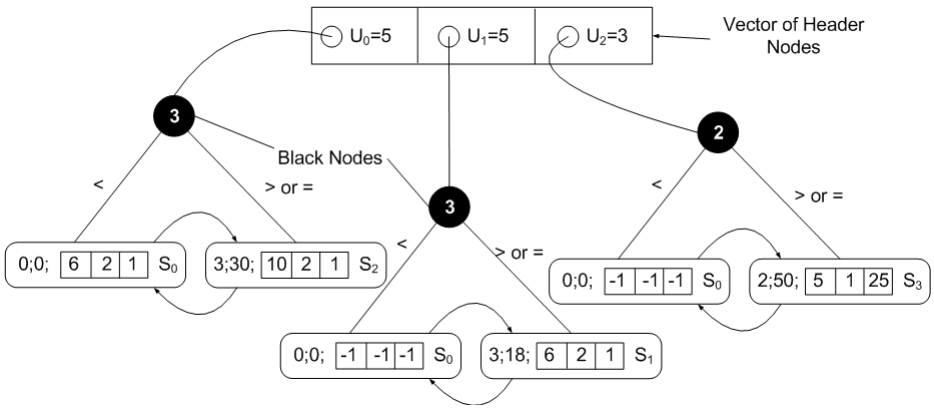


Figure 4.2.1: The O_2-Tree equivalent of axial-vector of Figure 3.3.2

a feature that differentiates it from the Red-Black tree [32]. The internal nodes only store separator keys. All searches terminate at the leaf-node. At this terminal leaf node, the requisite axial vector is identified by performing sequential or binary search. During traversal of the tree, on arrival at an internal node, a left branch is taken if the search key is less than the key at that node; otherwise a right branch is taken. Every other property of the Red-Black tree is preserved through standard left and right rotations.

The internal nodes of the O_2-Tree form a simple binary search tree that is balanced using the Red-Black Tree algorithm. It has the advantage of being easily reconstructed in memory by reading the starting index of each axial-vector of the leaf node. The construction of the O_2-Tree in memory can use either Top-Down or Bottom-Up insertion operations of the Red-Black Tree. The difference between these two operations is in the manner of the rebalancing during update operations. In our implementations, we employ the Top-Down approach as illustrated in Algorithm 3. In the Top-Down approach, rebalancing of the tree is done while the tree is been traversed. This ensures that the tree remains balanced at each insertion or *put*

operation. To insert an axial vector v, the bounding leaf node is located using Red-Black Tree traversal and the key-value pair $\langle v\langle 0\rangle, v\rangle$.

Algorithm 3: Insertion($Key\ v\langle 0\rangle$, $value\ v$, $Header\ H$) : Top-Down

input : $key\ v\langle 0\rangle,\ value\ v$
output : $true$ for success $false$ otherwise

```
begin
    node ⟵ root(H);
    while node.nodeType ≠ leaf do
        if v⟨0⟩ < node.key then
            node ← node.left;
        else
            node ← node.right;
    if node.left.color = Red and node.right.color = Red then
        /* use standard Red-Black Tree top down algorithms */;
        preBalance(v⟨0⟩, node) ;
    if !leaf.isfull() then
        return insertInOrder(v⟨0⟩, v, node);
    else
        return splitInsert(v⟨0⟩, v, node) /* Algorithm 4 */ ;
    return done;
```

The height of the O_2-Tree can be considerably reduced by keeping multiple axial-vectors at a leaf-node. The maximum number of axial vectors that can be contained in a leaf node is usually refereed to as *bucket size*. If the number of axial vectors in the leaf is less than the bucket size of the O_2-Tree, then the new value-pair is inserted in-order into the bucket. If on the other hand, the leaf is full, then a split is performed using Algorithm 4. After the split, a new internal node is inserted

Algorithm 4: splitInsert($Key\ v\langle 0\rangle$, $value\ v$, $O2node$ node)

input : $Key\ v\langle 0\rangle,\ value\ v, O2node\ node$
output : $true$ for success $false$ otherwise

begin

$newLeaf \longleftarrow new\,leaf();$
$newNode \longleftarrow new\,internalNode();$
$midpoint \longleftarrow \frac{m}{2}$;
where m is the order of the tree;
$j \longleftarrow 0$;
for $i \leftarrow midpoint$ **to** $m - 1$ **do**
    $newLeaf[j] \longleftarrow leaf.remove(i);$
    $j++$;
insert "key, value" into the appropriate leaf ;
$newNode.key \longleftarrow newLeaf.key[0];$
$newLeaf.parent \longleftarrow newNode$;
$leaf.parent \longleftarrow newNode$;
reset forward and backward links of leaf nodes;

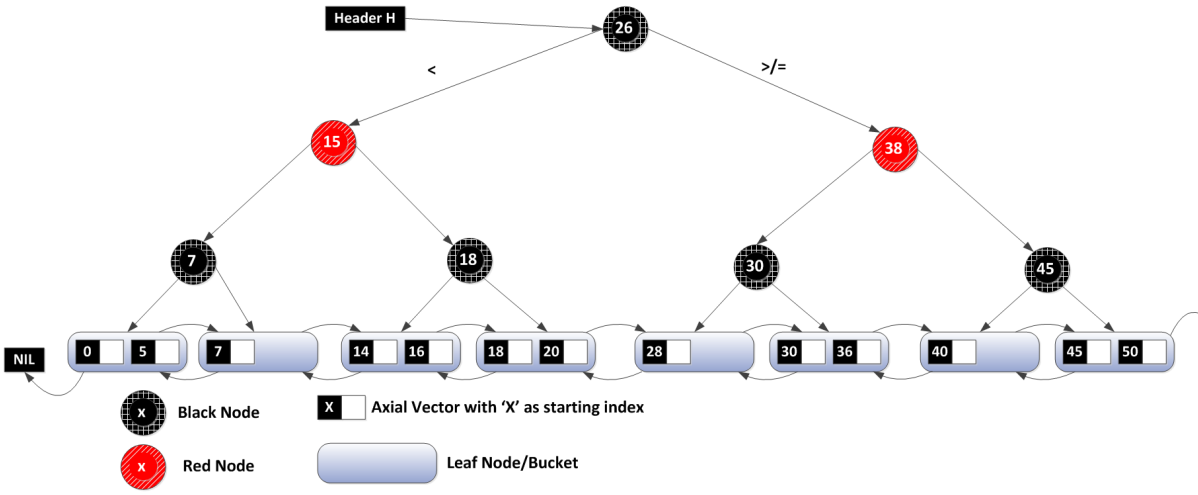


Figure 4.2.2: General Structure of the O_2-Tree

and becomes the *parent* of the two new leaf nodes. The choice of the bucket size of an O_2-Tree can impact the performance of search operations as well as update

operations. Generally there is performance drop for very large bucket size. As the bucket size increases, the height of the tree get reduced, resulting in fewer splits and fewer internal nodes. However the number of data comparisons within the leaf nodes increases.

Algorithm 5: Computation of an element location given its k-dimensional indices

input : $\langle i_0, i_1, i_2, ..., i_{k-1} \rangle$, k-dimensional index
output : q^*,the linear address of the k-dimensional index
begin
    Initialise:
    $I \leftarrow 0$;
    $j_i \leftarrow i_i$;
    $V \leftarrow O2TreeSearch(tree[I], j_0)$;
    $S_i \leftarrow V.startIndex$;
    $A \leftarrow V$;
    for $i \leftarrow 1$ **to** $k-1$ **do**
        $V \leftarrow O2TreeSearch(tree[i], j_i)$;
        if $S_i < V.startIndex$ **then**
            $A \leftarrow V$;
            $S_i \leftarrow V.startIndex$;
            $I \leftarrow i$;
    for $i \leftarrow 0$ **to** $k-1$ **do**
        if $i \leftarrow I$ **then**
            $prod \leftarrow prod + (j_i - A.key) * C_l^*$;
        else
            $prod \leftarrow prod + j_i * C_m^*$;
    $base \leftarrow A.startLoc + prod * chunked_{size}$;
    $i_{ci} \leftarrow i_i$;
    return $base + ChunkDisp(i_{c0}, i_{c1}, ..., i_{ck-1})$

The general structure of our index scheme is illustrated in Figure 4.2.2. The leaf-nodes are double linked to provide easy traversal during key range searches. For our chunking approach, we maintain the *chunked axial-vector* at the leaf node. The search for the location of an element given its k-dimensional indices can be obtained by using Algorithm 5. $ChunkDisp()$ can be implemented as in row-major order

or column-major order. It can also be implemented using Peano-Hilbert scan or Z-sequence order if the dimensional chunk-size is 2^n, where $n \geq 2$. $O2TreeSearch()$ takes as input, an O_2-Tree and a key and returns an axial-vector. $O2TreeSearch()$ function is given as Algorithm 6.

Algorithm 6: Retrieval of Axial-vector given an O_2-Tree and a key

input : $\langle tree, key \rangle$,$O_2$-Tree and key for searching
output : Axial-Vector
begin
    $node \leftarrow tree.root$;
    while $node \neq leaf$ **do**
        if $key < node.key$ **then**
            $node \leftarrow node.left$;
        else
            $node \leftarrow node.right$;
    return $node.axialVector$

4.3 Analytical Evaluation

We present some analytical results on properties of the extendible array realisation. We present the analysis for a memory resident extendible array first. Consider the process of extending a dimension j of an array bound $\mathbb{U}_j$. When an extension is made for this dimension, an axial-vector component, say $\Gamma_j\langle z_j \rangle$ is created. Denote the fields in $\Gamma_j\langle z_j \rangle$ by

$\Gamma_j\langle z_j \rangle.StrtIdx$**:** Starting index of the hyperslab added for the expansion;

$\Gamma_j\langle z_j \rangle.StrtAddr$**:** Starting integer address of the hyperslab;

$\Gamma_j\langle z_j \rangle.v[]$**:** The vector of stored coefficients;

$\Gamma_j\langle z_j\rangle.S_{z_j}$**:** Pointer to the start of consecutive locations where elements of the adjoined hyperslab are stored.

If subsequent extensions are on the bound of dimension j, no axial-vector is inserted. In other words, if extensions on dimension j is *uninterrupted*, extensions continue with no additional axial-vector. Let $\mathcal{E}_j$ be the number of *uninterrupted* extensions, then we have the following theorem.

Lemma 4.3.1. *The number of the axial-vectors per dimension are maximum in cyclic interruptible uniform extensions.*

Proof. After every extension in a cyclic interruptible uniform extension (also called cubical extension), the array is checked and further padded such that constraint $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$ is maintained. Suppose we have a $k-$dimensional unit extendible array, i.e., $\mathbb{U}_j^* = 1$ where $0 \leq j < k-1$, then an instantiation of the initial array will result in k axial vectors. In normal extension, if dimension l is increased by some λ, then an additional axial vector is added; hence the total number of axial vectors becomes $T_n = k+1$. In cubical extension however, an addition of $k-1$ will be added (due to padding on other dimensions); hence the total number of axial vectors becomes $T_c = 2k-1$. As the number of dimensions, k increases, $T_c \gg T_n$. □

Theorem 4.3.1. *Given that in a k-dimensional extendible array, dimension j undergoes $\mathcal{E}_j$ uninterrupted expansions, such that the corresponding axial-vector of dimension j has $\mathcal{E}_j$ entries, then if $\mathcal{E} = \sum_{j=0}^{k-1} \mathcal{E}_j$, the complexity of computing the function $\mathcal{F}^*()$ using the axial-vector supporting data structure is $O(k + k\log(\mathcal{E}) - k\log k)$ with $O(k\mathcal{E})$ addition space.*

Proof. To compute $\mathcal{F}^*(\langle i_0, i_1, \ldots, i_{k-1}\rangle)$, we first determine the dimension j and the index z_j that has the maximum of the value for $\Gamma_j\langle z_j\rangle.StrtAddr$. This takes $\sum_{r=0}^{k-1} \log \mathcal{E}_r$ using binary search on each axial-vector. Once the correct entry is found, the computation follows from using $k+1$ additions and $k-1$ multiplications. The computation time is then $O(k + \sum_{r=0}^{k-1} \log \mathcal{E}_r)$. Setting $\mathcal{E}_j = \mathcal{E}/k$, we get that the time to compute $\mathcal{F}^*()$ is $O(k + k\log(\mathcal{E}) - k\log k)$. The additional space to store the axial-vectors is $(K+3) \times \mathcal{E}$ which is $O(k\mathcal{E})$. □

Corollary 4.3.1. *Given a k-dimensional dense extendible array of N elements that is realised using the axial-vector representation. Then N elements can be stored in space $O(k^2 N^{1/k})$, with an element access function computable in $O(k + \log N)$.*

Proof. This follows from Theorem 4.3.1 once we realise that the maximum number of *uninterrupted* expansion occurs if, after every single extension of an array bound, it is immediately interrupted. In this case we generate a cubical array with $\mathcal{E}_j = N^{1/k}$. Substituting this in the result of the Theorem 4.3.1, the corollary follows. □

Theorem 4.3.2. *Given a linear address q^* of an element in an N-element extendible array realised with the aid of k axial-vectors, the k-dimensional index of an element is computable by the function $\mathcal{F}_*^{-1}$ in time $O(k + \log N)$.*

Proof. The major step in computing the inverse function is in identifying the dimension and the axial-vector component of the dimension that has the minimum upper bound on q^*. Once this is found, the computation of the indices, is done by repeated application of k remainder and division operations. Identifying the dimension and

the axial-vector with the minimum upper bound on q^* requires binary searches where the worst case is $O(k \log N^{1/k})$ giving a total time complexity of $O(k + \log N)$. □

The O_2-Tree representation of the axial-vectors simply allows one to make a binary tree traversal when searching for the correct axial-vector component instead of a binary search. Consequently, the access time complexity and storage overhead hold for both the axial-vector representation and the representation of the axial-vectors using the O_2-Tree.

Theorem 4.3.3. *Given an N-element k-dimensional extendible dense array $A[\mathbb{U}_0][\mathbb{U}_1]\ldots[\mathbb{U}_{k-1}]$ that is allocated in chunks of size $\chi_0, \chi_1, \ldots \chi_{k-1}$. Then the complexity of accessing an element $A\langle i_0, i_1, \ldots i_{k-1}\rangle$ is $O(k + \log n)$ where $n = N/\prod_{j=0}^{k-1} \chi_j$.*

Proof. In the case of chunked extendible array, accessing elements of an array occurs at two levels. The first level computes the address of the chunk. The second level computes the linear address as a displacement within the chunk, from the first element position of the chunk. Since elements within the chunk are allocated in a linear address location using a *row-major* mapping function, the time to identify the location of an element with a chunk is $O(k)$. The first level organisation is a straightforward O_2-Tree representation of the axial-vectors of appended chunks of the array. This is equivalent to managing an extendible array of size $n = N/\prod_{j=0}^{k-1} \chi_j$ with an access cost of $O(k + \log n)$ for the chunk. The result follows from the fact that $O(k) + O(k + \log n)$ gives $O(k + \log n)$ access cost. □

The main results of extendible array realisations are summarised in Table 4.1 where N is the number of elements, and a chunk size is defined as $\prod_{j=0}^{k-1} \chi_j$.

Array Method	Linear Address Calculation	k-Dimensional Index Calculation	Extra Storage Overhead
Conventional - No Chunking	$O(k)$	$O(k)$	0
With Axial-Vectors - No Chunking	$O(k + \log N)$	$O(k + \log N)$	$O(k^2(N)^{1/k})$
With O_2-Tree - No Chunking	$O(k + \log N)$	$O(k + \log N)$	$O(k^2(N)^{1/k})$
With O_2-Tree - With Chunking	$O(k + \log n)$	$O(k + \log n)$	$\prod_{j=0}^{k-1} \rho_j \times \prod_{j=0}^{k-1} \chi_j - \prod_{j=0}^{k-1} \mathbb{U}_j^* + O(k^2(n)^{1/k})$

Table 4.1: Summary of features of extendible array realisations, where N = array size, n = # chunks, and χ_j = chunk size of dimension j.

5

Block Allocation Schemes and Access Methods

This chapter illustrates various allocation strategies at the chunk/block level and single elements within blocks

Contents

5.1 Overview of Allocation Schemes for Array Chunks

Multidimensional arrays used in data intensive scientific computations are usually large and are as such stored on secondary and tertiary storage systems. In such computations or processing, various blocks of the arrays are typically retrieved into memory using range indices. In such a scenario, the multidimensional array is decomposed into smaller arrays called *chunks* or blocks. In Chapter 4, we illustrated how chunking provides improved performance by reducing significantly the height of the O_2-Tree. It also provided high storage utilization. However, we did not provide a formal approach of determining the chunk size and shape. The problem of determining optimal chunk size and shape has been addressed in [33, 34]. Arrays chunks are usually stored on disk as I/O blocks.

The process of chunking is parametrized by *chunk size* and *chunk shape*. These parameters are determined by storage system characteristics and pattern of accessing array elements [33, 34]. The chunk size is defined as the number of the number of elements that can be contained in a chunk. The chunk shape on the other hand is the length of the chunk in the direction of each dimension. The optimization goal is to find the optimal chunk size and optimal chunk shape so that the number of accessed disk blocks or I/O blocks per query is minimized. Usually the chunk size is chosen to be the size of the I/O block. The chunk shape is chosen such that it minimizes the average number of accessed blocks. Large chunk size may cause unwanted data to be read for a query while small chunk size may require more accesses to retrieve all chunks needed for a query.

Consider a 3-dimensional extendible array of shape $\langle 2, 10, 12\rangle$ and a query range

that retrieves a chunk shape of $\langle 1, 5, 4 \rangle$. In Figure 5.1.1, the array is organised in row-major order. When the query range $\langle 1, 5, 4 \rangle$ is applied, it retrieves a block of size 60 instead of 20. Similarly, if the array is organised in column-major order as in Figure 5.1.2, it also retrieves a block of 60 instead of 20, for the same query. However if the array is chunked using a chunk shape of $\langle 1, 5, 4 \rangle$, then the query request, $\langle 1, 5, 4 \rangle$, retrieves exactly one block of size 20 (see Figure 5.1.3).

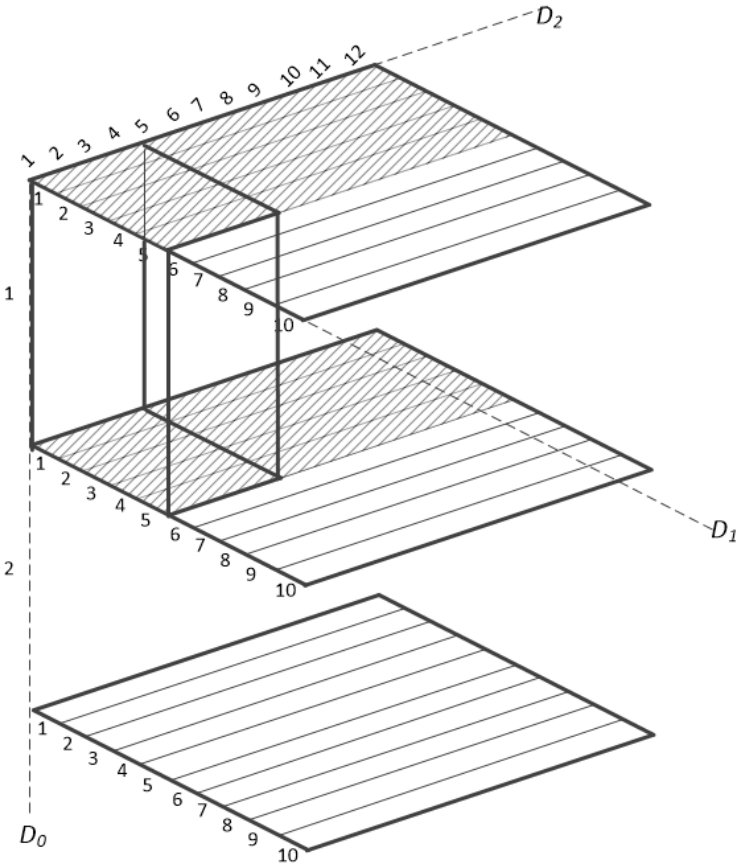


Figure 5.1.1: Row-Major Organisation

The optimal chunk shape depends on the query shape. For instance if the query range range was $\langle 2, 5, 2 \rangle$ instead of $\langle 1, 5, 4 \rangle$, the optimal shape would have been $\langle 2, 5, 2 \rangle$. This implies that to obtain an optimal shape and size, some information

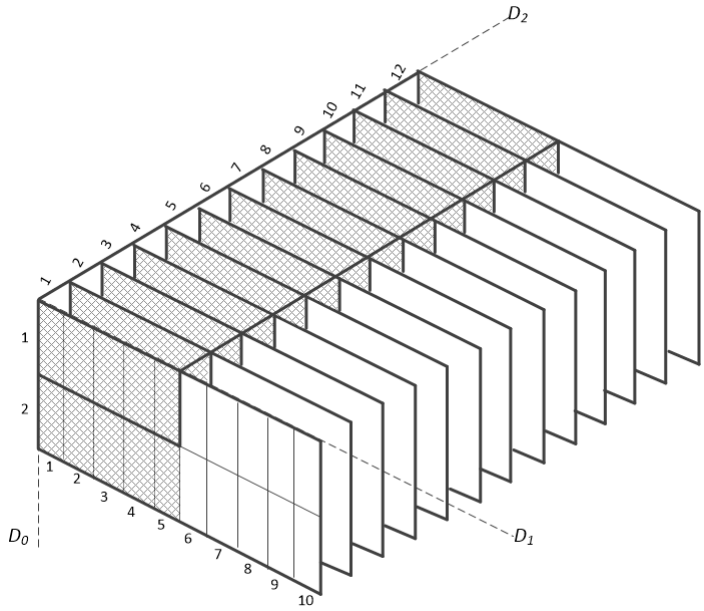


Figure 5.1.2: Column-Major Organisation

about expected pattern of access is required. This information is usually provided by system users or by statistically sampling real queries from access log history.

Suppose an initial k-dimensional extendible array block $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$ is partitioned such that $\mathbb{U}_j^*$ is split into χ_i intervals for $1 \leq i \leq k$. The chunk shape is given by $\langle \rho_1, \rho_2, ..., \rho_k \rangle$ where $\rho_i = \left\lceil \frac{\mathbb{U}_i^*}{\chi_i} \right\rceil$ (see Equation 4.1.1 in Chapter 4). ρ_i is the length of the ith axis of the multidimensional chunk. We usually store A on disk subject to the constraint that each disk block can hold C elements of A. This implies that given a chunk shape $\langle \rho_1, \rho_2, ..., \rho_k \rangle$, the chunk size given by $\prod_{i=1}^{k} \rho_i$ should satisfy the condition:

$$C \geq \prod_{i=1}^{k} \rho_i \tag{5.1.1}$$

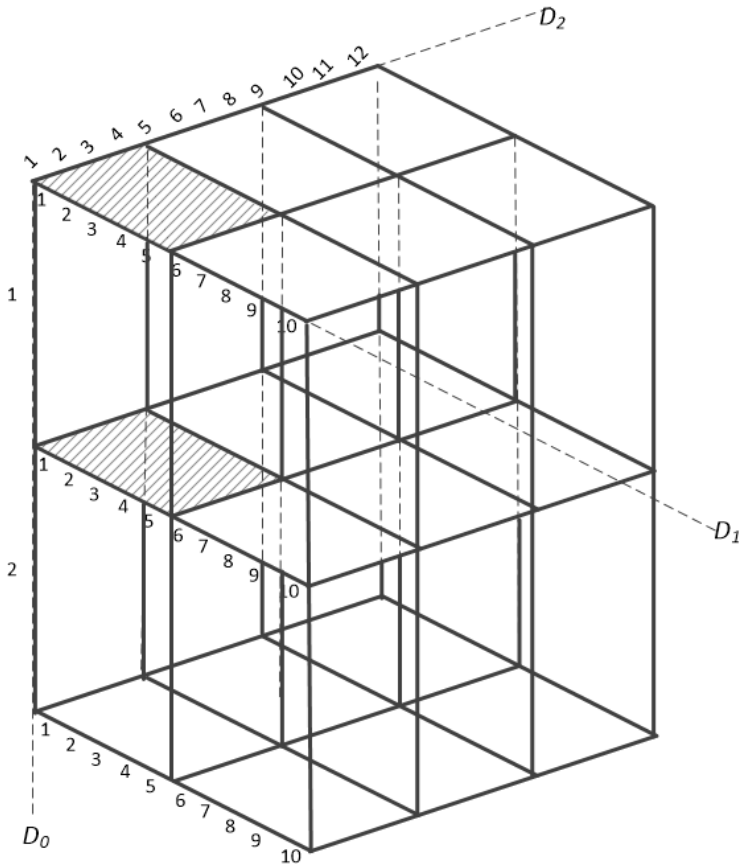


Figure 5.1.3: Chunking

5.2 Access Pattern and Expected Cost of Access

Access pattern is formally defined as the set of range of queries applied to a k-dimensional array A [33, 34]. Each access request is a k-dimensional block located within the array A. The queries are grouped into classes $L_1, L_1, ..., L_n$ such that each class L_i contains all the query blocks of shape $\langle A_{i1}, A_{i2}, ..., A_{ik} \rangle$ located within A. Each class has a weight or probability P_i, which represent the likelihood that a query will result in that class. The access pattern is defined by the set $\{\langle A_{i1}, A_{i2}, ..., A_{ik} \rangle; P_i\}$ such that $\sum_{i=1}^{n} P_i$ and $1 \leq i \leq n$. Given an access pattern set $\{\langle A_{i1}, A_{i2}, ..., A_{ik} \rangle; P_i\}$

and disk block size C, the average cost c_a for a query [34] is given by :

$$c_a = \sum_{j=1}^{n} \left(\prod_{i=1}^{k} \left\lceil \frac{A_{ij}}{\rho_i} \right\rceil \right) P_j \tag{5.2.1}$$

The exact cost c_e for a query is [33] is given by :

$$c_e = \sum_{j=1}^{n} \left(\prod_{i=1}^{k} \left(\frac{A_{ij} - 1}{\rho_i} + 1 \right) \right) P_j \tag{5.2.2}$$

The relative error of computing Equation 5.2.1 is

$$\frac{c_e - c_a}{c_a} \tag{5.2.3}$$

The proof of equation 5.2.2 is given in [33]. For each class $L_1, L_1, ..., L_n$, the cost of query is evaluated using 5.2.2. The optimal chunk shape is the shape that minimizes the cost function.

5.3 Cubical Block Allocation of Extendible Array Files

In uniform expansion or extension (Definition 3.3.4), an extendible array $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$, undergoes linear expansion such that an index range is always increased by some value θ, from $\mathbb{U}_j^*$ to $\mathbb{U}_j^* + \theta$, at each extension step. If $\mathbb{U}_0^* \simeq \mathbb{U}_1^* \simeq ... \simeq \mathbb{U}_{k-1}^*$, then we can pad the array such that $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$. We say that the array A has a *cubical shape*. If already the initial array block to be allocated satisfies condition $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$, then no padding will be required.

Initial cubical padding can only be performed if the conditions $\mathbb{U}_0^* < \mathbb{U}_1^* \mid ... \mid \mathbb{U}_{k-1}^*$ and $\mathbb{U}_{k-1}^* < \mathbb{U}_1^* \mid ... \mid \mathbb{U}_{k-2}^*$ are satisfied for row-major or column-major realizations respectively.

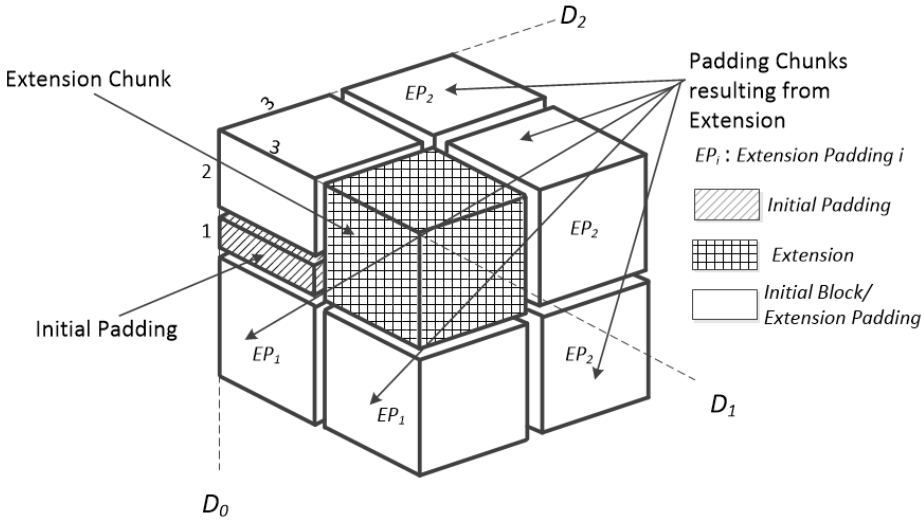


Figure 5.3.1: Cubical Block Allocation for a $3D$ Extendible Array

For scientific applications that prioritise computation speed over storage utilization, we can employ *cyclic interruptible uniform expansion* (simply termed cubical expansion) to significantly reduce the height of the O_2-Tree. In such a case, the optimal chunk shape is given by $\langle \theta_0, \theta_1, ..., \theta_{k-1} \rangle$ where θ is computed such that it minimizes Equation 5.2.2. Before an extendible block or chunk is appended or allocated, we first check whether there are enough ghost regions to subsume the new chunk. This is done by using $[\mathbb{U}_l^* + \lambda_l] > [\rho_l \times \chi_l]$ (The Inequality 4.1.2 as used in Section 4.1, Chapter 4). After every extension, we check for cubical shape (i.e. $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$). If A is non-cubical, we pad the array by appending the chunk shape $\langle \theta_0, \theta_1, ..., \theta_{k-1} \rangle$ in a cyclic fashion until $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$.

Suppose we are to allocate a 3-dimensional array $A[2][3][3]$ (as illustrated in Fig-

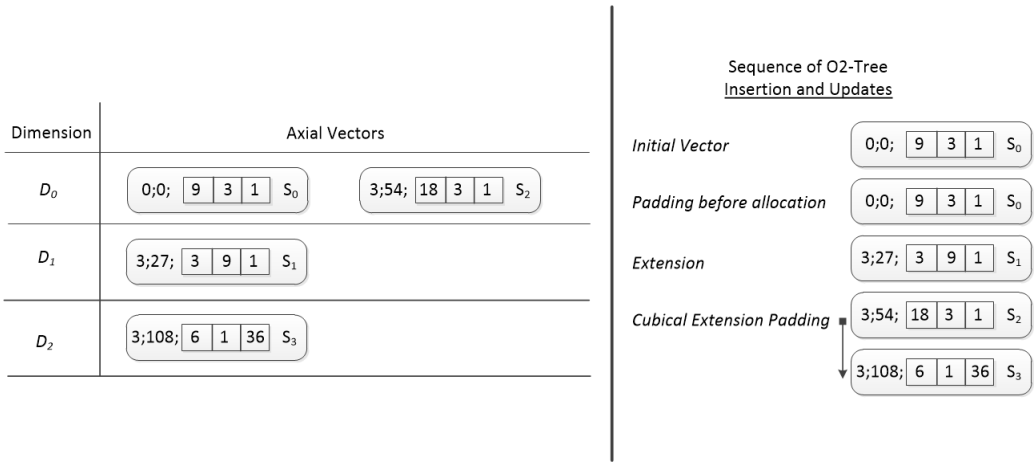


Figure 5.3.2: Sequence of Axial Vector Insertion onto the O_2-Tree

ure 5.3.1, using cubical block allocation, before the allocation, dimension 0 needs to be padded by appending 1-dimensional hyperslab block to the initial array. Let the optimal chunk shape be $\langle 3, 3, 3 \rangle$, then by extending dimension 1 by one chunk block, A becomes $A[3][6][3]$. To ensure A is cubical, we perform *cubical extension*. This is done by cyclically adding chunk blocks to all dimensions except dimension l (in this case $l = 1$) until $\mathbb{U}_0^* = \mathbb{U}_1^* = ... = \mathbb{U}_{k-1}^*$. Hence in Figure 5.3.1 blocks EP_1 are appended before blocks EP_2. The sequence of axial vectors updates and insertions are illustrated in Figure 5.3.2.

5.4 Space Filling Curves Allocation Schemes

A space filling curve is the mechanism of mapping a multidimensional space one-to-one onto one-dimensional space. It reduces the number of dimensions to one, while preserving neighbourhood relations. A space filling curve passes through every point in space, once, giving a one-to-one correspondence between the coordinates of the

points and the one-dimensional sequence numbers of the points on the curve.

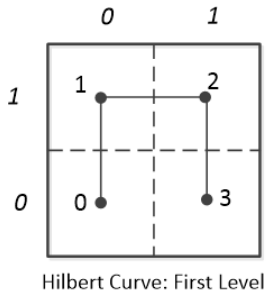
Such mapping of multidimensional datasets onto a one-dimensional space was first suggested by Peano [35] who expressed it in mathematical terms. In [35], the first graphical or geometrical 2-dimensional representation of space-filling curve was presented. Although the first representation was 2-dimensional, it can be extended to multiple dimensions. Other graphical representations include z-order or Lebesgue curve [36], Moores curve, Sierpinski curve [37] etc (see Figure A.2.1 in Appendix A). Of all the space-filling curves, Hilbert curve [35] is the most widely used space-filling curve. This is attributed to its neighbourhood preserving characteristics and superior data clustering properties compared to other curves [35].

To use any space-filling curve in an application, there must exist some mapping function/technique for addressing and querying data elements with the space. In this thesis we employ Hilbert curve as a mapping curve for the individual data elements within chunk blocks. We only demonstrate cases where the chunk size is 2^n, $n \geq 2$. In cases where this condition is not applicable, the cubical padding in section 10.2 can be adopted. Alternatively, the compact Hilbert indices mapping which permits the Hilbert curve to be used for multidimensional arrays with varying dimension cardinality sizes [38] can be used. Once the chunked block $Q[\chi_0][\chi_1][\chi_2]...[\chi_{k-1}]$ where $\chi_i = 2^n$, $n \geq 2$ has been determined, we utilized Hilbert curve mapping allocation structure or technique to represent the individual data elements within the chunk.

5.4.1 Hilbert Curve : Tree Representation

To utilize Hilbert curve within the chunks, an index tree structure representing the curve transversal is constructed [39]. The *order* of the curve d, is determined by $d = \chi_i$. For chunk sizes of 2, 4 and 8 of a $2D$ array, the corresponding highest orders will be 1, 2 and 4 respectively. For k-dimensional multidimensional array, the hyperslab/hyper-rectangle share common faces and the curve passes through 2^{dk} points.

To partition a chunk space using Hilbert curve, a tree structure corresponding to the order of the space is constructed. Any chunk within the multidimensional array having the same dimensional chunk size is pointed to the same tree structure. This minimizes the spaces required for the tree structures. The process of locating an element within a chunk starts by translating the input indices $\langle i_0, i_1, i_2, ..., i_{k-1}\rangle$ into local indices $\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)}\rangle$ within the chunk by taking simple modulus (i.e. $i_{cm} = (i_m \mod \chi_m)$) as in Chapter 4. The tree structure gives the mapping from local indices $\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)}\rangle$ or coordinate point to allocation point (also called *derived key*).



Coordinates	Location Point	Derived Key	n-point
(0,0)	0	00	00
(0,1)	1	01	01
(1,1)	2	10	11
(1,0)	3	11	10

Figure 5.4.1: First Order of Hilbert Curve in $2D$

A *derived key* is the binary sequence number of a location point in the first order (see

Figure 5.4.1) while an n-point is the concatenation of the indices or coordinates of a point, also in binary. Hence in Figure 5.4.1, the derived key of n-point 11 is 10.

With the base address of the local chunk as q_c^* (see Equation 4.1.5) and the local chunk indices, a derived key essentially gives the displacement within the chunk. Thus, it points to an address within the chunk for the data element. Given the translated coordinates $\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)} \rangle$ of a chunk element, the derived key, can be determined by using Algorithm 7.

Algorithm 7: Finding Derived Key or Pointer of chunk element using Hilbert Tree Traversal

---

input : $P\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)} \rangle$: *In Binary*
output : D : *Derived Key Bit String*

begin
  $K \longleftarrow 1$;
  $C_n \longleftarrow root$;
  $D \longleftarrow empty\ string$;
  do
    $p \longleftarrow$ 1 bit taken from each coordinate in P, concatenated into an n-point position K;
    $d \longleftarrow$ the n-bit derived key from C_n corresponding to p;
    append d to D;
    if $K < leaf_l$ **then**
      $C_n \longleftarrow$ node pointed to by $\langle p, d \rangle$ within C_n;
    $K = K + 1$;
  while $K > leaf_l$;
  return D;

For illustrative purposes, suppose we wish to find the displacement of a translated coordinate $\langle 5, 3 \rangle$ within a $2D$ chunked block of order 3, then the binary representation will be $P\langle 101, 011 \rangle$. We pick top bit in each coordinate value of $P\langle 101, 011 \rangle$ to form an n-point "10". The corresponding derived key is 11 (see Appendix A.2, Figure A.2.2);

thus $D\langle \_\ \_\ \_\ \_\ \_\ \_ \rangle$ becomes $D\langle 11\ \_\ \_\ \_\ \_ \rangle$. C_n and $leaf_l$ in algorithm 7 refers to *current node* and *leaf level* respectively. At this instance, $C_l = 1$ and $leaf_l$ is always 3. The pair $\langle 11, 10 \rangle$ points to the last node on the second level. The next n-point concatenation, "01" in $P\langle 101, 011 \rangle$, is performed because $C_l < leaf_l$. The corresponding derived key is "01", hence D becomes $D\langle 1101\ \_\ \_ \rangle$. The pair $\langle 01, 01 \rangle$ points to the second sub-node in the last node of the third level. The final concatenation in $P\langle 101, 011 \rangle$, "11" has "00" as its derived key. Hence the algorithm terminates returning D as $D\langle 110100 \rangle$. This indicates that the element corresponding to translated coordinates $\langle 5, 3 \rangle$ is at position 52 within the contiguous location of the chunked block.

5.4.2 Butz Algorithm

The storage requirement for tree representation diagrams of Section 5.4.1 increases exponentially with the number of dimensions. We employ iterative algorithms such as the Butz algorthim [40] which avoids the storage requirement of tree and state digram structures, as an alternative to row/column-major order within the chunks. This algorithm calculates the coordinates of a point corresponding to an arbitrary ordinal position on the curve. In [41], an inverse algorithm for mapping the coordinates of a point to its derived key was given. We illustrate this algorithm in the context of mapping an element's location within a chunk block of an extendible array. The algorithm is iterative and it requires a number of iterations equal to the order of the curve.

The algorithm uses the several variable definitions and computes the values for those

variables in succession. If k is the number dimensions of the chunked block and m is the order of the Hilbert curve through the block space, then the number of bits (N) for the derived-key is given by $n \times m$. Our desired derived-key r within a block is an N-bit binary expressed as a real number. Given the translated coordinates $\langle i_{c0}, i_{c1}, i_{c2}, ..., i_{c(k-1)} \rangle$ of a chunk element, the algorithm start by by computing α^{η}s for the ηth iteration. α^{η} is a byte of k bits, such that $\alpha_0^{\eta} = i_{c0}^{\eta},\ \alpha_1^{\eta} = i_{c1}^{\eta},\ \alpha_2^{\eta} = i_{c2}^{\eta}, ...,\ \alpha_{k-1}^{\eta} = i_{c(k-1)}^{\eta}$. The remainder of the variables are computed as follows:

1. ω^{η}: This consists of k bits where $\omega^{\eta} = \omega^{\eta-1} \oplus \tilde{\tau}^{\eta-1}$ and $\oplus$ indicates EXCLUSIVE-OR bit operation. For the first iteration, $\omega^0 = 00...0$ and $\tilde{\tau}^0 = 00...0$.

2. $\tilde{\sigma}^{\eta}$: This consists of k bits where $\tilde{\sigma}^{\eta} = \alpha^{\eta} \oplus \omega^{\eta}$. For the first iteration, $\tilde{\sigma}^0 = \alpha^0$

3. σ^{η}: This consists of k bits obtained by shifting $\tilde{\sigma}^{\eta}$ left circular a number of positions equal to $(J_1 - 1) + (J_2 - 1 + ... + (J_{\eta-1} - 1)$. There is no shift for the first iteration.

4. ρ^{η}: This consists of k bits such that $\rho_0^{\eta} = \sigma_0^{\eta},\ \rho_1^{\eta} = \sigma_1^{\eta} \oplus \sigma_0^{\eta},\ \rho_2^{\eta} = \sigma_2^{\eta} \oplus \sigma_1^{\eta}, ..., \rho_{k-1}^{\eta} = \sigma_{k-1}^{\eta} \oplus \sigma_{k-2}^{\eta}$

5. J_{η}: This is an integer between 0 and $k-1$ equal to the subscript of the *principal position* of ρ^{η}. The principal position is the least significant bit position, j, in ρ^{η} such that $\rho_j^{\eta} \neq \rho_{k-1}^{\eta}$. The most significant bit position is considered to be position 0.

6. τ^{η}: This consists of k bits obtained by complementing σ^{η} in the $k-1$ position and then, if and only if the resulting byte is of odd *parity*, complementing in

the principal position. Parity is the number of bits in a byte which are set to 1.

7. $\tilde{\tau}^{\eta}$: This is computed in the same fashion as σ^{η} is derived from $\tilde{\sigma}^{\eta}$.

The $\tilde{\tau}^{\eta}$ in each iteration are concatenated together to form the N-bit binary derived key. These iterative computations come with a little cost in accessing a chunked array element as illustrated in Figure 5.4.2. Further details of the Butz algorithm can found in [40]. We conducted an experiment in Figure 5.4.2 to compare the performance in access operations for Hilbert curve and row-major order traversals within extendible array chunks. We randomly interleaved access operations with extensions for $2D$ and $3D$ extendible arrays. The results indicate that for normal

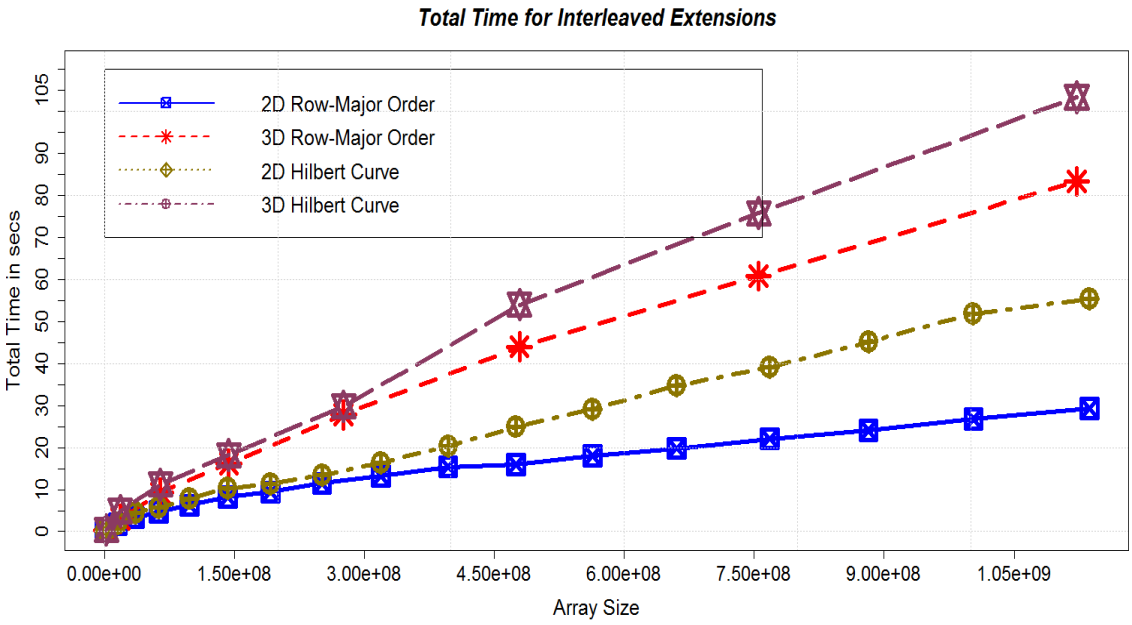


Figure 5.4.2: Hilbert Curve and Row-Major Order Access Cost

random accesses, the row-major order out performs the Hilbert curve, particularly for higher dimensional arrays. This is due to either traversals within the tree structure (using the approach in Section 5.4.1) or using bit calculations and lookup tables in

the Butz algorithms. However as reported in [42], Hilbert curve performs better than row-major order and z-order when the average nearest neighbor stretch (ANNS) metric [43] is used for the performance analysis and comparisons.

6

Experimental Results and Analysis on Extendible Arrays

This chapter describes some experiments conducted to illustrate the performance of the extendible multidimensional array files

Contents

6.1 Experimental Setup

All implementation and experiments were performed on a 64-bit Intel Core 2 Quad Q9650 processor with L2 Cache size of 2×6 MB and a frequency of 3 GHz as well as 12GB of RAM running Ubuntu Linux 12.04 LTS. We used GCC 4.6.3 compiler for the experiments enabling the following option flags; `-O2 -Wall -Wextra -ftime-report`. All array elements are double floating point numbers. Other data types are left for future experiments.

All initial extendible array data files were created before the start of experiments. The array files were created using the Unix file I/O functions. Data elements of each extendible array file was generated randomly using `random()`. The dimension to extend in each experiment was randomly selected.

6.2 Experiments Conducted

A number of experiments were conducted to compare the performance of our scheme with other proposed schemes. Although most of the experiments were conducted predominantly for disk resident chunked extendible arrays, we also tested the performance of the schemes for memory resident arrays as well. The expected times for random accesses of array elements for memory resident chunked extendible arrays (with array size of 10^9) when there are no expansions (i.e. *Static Arrays*) were compared with conventional arrays for $k = 2, 3, ..., 7$. The total cost for random element accesses, when there are random requests to extend the index range of

the dimensions, were compared with similar operations on a conventional array, for $k = 2, 3, 4$.

The rest of the experiments involved storing all array elements on disk. We compared the total random access times when there are interleaved extensions of our scheme with similar interleaved extensions using the HDF5 storage and also with the conventional array storage method. The extendibility was for single index extensions on any arbitrary dimension.

The *storage utilisation*, which is the amount of storage used for any given allocation when there are random requests for arbitrarily random chosen extensions was also computed for $2D, 3D$ and $4D$ (i.e. $k = 2, 3, 4$) using an array size of $1e9$. The final sets of experiments were conducted to measure the performance of our scheme for different *access orders* or *patterns*. These final experiments were necessitated by the fact that using an access order different from the storage order of array files can increase the seek time of an array element significantly. The experiments were performed by first extending the arrays arbitrary and then timing the random accesses of all array elements after all expansions were done. This was done initially for the instance when the access pattern was the same as the storage order of array elements and then for the case when the access pattern was different. The total time for the random accesses of the chunked extendible array was compared to those of accessing conventional arrays in both cases.

6.3 Experimental Datasets

In this section we explain how the simulation datasets used in the four experiments were allocated. In the first experiment (in-memory access without extension), we allocated conventional and extendible multidimensional array for $k = 2, 3, ..., 7$. The size of each multidimensional array for each k was approximately $1e9$. When an array is allocated we assign a random number between the range $1 - 100$ to a cell. For each k, the allocation and assignments are done before the random access described in 6.2. For $k = 2$, we allocated $A[31622][31622]$ with size, $t_s \approx 1e9$. Similarly the dimensional bound for $k = 3, 4, 5, 6, 7$ are $1000, 177, 63, 31, 19$ respectively.

In the second and third experiments, we used the same simulated datasets with the exception that one experiment was completely performed in memory and the other on disk. In both conventional and extendible arrays, we initially allocated an array block and incrementally extend the bounds of the dimensions. For $k = 2$, we initially allocate $A[1976][1976]$ and then we had 15 successive and arbitrary extensions. In the case of $k = 3$ and $k = 4$ we initially allocated $A[142][142]$ and $A[44][44]$ and subsequently had 7 and 4 arbitrary extensions respectively. We used chunk sizes of 8, 16 and 32, however the results reported in this thesis used chunk size of 32. Similar pattern of data generation was used for the fourth experiment (i.e. storage utilization).

6.4 Performance Results

6.4.1 Element Access Cost

Figure 6.4.1 shows a comparison of the average access cost of accessing an array element of our in-memory chunked extendible array and the average access cost for the conventional array for static arrays. We will use the abreviation XTA, to mean *extendible array* in our graph labels.

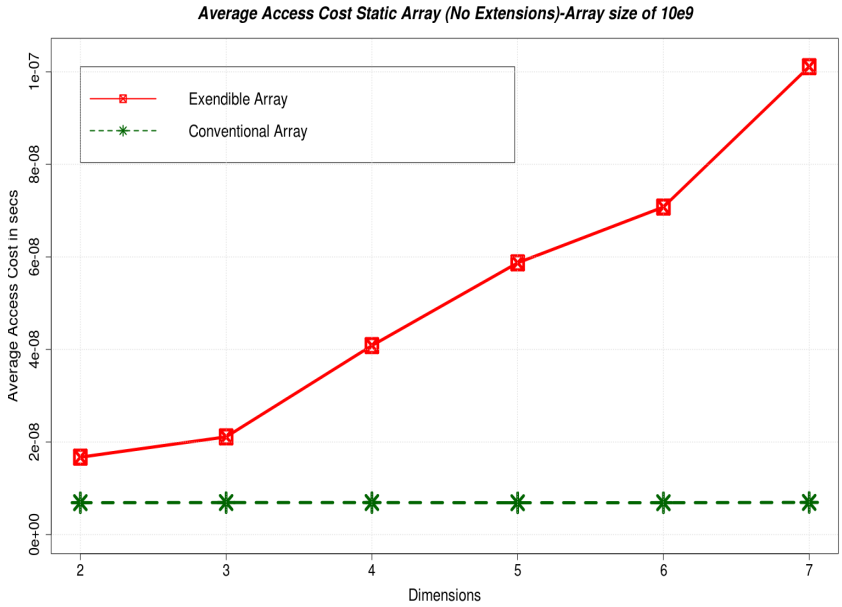


Figure 6.4.1: Average Access Cost without Extensions (in Memory, the case of Chunking)

Similar to Figure 3.5.1, the graphs in Figure 6.4.1 show that the chunked extendible array access function varies significantly with the dimension or rank of the array. This dependency on dimension k is due to the fact that a binary search tree (O_2-Tree) was used for our axial-vector searches. The conventional array in this case performs better because of the traversals within the O_2-Trees even when there are no expansions. But with interleaved extensions, as illustrated in Figure 6.4.2, the chunked extendible

array performs considerably better than the conventional array. Chunking provides an improved performance in access operations as compared the results in Figure 3.5.2 in Chapter 3. The high average access cost of conventional array is the result of the reorganisation of array elements each time an arbitrary dimension is extended.

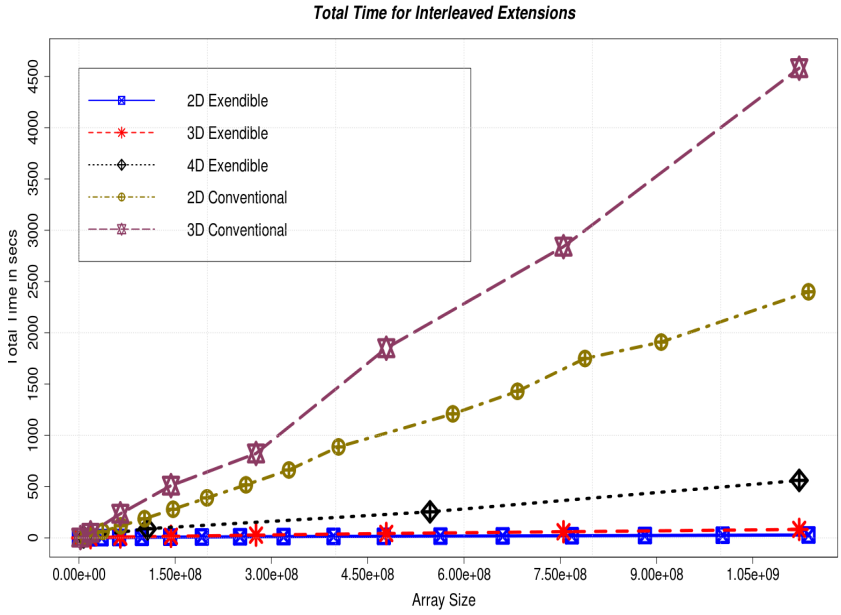


Figure 6.4.2: Total Access Cost for Interleaved Extensions in Memory, the case of Chunking

The experiments whose graphs are shown in Figure 6.4.3 are similar to those conducted to obtain the graphs in Figure 6.4.2, with the exception that the the data is stored on disk. In these experiments, we considered the impact of the total access cost when array files undergo interleaved extensions. We compared our chunked extendible arrays under similar situation with HDF5, conventional arrays as well as extendible arrays with single extensions. The experiments considered arrays of ranks 2, 3 and 4 where the array files grew from 3×10^6 to 1.2×10^9 array elements. Under this situation, it was found that the total access cost of our chunked extendible array was significantly less than that of the conventional array, HDF5 and even extendible

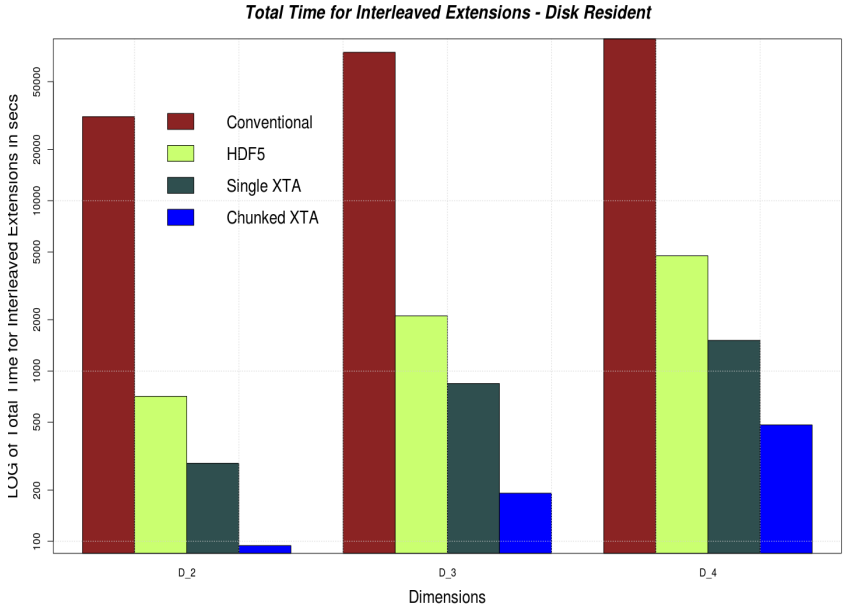


Figure 6.4.3: Total Access Cost for Interleaved Extensions on Disk

arrays with single extensions. This improved performance is due to a number of reasons. Firstly, the indexing scheme used in our scheme (the O_2-Tree) performs query operations (searches) in $O(\log_2 n)$ time, where n is the number of internal nodes. Secondly, unlike extendible arrays with single extensions, in our chunking scheme some expansions do not see entries of the axial-vectors onto the O_2-Tree, thereby reducing the height of the O_2-Tree significantly. The conventional arrays and HDF5 however incurred the cost of reorganising already allocated array elements.

6.4.2 Storage Utilization

The storage utilisation is given by

$$\frac{\prod_{i=0}^{k-1} \mathbb{U}_i^*}{\prod_{i=0}^{k-1} \rho_i \times \prod_{i=0}^{k-1} \chi_i} \tag{6.4.1}$$

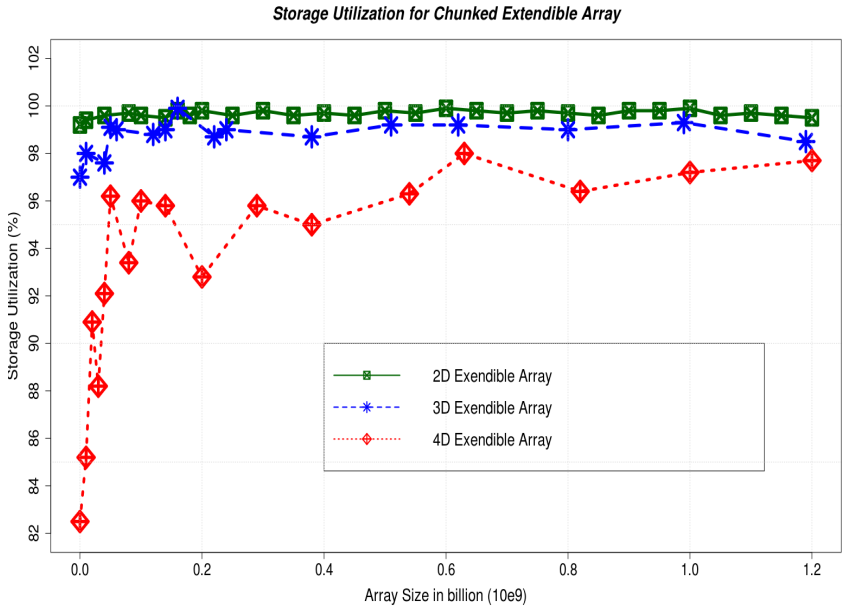


Figure 6.4.4: Storage Utilization for Chunked Extendible Array

where $\prod_{i=0}^{k-1} \mathbb{U}_i^*$ is the array space used and $\prod_{i=0}^{k-1} \rho_i \times \prod_{i=0}^{k-1} \chi_i$ is the total chunked space allocated.

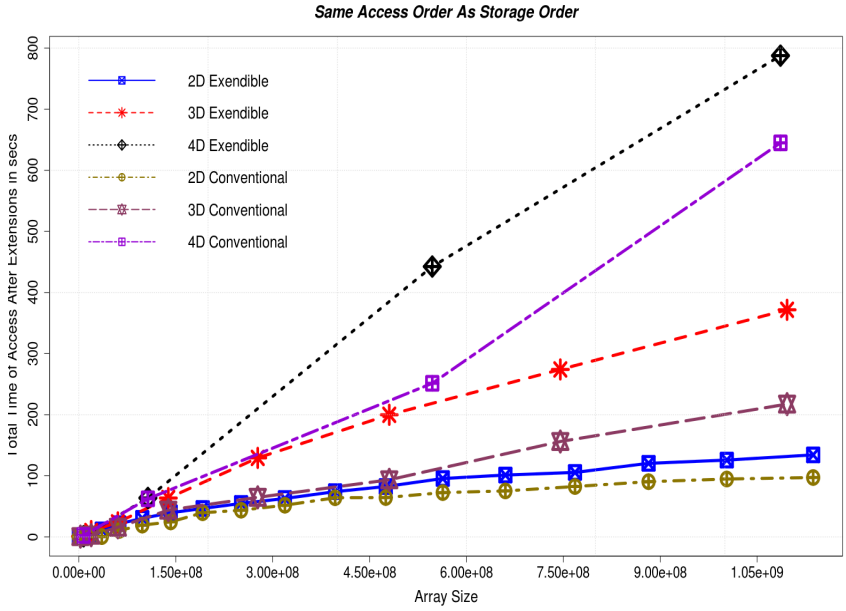


Figure 6.4.5: Total Access Cost for Same Access Order as Storage Order

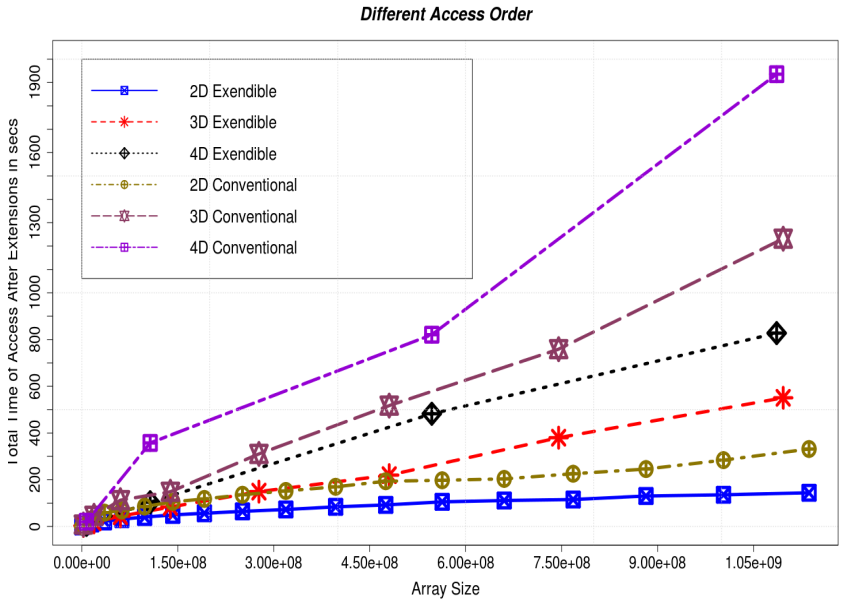


Figure 6.4.6: Total Access Cost for Different Access Order

The storage utilisation decreases with increasing ranks of the array as illustrated in Figure 6.4.4. The results show that, even with large chunk size (chunk size of 32) the usage of space is within 80 to 90 percentiles.

We finally compared the total access cost when the order of accesses differs from the order of storage. The total access cost for the chunked extendible array when the access pattern is the same as the storage pattern in Figure 6.4.5, is slightly higher than conventional array because all arbitrary extensions were fully completed before accessing all array elements. Changing the access pattern in Figure 6.4.6, increases the total access cost for conventional array significantly. The access cost for the chunked extendible array in the case when the pattern of access differs from the storage does not increase significantly. This makes our scheme suitable for applications that require different patterns of access.

7

Scientific File Formats

This chapter describes translation functions that map extendible array files to well known file formats such as HDF5 and NetCDF

Contents

7.1 Overview of HDF5 and NetCDF File Formats

Large scientific datasets are usually stored in scientific data formats such as Network Common Data Form (NetCDF) [44], Hierachical Data Format, version 5 (HDF5) [45], Flexible Image Transport System (FITS) [46] etc. These file formats provide scientists the ability to store and retrieve multidimensional arrays, which are the building blocks of scientific data exploration. Most of these formats have been extended to support parallel data access (e.g., parallel NetCDF). However one of the challenges that is common to almost all scientific data formats is that their interfaces do not support dynamic extensions of array bounds during computation. In this chapter, we introduce the two most widely used scientific file formats namely NetCDF and HDF5, and provide mapping techniques of translating datasets in these formats onto the chunked extendible array file structure and vice versa.

7.1.1 Hierachical Data Format, version 5 (HDF5)

HDF5 is a data library and file format for managing and storing scientific datasets [45]. HDF5 files are self-describing and allow users to specify complex relationships between data and dependencies between objects. HDF5 architecture has three levels, namely the data model, software library and file format. The data model provides the classes used to construct models for user-defined objects. The following are the abstract classes specified in the data model; the *file class* holds the HDF5 dataset and group objects, the *dataset class* is a multidimensional array of data elements, the *group class* is a set of links between HDF5 datasets and group objects, the *linked class* provides a named reference to an HDF5 dataset or object and the *attribute class* provides

the metadata associated with HDF5 data or group. Figure 7.1.1 shows an HDF5

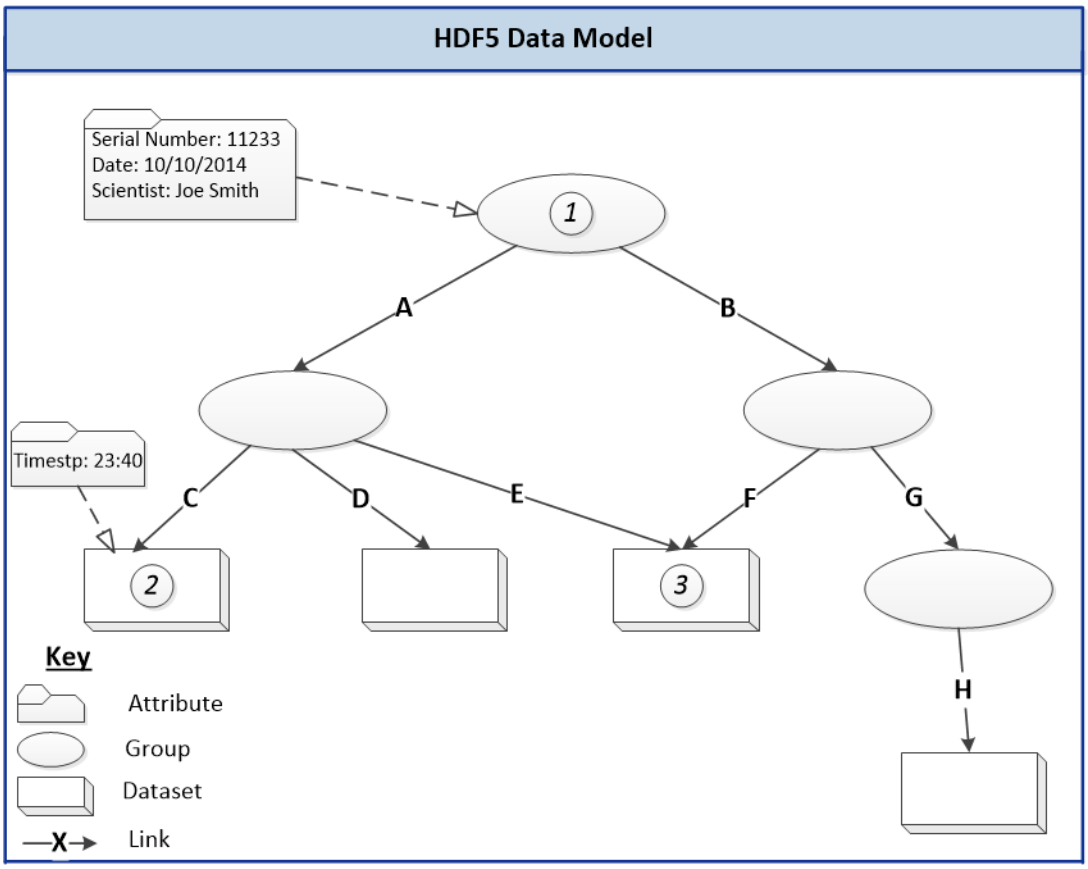


Figure 7.1.1: An illustration of HDF5 Data Model

container with data model classes. HDF5 objects do not have explicit names but are referred to by their links from multiple locations in a hierarchical fashion. Object 1 in Figure 7.1.1 is the root group for the file and it is referenced by the name "/". Object 2 is a dataset, referenced to by the name "/A/C". Object 3 shows a dataset object that is from two different groups/locations hence, it is referenced by either "/A/E" or "/B/F". Each HDF5 dataset has metadata that describes the array dimensions, the data element type and the storage layout. Figure 7.1.2 shows a three-dimensional

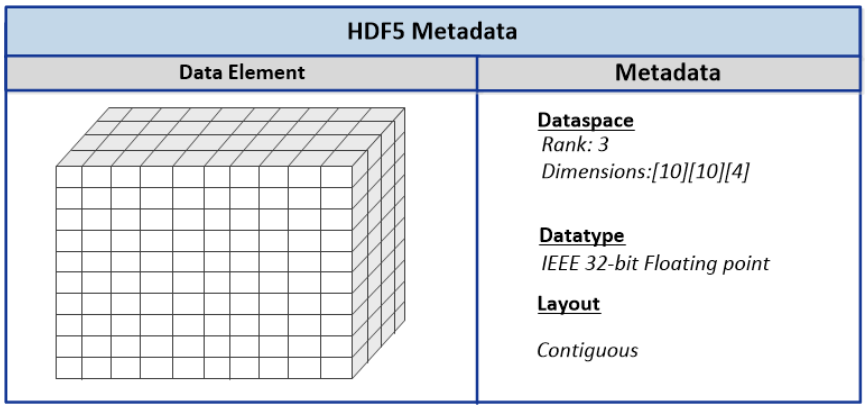


Figure 7.1.2: An example of a $3D$ HDF5 Data Structure

dataset with 32-bit floating point data elements, stored contiguously.

The dataspace describes the dimensions (or the rank) and it can range from 0 (single element) to 32-dimensions. Our extendible array model can however accommodate any number of dimensions. The dimensional bound on an HDF5 file is either set to fixed size or set to unlimited (the C implementation uses `H5S_UNLIMITED`). To enable extensions, the dimensional bound must be set to unlimited. HDF5 allows users to extend the bounds of a multidimensional array dataset by using the following steps:

i Declare the dataspace with dimension bound set to unlimited.

ii Enable chunking for the dataset properties.

iii Create the dataset.

iv Extend the size of the first dimensional bound.

The layout of an HDF5 dataset describes how the elements are stored in a file. HDF5

allows multiple layout optimizations such as *contiguous layout* for storing elements of very small array files (i.e. for faster access), *chunked layout* for storing large arrays in regular chunks (this allows for dimension expansion and data compression) etc. The attribute object permits user-defined metadata information to any object container.

The HDF5 library is implemented in C with wrappers for languages such as FORTRAN, C++, Perl, Java etc., all of which call the C API routines. The function `H5Dset_extent(hid_t dset_id, const hsize_t size[])`, responsible for dimensional bound extension in the library has the following limitations:

- it can only be applied to chunked HDF5 datasets (see Section 7.2).
- the dimensional bound of the external dataset with which the in-memory data is written to must set unlimited.
- the external dataset must be set to unlimited.
- extension can only be performed on the first dimension.

In addition to the classes described in the data model section earlier, additional classes are available at runtime for setting properties that control the behaviour of objects and operations, selecting regions of dataspaces to perform partial I/O, and a variety of classes that support infrastructure operations like error reporting and object management.

HDF5 file format is designed to be portable and resilient to data corruption. It has micro-versioning features which allow individual aspects of each object to be stored with a different version. This versioning allows the format of individual components

of objects to be incrementally extended without version changes for the entire file format.

HDF5 file systems are organined hierarchically into groups analogous to directories and datasets of similar files. As a result, it uses indexing structures such as B-trees, heaps, hash tables etc. It uses B-trees to store the records used in locating a chunk. These records store the coordinates of the chunk in the dataset's dataspace (this is used as the key for locating chunks in the B-tree), the size of the chunk, in bytes, and the address of the chunk in the file [1]. Each leaf node of the B-tree points to the address of a chunk. When a dataset is extended, a new entry is added, which may generate a node split if the node where the record needs to be inserted is full. In HDF5, extensions are only done in one dimension. This limitation that can be overcome by using our extendible array structure to allow extension of any dimension. This is however not the case for HDF5. The B-tree requires frequent re-balancing to avoid empty nodes. Appending records to the dataset also requires traversals and/or multiple B-tree nodes' updates imposing additional performance penalty.

To overcome this challenge, in [1] a dynamic array structure, called Extendible Array (EA) was introduced. The structure defines *super blocks* which conceptually group data blocks. Super blocks can be seen as indirect blocks in the data structure and help to show the pattern of changes to the data blocks. The EA structure is illustrated in Figure 7.1.3. The first two super blocks are omitted and the pointers in the index block point directly to the data blocks. The structure only supports extension in one dimension, hence its simplicity. It has a constant lookup time (i.e., O(1)), the number of I/O's to find any node is 2 or 3. The height of EA is constant as the

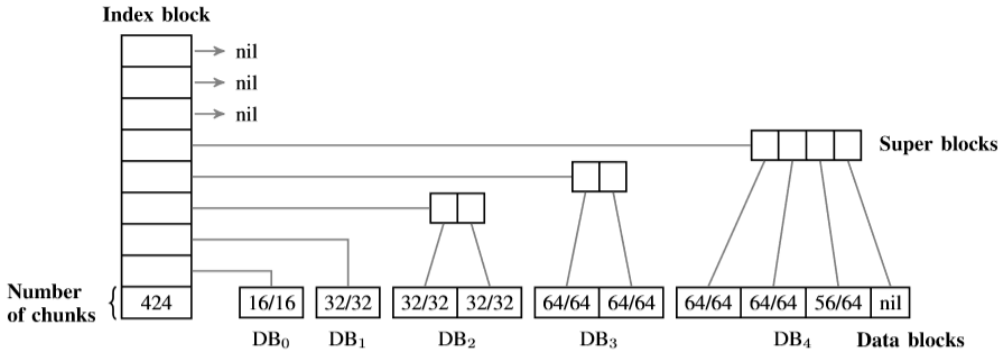


Figure 7.1.3: Extensible array structure used for indexing HDF5 chunks [1].

number of chunks needed is allocated in advance. Super blocks not yet allocated are assigned nil pointers.

7.2 HDF5 Chunking Allocation

HDF5 requires the use of chunking when defining extendible datasets and enabling compression and other filters like checksum. Chunking makes it possible to extend datasets in the first dimension efficiently without having to reorganise contiguous storage excessively. Chunking improves I/O for big datasets.

```
dcpl_id    = H5Pcreate(H5P_DATASET_CREATE);
rank = 2;
ch_dims[0] = 100;
ch_dims[1] = 200;
H5Pset_chunk(dcpl_id, rank, ch_dims);
dset_id = H5Dcreate (..., dcpl_id);
H5Pclose(dcpl_id);
```

Listing 7.1: Creating HDF5 Chunked Dataset

HDF5 data chunks are stored in predefined sizes. HDF5 library always writes/reads the whole chunk. As illustrated in Figure 7.2.1, each chunk is stored separately as a contiguous block in HDF5 file. The chunked block is indexed using the extendible

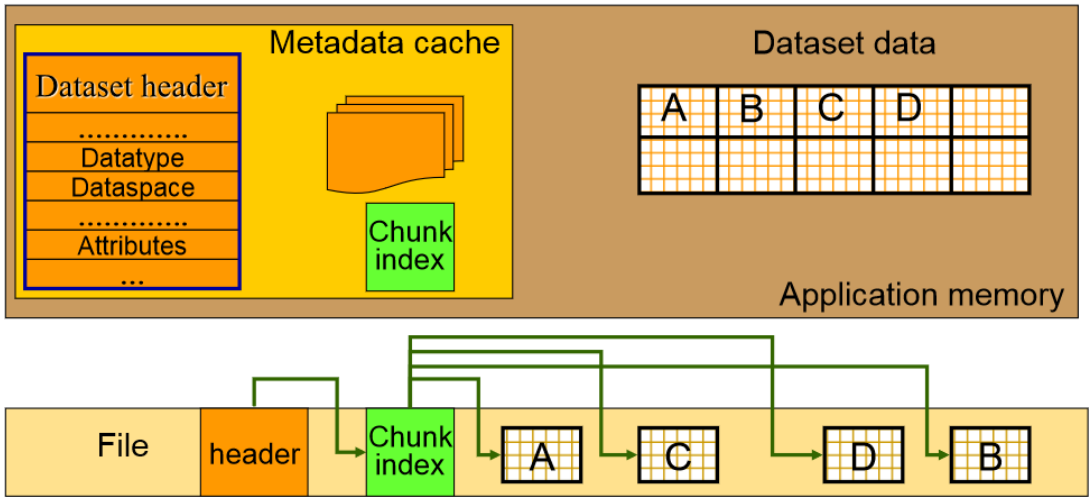


Figure 7.2.1: HDF5 chunking.

array (EA) structure in Section 7.1.1. Chunked HDF5 dataset is created by first creating the property list, setting the property list to use the chunked storage layout and then eventually creating the dataset with the property list. This is illustrated in code listing 7.1.

Inappropriate chunk size can dramatically reduce performance. When chunks are made smaller, there is usually I/O overhead due to the extra time needed to look up each chunk. In addition, since the chunks are stored independently, more chunks results in more I/O operations. The extra metadata needed to locate the chunks also causes the file size to increase as chunks are made smaller. Larger chunk size results in fewer chunk lookups, smaller file size, and fewer I/O operations in most cases.

However, if the in-memory cache is not large enough to hold large chunk dataset, it may stall the I/O operations.

7.2.1 Network Common Data Form (NetCDF)

NetCDF (network Common Data Form) is an interface library of access functions for storing and retrieving array-oriented scientific datasets. It is self-describing and provides portable objects that can be accessed through simple interfaces. NetCDF data model contains dimensions, variables and attributes. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset.

A convenient way of describing NetCDF datasets is by the use of a descriptive language called Common Data Form Language (CDL). The CDL notation for a NetCDF dataset can be generated automatically by using the `ncdump` utility. In CDL, statements are terminated by a semicolon. Spaces, tabs, and newlines can be used freely for readability. A CDL description of a NetCDF dataset takes the form in code listing 7.2.

```
netCDF name {
  types: [netcdf-4 only]
  dimensions: ...
  variables: ...
  data: ...
}
```

Listing 7.2: Basic structure of CDL

The CDL consists of three optional parts namely dimensions, variables, and data. NetCDF dimension is used to represent a real physical dimension, for example, height, latitude, longitude etc. A dimension has both a name and a length. The length is usually an arbitrary positive integer.

Variables are used to store the bulk of the data in a NetCDF dataset. A variable represents an array of values of the same type. A scalar value is treated as a 0-dimensional array. A variable has a name, a data type, and a shape described by its list of dimensions specified when the variable is created. Variables sometimes have associated attributes [44]. The CDL declaration for variables appear after the `variable` keyword. They have the form `datatype variablename( dimname1, dimname2, ...)` for variables with dimensions, or `datatype variablename` for scalar variables. Variables with the same name as a dimension are called *coordinate variables*. It typically defines a physical coordinate corresponding to that dimension.

NetCDF attributes are used to store metadata information of the datasets, similar to the use of attributes in HDF5. They can provide information about a specific variable. This is identified by the name or ID of that variable, together with the name of the attribute. Attributes that provide information about the entire datasets are called global attributes. These are identified by the attribute name together with a blank variable name (in CDL) or a special null global variable ID (in C or Fortran). The CDL notation for defining an attribute is `variablename:attributename = list_of_values;` for a variable attribute, or `:attributename = list_of_values;` for a global attribute.

7.3 Mapping Functions for HDF5 and NetCDF

This section presents one of the contributions of this thesis. We provide routines that extend HDF5 and NetCDF to support extendibility across multiple dimensions. In order to provide extendibility of the bounds of the dimensions for scientific file formats, there is a need for an implementation that provides routines for reading in and writing out multidimensional dataset for these formats. Users of NetCDF will benefit from support for packed data, large datasets, and parallel I/O for other operations. Users of HDF5 will benefit from the availability of a simpler high-level interface suitable for array-oriented scientific data, wider use of the HDF5 data format, and more software for data management, analysis and visualization that has evolved among the large NetCDF user community.

Applications may continue to use the HDF5 and NetCDF interface directly. However to enable extendibility in multiple dimensions, datasets in these formats need to be read into our data model. Once the application is done with extensions, data can be written out into these data formats.

To read an HDF5 file, we first open the file using:

```
fhdl          H5Fopen(filename, flags, propertylist)

              char *filename_input
```

which opens a named file (`filename`) in the specified access mode (`flags`) and with the specified access property list (`propertylist`). This returns a file identifier, `fhdl` which is then passed to `H5Dopen(fhdl, datalocname, dataproplist)` to open existing dataset specified by a location identifier, `datalocname` and dataset access property `dataproplist`. We now provide two mapping functions namely

xtaHDFmap and xtaHDFwrite that enable HDF5 object handles to undergo dimensional bound extensions. The function xtaHDFmap takes as input an HDF5 data handle (dathdl) and returns a distinctive extendible array data object handle xta_hdf, containing all the structures of chunked extendible arrays as well as the attributes and hierarchical data objects in dathdl. xtaHDFmap only accepts data handles of chunked HDF5 datasets. xta_hdf can now undergo extensions on multiple dimensions. The xtaHDFwrite(xta_hdf, dathdl) writes the extended dataset from the xta_hdf buffer to an HDF5 dataset (dathdl). Once data has been written to dathdl, all the HDF5 library functions including H5Dread can be applied as specified in [47].

NetCDF files are accessed using nc_open(path, omode, ncidp). ncidp is a pointer to the location where the NetCDF ID is to be stored. This is then passed to nc_inq_varid(ncidp, name, varidp) which returns the ID of a NetCDF variable, given its name. Once the variable is retrieved, the variable ID (varid) is passed to nc_get_var_type(ncidp, varid, datap) to read the multidimensional dataset for that variable. We then pass datap (the data pointer), ncidp and varid to xtaNetCDFmap which maps the dataset onto the chunked extendible array structure. This returns an extendible array object handle called xta_ndcf. Similar to HDF5, xtaNetCDFwrite(xta_ndcf, varid) writes an extended dataset to the NetCDF variable.

8

PEXTA: A Parallel Chunked Extendible Dense Array I/O for Global Array

This chapter introduces PEXTA, a new parallel I/O model for the Global Array Toolkit.

Contents

8.1 Global Array

High performance computing infrastructure will continue to have physically distributed memories with Non-Uniform Memory Access (NUMA) characteristics. The message passing programming model is frequently used because of its portability but some problems or applications are too complex to be programmed with it in order to maintain balanced computations and avoid redundant computations. Although shared memory programming model is not portable and lacks full control over interprocess data transfer, it simplifies coding and is very easy to code in. Partitioned global address space (PGAS) programming model combines the good features of both models leading to a simple coding and efficient execution [48, 49]. PGAS models assume a global memory address space that is logically partitioned and a portion of it is local to each process or thread. Unified Parallel C (UPC) and Global Array (GA) are two widely used PGAS programming environment.

GA enables a portable interface through which each process in a multiple instruction, multiple data (MIMD) program can independently and asynchronously access logical blocks of physically distributed matrices with no need for explicit cooperation by other processes. GA allows data locality to be explicitly specified and is used because remote data is slower to access than local. GA uses the NUMA model of current parallel architectures but removes unnecessary processor interactions that are required in the message-passing paradigm to access other processes thereby simplifying the parallel programming model.

8.1.1 Disk Resident Array (DRA)

The DRA library extends the GA NUMA programming model to disk. It supports explicit transfer between global memory and secondary storage. Its operations are similar to the GA except that they reside on disk instead of memory. DRA along with GA provides a unified programming model for handling different levels of the memory hierarchy in which the user controls the location of data. It also supports persistent and temporal disk resident arrays. The temporal or persistent arrays are saved after program termination and available for subsequent runs. Reshaping and transpositions are allowed during these transfers. I/O operations have nonblocking interface to allow overlapping of I/O with computations. All I/O operations in DRA are collective. DRA provides the functionality so that either the chunks or the entire global arrays can be transferred between GA memory and the disk. Persistent disk arrays can be accessed by any program executing on arbitrary number of processors.

The DRA provides about 16 routines or functions for creating, opening, closing and accessing disk arrays. For instance, the `dra_int` function initializes the DRA I/O system, the `dra_create` creates a new disk resident array with specified dimension and data type, the `dra_open` opens and assigns DRA handle to disk array. Others include `dra_write`, `dra_read`, etc. Although the DRA and the GA provide high performance and a programming model that is simpler than message passing, it does not provide dimensional bound extendibility. We provide an equivalent I/O library or API using disk resident extendible dense array scheme presented in Section 5.1. This variant API is referred to as Parallel Extendible Chunked Dense Array (PEXTA).

8.2 Parallel HDF5 Files

Parallel HDF5 (PHDF5) [47] enables developing parallel applications using standard paralling programming models like MPI in conjunction with HDF5. PHDF5 exports a standard parallel I/O interface which itself uses MPI's parallel I/O functionality. This is used along with parallel file systems (See Figure 8.2.1) to achieve high performance I/O. Parallel HDF5 is designed to have a single file image to all processes, rather than having one file per process. Most PHDF5 APIs are collective. Parallel HDF5 files are created and opened using `H5Pcreate` and `H5Fopen` and H5Dopen respectively. These functions return a file-handle that is used for future HDF5 access functions' calls. All processes that have opened a dataset may perform independent and arbitrary number of data I/O access calls using `H5Dwrite` and/or `H5Dread`. HDF5

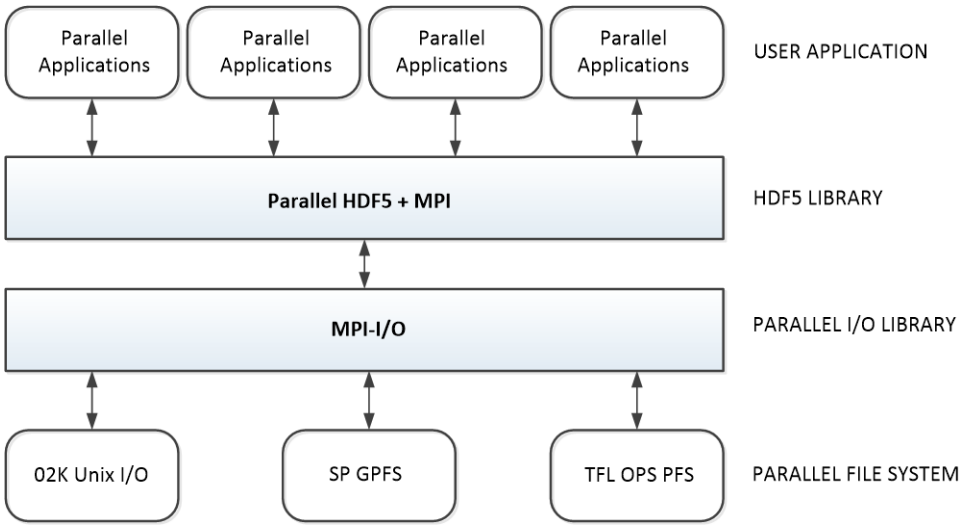


Figure 8.2.1: Parallel HDF5 model

uses access template object (property list) to control the file access mechanism. The general model to access HDF5 file in parallel is to first setup MPI-IO access template, open the file and access the data. Data is split among parallel processes. PHDF5

uses the HDF5 hyperslab model. Each process defines memory and file hyperslabs. We compared our parallel I/O to PHDF5 for two reasons; its layout is consistent with PEXTA and it allows extendibility in one dimension. Although other high level parallel I/O APIs such as Adaptable IO System (ADIOS) [50, 51] have similar layout as PEXTA, its does not support extendibility of the bounds of each attribute dimension.

8.3 PEXTA I/O Model

Unlike the DRA, we modelled the PEXTA input/output (I/O) routines such that internally, they make calls to MPI I/O functions. We use 3-datatypes for data organisations in file and in memory. We leverage strongly on MPI I/O.

For the organisations of PEXTA, the principal-array data and the axial-vectors, that correspond to the leaf-nodes of the O_2-Tree, are stored on disk as the principal-array file, F_p and metadata file, F_m, respectively.

The principal-array file, F_p, grows continuously by appending new data elements (in *chunks*) whenever a dimension is extended. Although F_p is stored on disk, it is accessed through the buffers of each process in a communicator group. Data stored on disk can be accessed by chunking through an in-memory cache. The axial vector file (F_m) is also memory resident but stored on disk after every session and loaded into memory before any usage session starts.

PEXTA supports both regular and irregular distributions of global array of the GA-Toolkit. Various patterns of regular distributions are illustrated in Figure 8.3.1. The

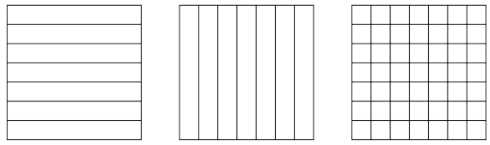


Figure 8.3.1: Regular Distribution of Global Array

third distribution in Figure 8.3.2 is a regular block distribution. Each process within a process group has a filetype that determines the pattern of access in the principal-array file, F_p. PEXTA also provides file *displacement*, which is the absolute byte

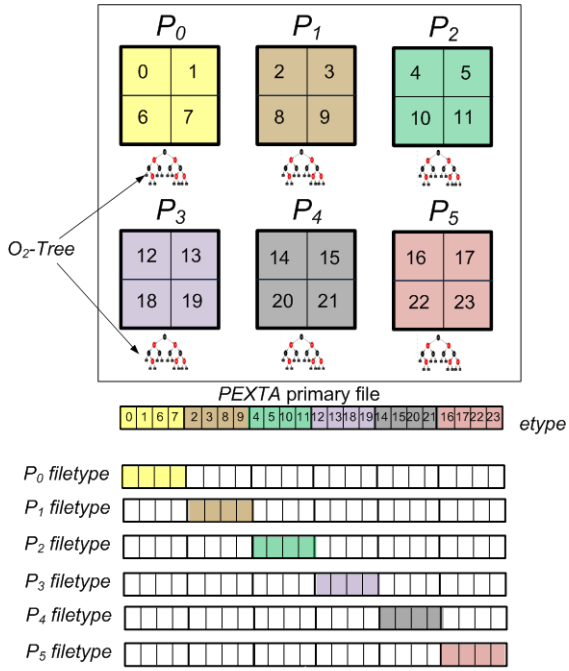


Figure 8.3.2: PEXTA Model for PGAS

position relative to the beginning of the file. The displacement defines the location where a chunk block allocated to a process begins (for regular distribution). In the implementation of PEXTA, the displacement information can also be maintained in the metadata file, F_m as the relative position of the hyperslab. In such a case, the displacement in the principal-array file, F_p, is implemented such that it is consistent

with MPI I/O in which the displacement defines location where a *view* begins.

Processes within GA can perform independent or collective reads and writes of their respective sub-arrays (i.e. hyperslab) as long as the accesses do not overlap new elements of the hyperslab being appended. Collective reads and writes are performed when the entire extendible array is being read into the GA's global memory or when it is being written to file. During collective reads, the metadata file, F_m, is read into local buffers by processes. Other information such as chunk block dimensional bound limits (also called *chunked block boundary metadata*) are added to this metadata. These additional datasets are however not written out to F_m during collective writes because they may change during collective reads (instantiating the extendible array in global memory). An external program can add chunks of data (hyperslab) to the principal-array file, F_p, by requesting an exclusive write lock token on the metadata file, F_m. This does prevent inconsistent metadata information in each process within a group. When the external writer for the new hyperslab completes, the metadata file is updated. In starting a new session, the processes within a communicator group read the newly updated metadata file into their respective local buffers as O_2-Trees, and as such can access the new hyperslab chunked datasets that have been added (either remotely or locally). If the writer is one of the parallel processes accessing the file, an appropriate record, corresponding to the new segment of added elements, is sent to every node to update their respective O_2-Trees held in memory. The appended dataset may however not be available in the global memory for access until the processes perform collective read. Our implementation of the PEXTA storage scheme does synchronisation of the vector of O_2-Trees.

8.4 Implementation of PEXTA for Global Array

We provide the implementation of the PEXTA storage scheme for the Global Array Toolkit. This will serve as a drop-in replacement for the DRA, one of the parallel I/O within ChemIO for NWChem. The PEXTA API for GA basically reads and writes pages (or chunks) of the global array. The PEXTA API provides similar routines for creating disk resident array with similar function signatures but having the definitions for allowing the principal-array file to grow. We achieve this similarity between the DRA and PEXTA routines by modifying objects associated with the handle (i.e.. `p_a`) that is returned by the create function (`PEXTA_Create()` in this case) of our library. This is a pointer to the memory resident header of the meta-data information for the PEXTA array. The PEXTA library provides a header file `pexta.h` that is included in any application wishing to use the functions of the PEXTA API. The meaning of most function parameters can be inferred from the names and data types used in the prototype definition. All PEXTA functions must be enclosed in between `GA_Initialise()` and `GA_Terminate()`. Some examples of the PEXTA functions are given below:

Initialization:

> *int* ***PEXTA_Init****(int maxArrays, double maxArraySize,*
> *double totalDiskSpace, double maxMemory);*
>
> This function initialises the buffers and sets the PEXTA I/O sub-system. `maxArraySize`, `totalDiskSpace` and `maxMemory` are given in bytes. `maxMemory` defines how much local memory per processor the application is willing to provide to the PEXTA I/O subsystem for

buffering.

Opening and Creating:

int ***PEXTA_Create****(int type, int ndim, size_t dims[],char *name, char* filename, int mode, int mode, size_t reqdims[], int g_a, int *p_a);*

This function opens and creates an N-dimensional PEXTA array with specified dimensions and data types. The dimension of the array is `ndim` and the bounds on each dimension is specified in the array `dims`. The `filename` is the *basename* of principal-array and metadata file used to store the PEXTA array. If a user requests the creation of a PEXTA array with base name, `abc`, the corresponding pairs of files created for the principal-array file and metadata file are `abc.pta` and `abc.pma` respectively. The `mode` is similar to the mode in DRA and consistent with MPI I/O. We use `PEXTA_R` `PEXTA_W` and `PEXTA_RW` for read, write and read/write respectively. The `p_a` is an integer handle that is assigned to the PEXTA array. This handle stores the filetype, buftype and displacement. The `PEXTA_Create()` unlike `DRA_Create()` requires `g_a` (a handle for the global array) as input to determine the filetype and buftype. If the global array has not been distributed before passing this handle to the function, then we use the default regular block distribution by calling `NAG_Distribution()`.

Reading:

int ***PEXTA_IRead****(int g_a, int p_a, int *request);*

This function does an N-dimensional asynchronous collective read from the specified PEXTA array, `p_a`, to specified global array, `g_a`. Although the data types and the dimension of the both `p_a` and `g_a` should match, their dimensional bounds can differ as the files corresponding to `p_a` might have been extended by an external process. We currently do not support shrinking, hence the bounds on the `p_a` should not be less than that of `g_a`. We leave the functions for shrinking the bounds of an array for future work. If the dimensional bounds of `p_a` has extended, then the bounds of `g_a` and the O_2-Trees are updated accordingly. We then apply the default regular block distribution on `g_a` due to the new additions. Since this function is asynchronous, the user has to provide `GA_Sync()` to guarantee that all transfers and updates are complete. We however provide `PEXTA_Read()` for collective block read.

Extending:

```
int PEXTA_Extend(int p_a, size_t edims[],size_t
odims[], int *request);
```

This provides asynchronous extensions to the PEXTA array `p_a` by adding new hyperslabs of data to the dimensions specified in the array `odims` as described in Section 8.3. The array `edims` contains the λs, $\lambda_0, \lambda_1, ..., \lambda_{k-1}$, to be added to to dimensional bounds $[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$. The array `odims` contains the order of the extensions. A negative value in this array means no extensions. The primary and metadata files are then updated.

The remaining 15 PEXTA routines have the same signatures as DRA, hence we do not provide their description here. In processing the extendible disk resident array file, each process of a group accesses and processes sub-arrays independently and may exchange copies of the memory resident arrays with other nodes. The mapping function in Section 5.1 is significant for such applications since large array files can grow by appending new data items to the file without reorganising the entire file. However, the increase in the index ranges of any dimension must be subsequently reflected in the O_2-Tree of the file before the new elements can be accessed. The technique for accessing sub-arrays, maintains distributed copies of the same O_2-Trees associated with the file, in each process.

8.5 Experimental Results and Analysis

8.5.1 Experimental Datasets

In section, we describe the simulated dataset used in the three experiments illustrated in Section 8.5.2. The first experiment evaluates the read and write access rates of PEXTA, DRA and HDF5. We allocated a $3D$ array $A[5000][5000][5000]$ of randomly allocated double precision array elements using PEXTA, DRA and HDF5 APIs. In the second experiments where we performed cubical extensions, we initially allocated array $A[12][12][12]$ and extended in a round robin fashion using $\lambda = 2$ (extended hyperslap dimensional size). These extensions were repeated until the size of the array $t_s \approx 64GB$.

The last group of experiments involves scientific applications that utilize 2×2 square matrices. These experiments were matrix multiplication, molecular dynamics of Lennard Jones System and Lattice Boltzmann simulation of flow. In the matrix multiplication, we allocated square matrices of dimensional bound size $N = 1000$ for both matrix A and B. Interleaved extensions with random λ between the range $[1 - 100]$ were performed on the matrices during the matrix multiplication. This was repeat dimensional bound sizes $N = 2000, 3000, ..., 7000$. Similar procedure for the data generation was followed for the Lattice Boltzmann simulation of flow and molecular dynamics of Lennard Jones System experiments. The initial dimensional bound sizes of the grid matrix and the force matrix were $N = 500, 1000, 2000, 4000, 8000, 12000, 14000, 16000$.

8.5.2 Experiments Conducted and Analysis

Experiments were conducted in a cluster (called ZA-Wits-Core Cluster) of 24 nodes having a total of 224 cores. Each machine runs Scientific Linux 6.3, at 2.45 GHz with an average of 24GB main memory on each machine. We use three experiments to illustrate the performance of the PEXTA library. The first set of experiments is a benchmark to assess the read and write performance of the API. The read and write are applied to a $5000 \times 5000 \times 5000$ double precision global array. We considered *aligned* and *unaligned* reads/writes. An I/O operation is said to be aligned if the corresponding section on disk array can be decomposed into subsections that can be read/written as a consecutive sequence of bytes. Otherwise it is unaligned.

Figure 8.5.1 and Figure 8.5.2 show the transfer rates for read and write operations

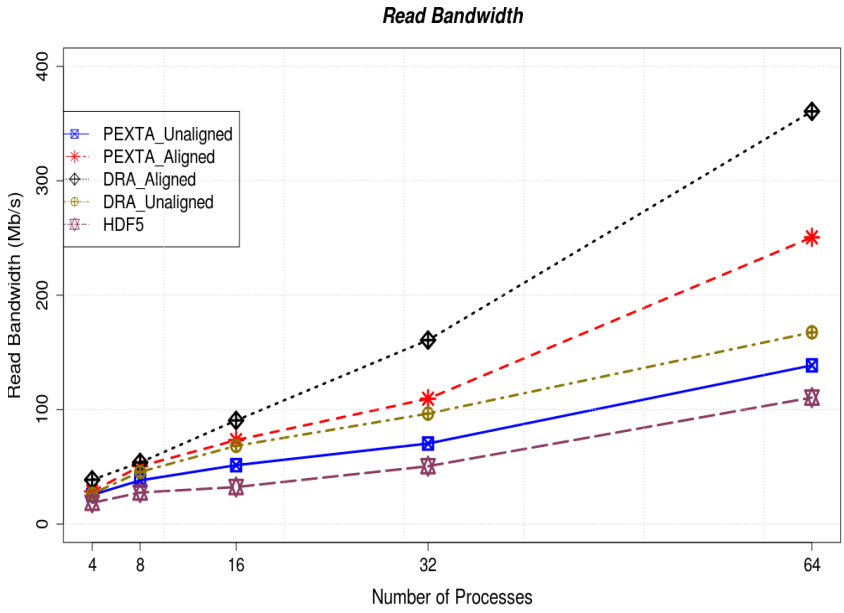


Figure 8.5.1: Transfer rate for aligned and unaligned Reads

respectively. We varied both the internal buffer and the number of processes that performed the reads or writes. Write operations appear to execute faster than read

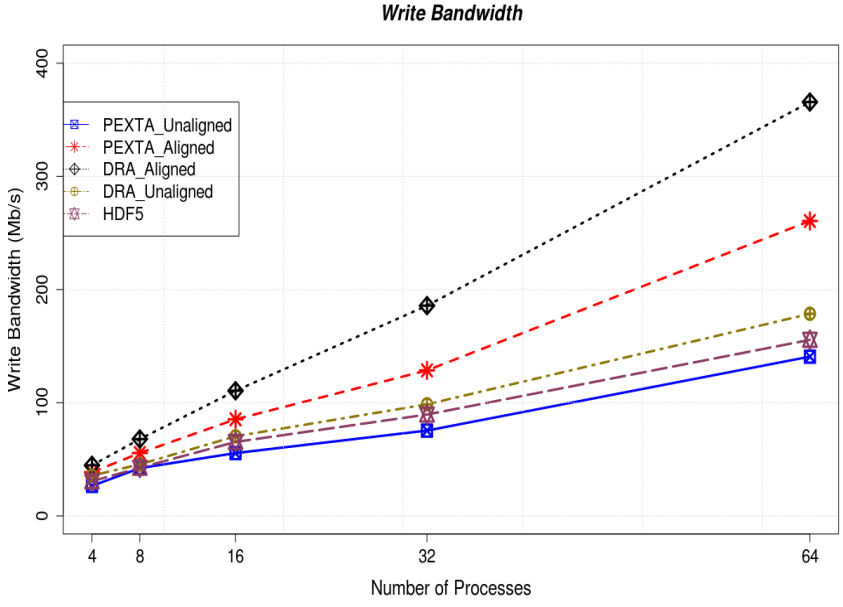


Figure 8.5.2: Transfer rate for aligned and unaligned Writes

operations when more processes are used. In general the DRA aligned I/O performed fastest and had the highest transfer rates. Performing aligned I/O in PEXTA also requires that each process builds its O_2-Tree from the metadata file. This comes with a little overhead. In unaligned I/O operations for both PEXTA and DRA, the disk array section is initially decomposed into aligned and unaligned subsections. For instance, write operations on unaligned disk array sections are done by first performing an aligned read to transfer an augmented aligned block to the buffers and then overwriting the relevant portions before writing the augmented block back to disk. In Figure 8.5.3, we considered a situation where the principal-array file on disk

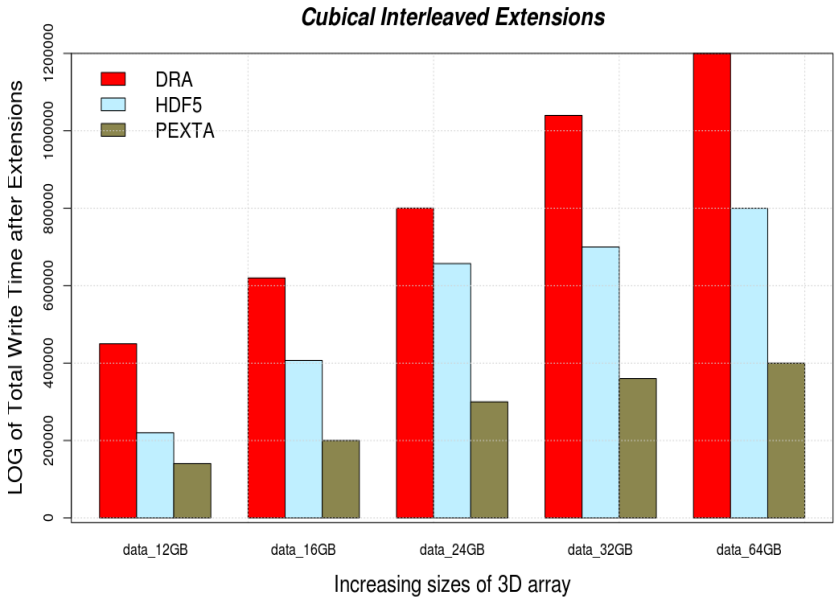


Figure 8.5.3: Log of Aligned Write Time after Cubical Interleaved Extensions

is extended by an external process while computation is been done on the in-memory global array. We then stopped the session and wrote out the content of the global array to the principal and metadata files. When a new session is started, we read the newly modified principal-array and metadata files. Parallel HDF5 and DRA had an

overhead in writing out such dataset because the incoming in-memory datasets and the newly appended hyperslabs need to be reorganised in order to accommodate the increased dimensional bounds. PEXTA writes however do not incur any additional overhead since the chunks of the in-memory global array are written to the same chunked locations on F_p. F_m is not overwritten by the in-memory O_2-Tree as it is already updated by the external process. This makes the total time for PEXTA reads and writes far less than those of DRA and parallel HDF5. The high I/O cost for DRA and parallel HDF5 is due to the fact that the already allocated array elements need reshuffling [16, 17]. The size of the $3D$ principal-array on disk was increased from 12GB to 64GB of data size, by cubical extensions, and we took the read/write times after each extension is completed by the external process.

The last three experiments follow the same pattern as described above for Figure 8.5.3. In Figure 8.5.4, we multiplied two matrices that resided on disk and wrote the results

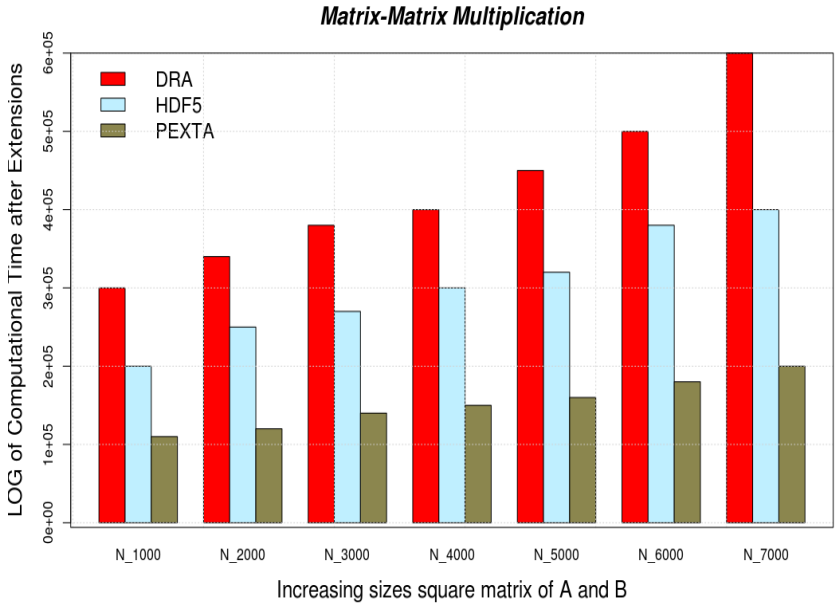


Figure 8.5.4: Log of Computational Time for Matrix-Matrix Multiplication with Extensions

to a third matrix also stored on disk. The matrix multiplication was performed using the Shared and Remote-memory based Universal Matrix Multiplication Algorithm (SRUMMA) [52]. We chose this algorithm because its simpler to implement out-of-core matrix multiplication based on SRUMMA. The distributed array layout required were well represented by the GA library without additional modification. As the multiplication was being done, an external process extended the bounds of the square input matrices. Before new session of the multiplication is started, the process reloaded the new bounds. The parallel HDF5 and DRA incurred a similar overhead of reorganising the matrices as explained in Figure 8.5.3.

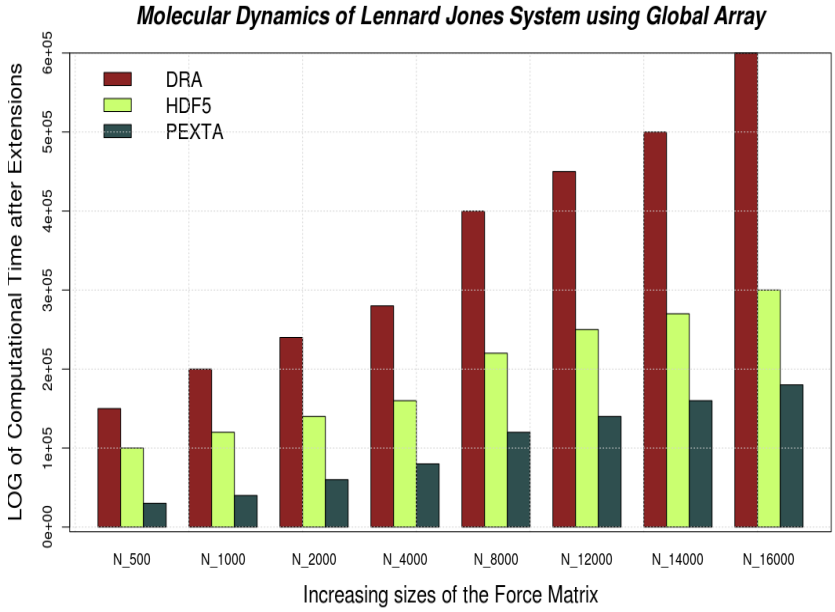


Figure 8.5.5: Molecular Dynamics of Lennard Jones System Simulation using Global Array

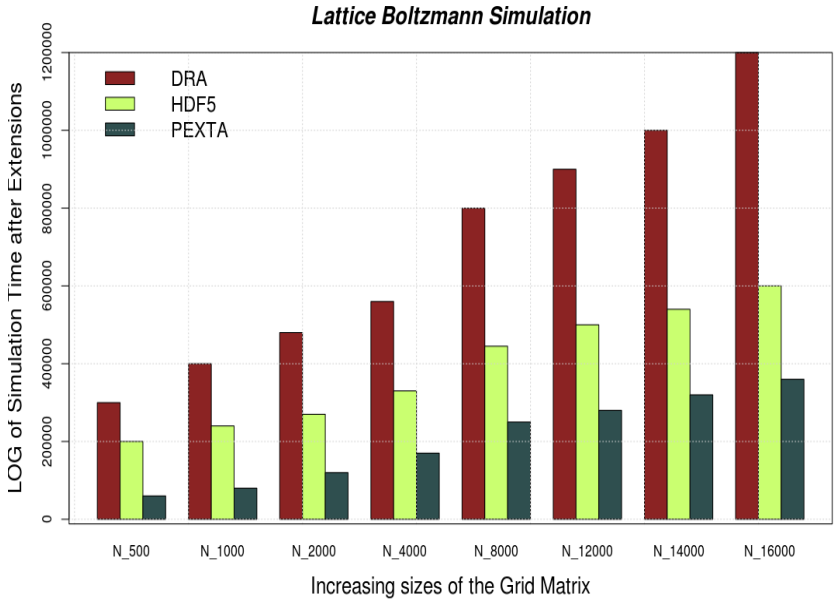


Figure 8.5.6: Lattice Boltzmann Simulation with Interleaved Extension of Grid Matrix

In Figure 8.5.6, we performed a simple Lattice Boltzmann simulation of flow in a lid-driven square cavity. We allowed an external process to extend the Grid matrix used for the simulation. The extensions were allowed to overlapped the simulation. A similar approach is used in the simulation of the Molecular Dynamics of Lennard Jones System (Figure8.5.5). The extensions of the Force matrix is allowed to overlap the computation. In both cases, there is the overhead of reorganisation of array element when using parallel HDF5 and DRA.

9

Hyperspectral Image Analysis using GA and PEXTA

This chapter describes the application of PEXTA APIs for Hyperspectral Image Analysis

Contents

The implementation of computational techniques for extracting valuable information from remote sensing datasets is crucial for space-based Earth science and planetary exploration. This extraction of information has aided environmental modelling and assessment for Earth-based studies, biological threat detection, monitoring of oil spills and other types of chemical contamination for military and defense/security purposes [25]. The quest for real time analysis and rapid inspection of massive data archives and extraction of information has introduced the the use of HPC technologies, such as parallel and distributed computing in the processing hyperspectral imagery datasets.

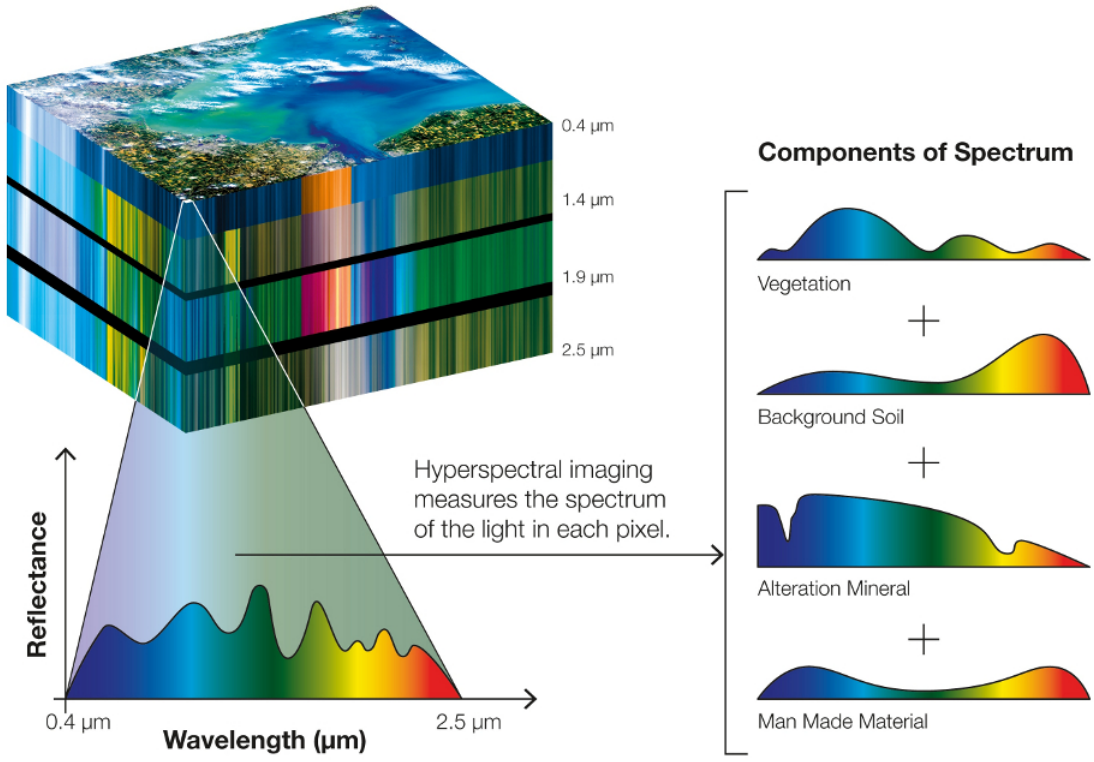


Figure 9.0.1: Hyperspectral Image Acquisition and Processing

Hyperspectral image spectrometer collects hundreds or even thousands of measurements (at multiple wavelength channels) for the same area on the surface of the Earth. The resulting dataset, called a data cube, is a stack of images in which each pixel (vector) (see Figure 9.0.1) has an associated spectral signature or finger-print that uniquely characterizes the underlying objects, and the resulting data volume typically comprises several GBs per flight. Usually the task involves implementing algorithms that automatically identify pure spectral signatures (normally called *endmembers*) from the input data cubes.

The recent use of HPC in hyperspectral imaging has offered opportunities to explore methodologies in areas such as multi-source data registration and mining [53] which previously looked too computationally intensive for practical applications due to the size of data files and high computational resource required. However, the problem of concurrently tiling of swathes and spectral bands to already allocated data cube while processing is in session has not been addressed in the literature. In this chapter we provide PGAS parallel implementation strategies for standard endmember extraction algorithms as well as storage and I/O of data cubes using PEXTA.

9.1 Data Partitioning Strategies using PEXTA

Data partitioning is usually the first step in any parallel image processing or analysis. In hyperspectral imaging, the dataset can be partitioned spectrally or spatially. In *spectral-domain partitioning*, the dataset is divided into data strips consisting of contiguous spectral wavelengths. Set of strips are assigned to a process in session. For this partitioning model, each pixel vector may be split among several processes as in-

dicated in Figure 9.1.1 and the communication cost associated with the computations based on spectral information is usually high [54].

Spatial-domain partitioning on the other hand divides the dataset into slices or slabs such that in each slice or slab the full spectral information associated with each pixel vector is entirely maintained. This partitioning is more amenable to parallel endmember extraction algorithms as these algorithms require the same function to be applied to groups of pixel vectors in the imagery data volume. Spatial-domain partitioning also supports code reusability. Normally it is suitable to reuse much of the code for the sequential algorithm in the design of its correspondent parallel version and spatial-domain partitioning lends itself to such reusability.

In representing spatial-domain partioning in the PEXTA model, we use 3-dimensional chunked extendible array $A_c[\rho_0][\rho_1][\rho_2]$ to represent the image cube. ρ_i is the number of chunk indices; thus $\rho_i = 1$ where $i = 1, 2$ and $\rho_0 = P$. P is the number of processes within a GA current session. Given a k-dimensional extendible array $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]...[\mathbb{U}_{k-1}^*]$, the chunk block $Q[\chi_0][\chi_1][\chi_2]$ is allocated such that χ_j is the chunk size along dimension j. χ_0 and χ_1 are computed as follow (see Figure 9.1.1):

$$\chi_i = \left\lceil \frac{\mathbb{U}_i^*}{P} \right\rceil \tag{9.1.1}$$

where

$$i = 0, 1 \text{ and } \chi_2 = \mathbb{U}_2^*$$

.

As illustrated in Figure 9.1.1, the value of χ_0 and χ_1 for process P_{P-1} are computed as $\mathbb{U}_i^* - \chi_0(P-1)$, for $i = 0, 1$. The spectral bands are not partitioned, hence $\chi_2 = \mathbb{U}_2^*$.

In reading the hyperspectral imagery data cube, the processes with the GA current session perform independent or collective reads of their respective hyperslab or slice. In the PEXTA model, the metadata file F_m corresponding to the O_2-Tree structure, is read into the local buffers of the processes.

Airborne hyperspectral imaging uses imaging spectrometers such as AVIRIS (Airborne Visible / Infrared Imaging Spectrometer) and HyMap (Hyperspectral Mapper). The sensors of such spectrometers capture one line image with all the bands at a time. The total number of line images that can be captured continuously at one time depends on the size of each line image and the available internal memory. Captured line images are stored temporarily in the internal memory before being stored on an external hard drive. Thus images are stored in swathes. The dimensional bound $\mathbb{U}_0^*$, $\mathbb{U}_1^*$ and $\mathbb{U}_2^*$ of an already allocated captured swathe is extended when new swathes are added. Ideally, endmember extraction processing techniques are supposed to apply to all swathes from the same area. For real-time analysis, the conventional collation of these swathes may incur an overhead. Our PEXTA model provides an alternate storage technique that allows endmember extraction processing algorithms to be applied to already stored swathes with aggregation.

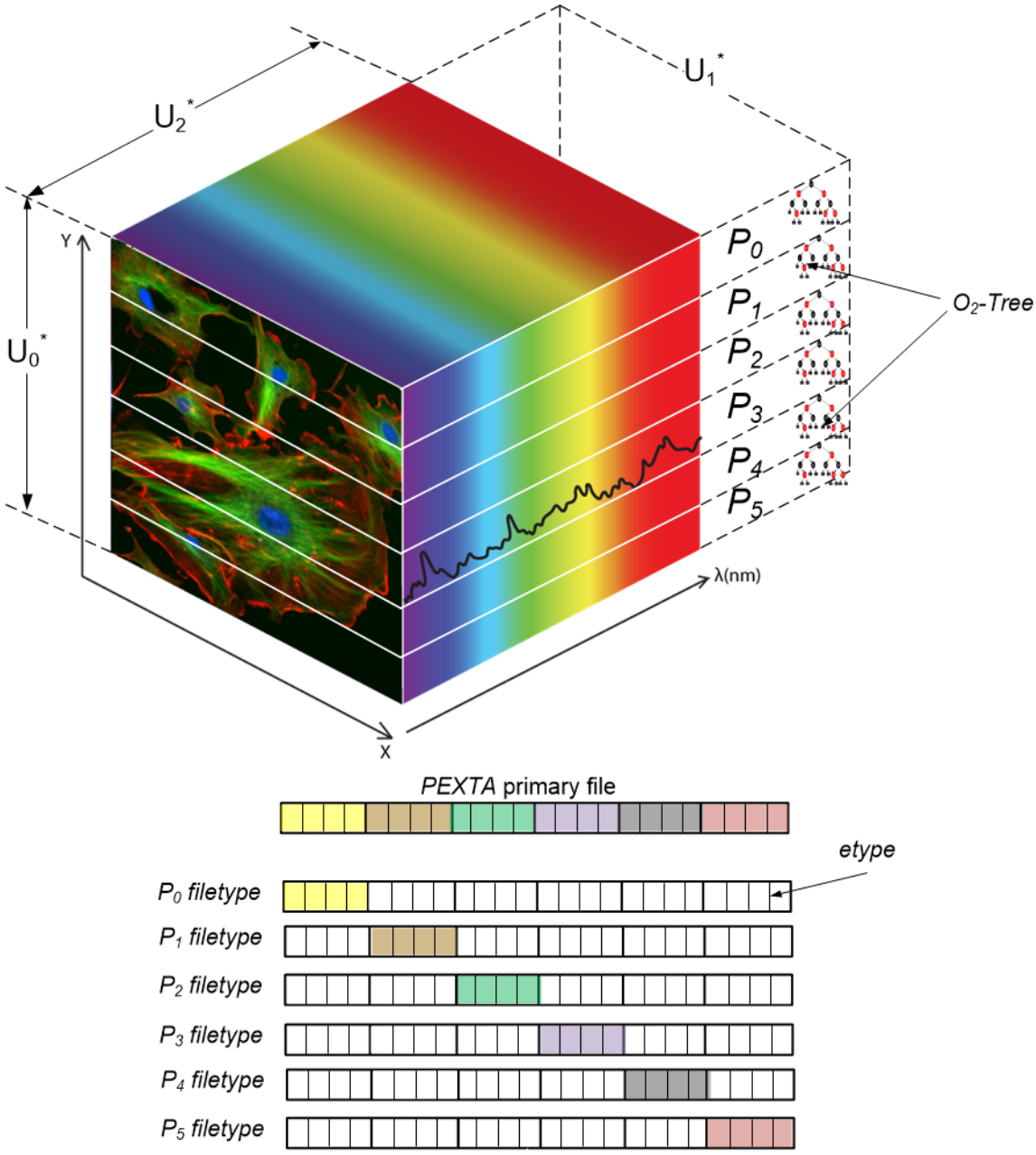


Figure 9.1.1: Spatial Partitioning for Hyperspectral Data Analysis

9.2 Parallel Dimensionality (Spectral Bands) Reduction

Spectral unmixing of hyperspectral datasets is made up of three stages: dimensional reduction, endmember selection and abundance estimation. The first step in any hyperspectral image analysis or processing is to reduce the number of spectral bands. Previous works have shown that a large number of spectral bands increases the complexity of endmember extraction algorithms [55, 56]. The accuracy of these algorithms is usually improved by focusing on relevant spectral bands or wavelength areas.

There are various techniques for reducing the spectral bands of hyperspectral datasets. These can classically be grouped into *feature extraction* and *feature selections* techniques. Feature extraction techniques combine or rearrange existing feature space into new features, e.g., maximum noise fraction (MNF) and principal component analysis (PCA).

In this thesis we explore the use of principal component transform (PCT) for the purposes of reducing the spectral bands of hyperspectral imagery datasets. PCT is a classic linear transformation approach and is used frequently for dimensionality reduction in pattern classification problems. It reduces redundancy and extracts residual information into small sets of features called principal components. PCT is a highly compute-intensive algorithm amenable to parallel implementation. Previous MPI implementations use the master-slave model of computation [57, 58]. We present a variant implementation using the PGAS model of shared memory computation. The PGAS parallel implementation provides data locality and granularity control for

easy MapReduce-like implementation [59]. This alleviates the task of circumventing communication overheads. Algorithm 8 illustrates our implementation. The eigenvalues of the covariance matrix must be computed in parallel. We left out the details of $CompTransp$ and $CompEv$ which are the computations of transpose matrix and eigenvectors respectively. We employ the Jacobi method [60] in computing eigenvalues and their corresponding eigenvectors. Other methods such as the Hotelling Power Method which compute eigenvalues iteratively can also be used.

Unlike PCT which only relies on spectral information, other techniques such as mathematical morphology integrate the spatial and spectral information to reduce the spectral bands. Mathematical morphology was originally designed for binary images, and then extended to both grayscale images [61] and hyperspectral images [62].

Algorithm 8: Parallel PCT of a Hyperspectral data cube A_c.

input : Hyperspectral Data File f or $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$
output : A_{pc} Reduced Hyperspectral Data with PC
begin

Initialise:
$PEXTA\_Init$;
$A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*] \leftarrow PEXTA\_Read(f)$;
$m \leftarrow 0$;
$Q^i \leftarrow GetLocalDim(A_c, rank)$;

/*Parallel Mean Vector Computation */;
for $x \leftarrow 0$ **down to** $\mathbb{U}_1^* - 1$ **do**
  for $y \leftarrow 0$ **down to** χ_i **do**
    for $\lambda \leftarrow 0$ **down to** $\mathbb{U}_2^* - 1$ **do**
      $m_l \leftarrow m_l + Q^i_{xy\lambda}$;
  $m_l \leftarrow m_l / Q^i_T$;
  $m \leftarrow m_l$;
  /*Q^i_T is the total number of pixel vectors*/

/*Parallel Covariance Matrix Computation */;
for $x \leftarrow 0$ **down to** $\mathbb{U}_1^* - 1$ **do**
  for $y \leftarrow 0$ **down to** χ_i **do**
    for $\lambda \leftarrow 0$ **down to** $\mathbb{U}_2^* - 1$ **do**
      $Q^i_{xy\lambda} \leftarrow Q^i_{xy\lambda} - m_l$;

$Q^{iT} \leftarrow CompTransp(Q^i)$;
$Q^c \leftarrow CompTransp(Q^i) \times Q^{iT}$;
/*Q^c is the Covariance Matrix*/

/*PCs Computation */;
$Q^{ev} \leftarrow CompEv(Q^c, \mathbf{m})$;
$Q^{pc} \leftarrow CompTransp(Q^{ev}) \times Q^i$;
/*Q^{ev} : Eigenvectors */

$A_{pc} \leftarrow Q^{pc}$;
return A_{pc}

9.3 Parallel Endmember Extraction Algorithms

Spectral unmixing is the separation of a pixel spectrum into pure endmembers and estimating the abundance of each endmember in every pixel. There are two types of unmixing techniques; linear and non-linear unmixing. In *linear spectral unmixing*, the collected spectra at the spectrometer can be expessed as a linear combination of endmembers weighted by their corresponding abundances. The pure components are assumed to be homogeneously distributed in separate patches within the field of view of the spectrometer. In the *non-linear spectral unmixing* however, the pure components are intimately mixed inside the pixel. The challenge is in the derivation of the nonlinear function. Nonlinear spectral unmixing requires detailed a priori knowledge about the materials. In this thesis, we only consider linear spectral unmixing. In linear mixture model, each $\mathbb{U}_2^*$-dimensional (using Figure 9.1.1) pixel vector of the hyperspectral data can be modelled using Equation 9.3.1 :

$$\mathbf{f}_i = \sum_{j=1}^{p} \boldsymbol{e}_j \cdot \Phi + \varepsilon \tag{9.3.1}$$

where $\boldsymbol{e}_j$ denotes the spectral response of an endmember, Φ is a scalar value that represents the fractional abundance of the endmember $\boldsymbol{e}_j$, p is the total number of endmembers, and ε is a noise or error term. Normally $p \ll \mathbb{U}_2^*$, thus the solution of Equation 9.3.1 relies on the correct determination of a set $\mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$ and their respective abundance fractions $\Phi = \{\Phi\}_{j=1}^{p}$ at each pixel $\mathbf{f}_i$.

Ideally, in dimensionality reduction, we identify the subspace spanned by the columns of matrix $\boldsymbol{E}$. Endmember extraction is the determination of the actual columns of $\boldsymbol{E}$.

The abundance estimation is the identification the endmember vector of proportions Φ of each pixel. There are several algorithms developed for the purpose of automatic spectral endmember extraction. The pixel purity index (PPI) [63], is perhaps the most popular endmember extraction algorithm due to its availability in commercial software packages such as ENVI [64]. PPI algorithm calculates a spectral purity score for each $\mathbb{U}_2^*$-dimensional pixel in the original data by generating random unit vectors, called skewers, so that all pixel vectors are projected onto the skewers and the ones falling at the extremes of each skewer are counted. After many repeated projections to different *skewers*, those pixels that count above a certain threshold are considered pure endmembers (see Figure 9.3.1).

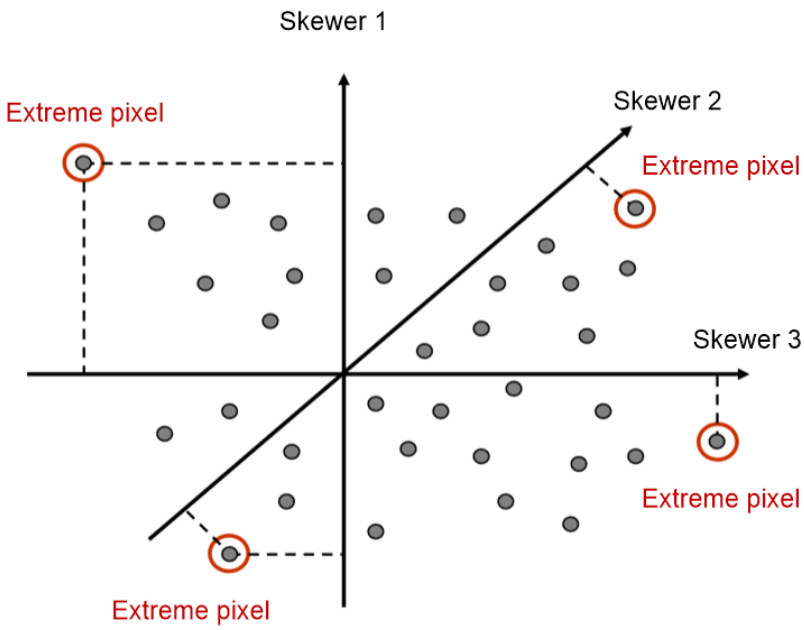


Figure 9.3.1: PPI in two dimensional space

Another widely used endmember extraction algorithm is N-FINDR [65]. This begins with a random initial selection of endmembers and iteratively looks for the set of

pixels that maximizes the volume of the simplex defined by the selected endmembers. This is computed for every pixel in each endmember position by replacing that endmember and finding the resulting volume. For an increase of volume, the previous pixel is replaced by the current pixel. The procedure is repeated until there are no more endmember replacements.

The last endmember extraction alogrithm we explore is Automated Morphological Endmember Extraction (AMEE) [66]. Unlike PPI and N-FINDR which tend to neglect the existing spatial correlation between pixels (i.e., they only rely on spectral band information), AMEE exploits both the spectral and spatial information within the hyperspectral dataset. The algorithm extends mathematical morphology [67] (which only applies to grayscale and binary images) to multispectral images such that spatial characteristics or properties are kept. AMEE combines both morphological *dilation* and *erosion* operations. The dilation operation selects the purest (maximum) pixel, erosion selects the most highly mixed pixel (minimum), in a certain spatial neighborhood. The linear distance between the maximum and the minimun of a pixel defines a value called morphological eccentricity index (MEI) that depicts the quality of the pixels in terms of its spectral purity. In the proceeding sections, we describe PGAS parallel implementation of these three widely used endmember extraction algorithms.

9.3.1 Parallel Pixel Purity Index-Like Preprocessing

The first step after reducing the spectral dimension in PPI algorithm is to generate the *skewers*. The skewers are $\mathbb{U}_2^*$-dimensional unit vector that passes through the

data.

Algorithm 9: Parallel PPI using GA.

input : Hyperspectral Data File f or $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$, I_{max}, p, K
output : $\tau[p] : \mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$
begin

Initialise:
$PEXTA\_Init$;
$A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*] \leftarrow PEXTA\_Read(f)$;
$c \leftarrow 0, chk_e \leftarrow 0,\ chk_{bool} \leftarrow false$;
$A_r \leftarrow ReduceDim(A)$;
$Q^i \leftarrow GetLocalDim(A_r, rank)$;
/*Generation of Skewers */;
for $r \leftarrow 0$ **down to** S_n **do**
  $sk_r \leftarrow GenerateSkewer()$;

/*Parallel Spectral Vectors Projections */;
for $iter \leftarrow 0$ **down to** S_n **do**
  for $x \leftarrow 0$ **down to** $\mathbb{U}_1^* - 1$ **do**
    for $y \leftarrow 0$ **down to** χ_i **do**
      $w_1 \leftarrow \left(\frac{\mathbf{f}(x,y) \cdot \mathbf{sk}[iter]}{\|\mathbf{sk}[iter]\|^2}\right) \times \mathbf{sk}[iter]$;
      $w_2 \leftarrow \mathbf{f}(x,y) - w_1$;
      if $\|w_2\| > chk_e$ **then**
        $chk_e \leftarrow \|w_2\|,\ x_i[iter] \leftarrow x,\ y_i[iter] \leftarrow y$;
  if $iter > 0$ **then**
    for $j \leftarrow 0$ **down to** $iter$ **do**
      if $x_i[j] = x_i[j-1] \wedge y_i[j] = y_i[j-1]$ **then**
        $chk_{bool} \leftarrow true$;
  if $chk_{bool} = false$ **then**
    $ske[c] \leftarrow chk_e,\ x_i^a[c] \leftarrow x_i[iter],\ y_i^a[c] \leftarrow y_i[iter],\ c \leftarrow c+1$;
  $chk_{bool} \leftarrow false$;

/*Collective Merge and Sort of Extrema Sets*/;
$ske_a \leftarrow MergeSort(ske)$;

/*Selecting the first p SKEs */;
for $j \leftarrow 0$ **down to** p **do**
  $\tau[j] \leftarrow ske_a[j]$;

return τ

Every pixel vector is projected onto each skewer sk_j^m, and those pixel vectors that are at its extreme positions form an extrema set, denoted by $S_{extreme}(sk_j^m)$. Although different skewers generate a different extrema set $S_{extreme}(sk_j^m)$, it is very likely that some pixel vectors may appear in more than one extrema set. We define the following expression to select unique extrema set per skewer:

$$I_S(r) = \begin{cases} 1, & \text{if } r \in S \\ 0, & \text{otherwise} \end{cases} \quad and \; N_{PPI}(r) = \sum_j I_{S_{extreme}(sk_j^m)}(r) \tag{9.3.2}$$

In algorithm 9, this criterion is represented by the use of chk_{bool}. The details of $GetLocalDim()$, $ReduceDim()$ and $GenerateSkewer()$ are left out of the algorithm to keep it simple. The corresponding pixel vectors $\mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$ that produce the first p largest values of $N_{PPI}(r)$ are then selected

9.3.2 Parallel N-FINDR-Like Endmember Selection

In N-FINDR, the spectral dimension is reduced to p, the expected number of endmembers. After reducing the spectral dimension, a set of p pixel vectors are selected and their corresponding simplex volume is computed. To maximize the volume, the simplex volume is computed for every pixel vector in each endmember position by replacing that endmember and recomputing the volume. If this results in an increased volume, the pixel replaces the endmember. This process is repeated until no replacement of endmember is left. Suppose $\{\mathbf{e}\}_{j=1}^{p}$ denotes the set of endmembers,

the simplex volume can be computed by:

$$V(e_0, e_1, ..., e_{p-1}) = \frac{\left\| det \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ e_0 & e_1 & e_2 & \dots & e_{p-1} \end{bmatrix} \right\|}{(p-1)\,!} \tag{9.3.3}$$

Algorithm 10: Parallel N-Findr using GA.

input : Hyperspectral Data File f or $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$, I_{max}, p, K
output : $\tau[p] : \mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$
begin
Initialise:
$PEXTA\_Init$;
$A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*] \leftarrow PEXTA\_Read(f)$;
$A_r \leftarrow ReduceDim(A)$;
$Q^i \leftarrow GetLocalDim(A_r, rank)$;
/*Generation Initial Endmembers 9.3.3 */;
for $j \leftarrow 0$ **down to** p **do**
  $\mathbf{e}_j \leftarrow RandomSelect()$;
/*Compute Initial Volume using Equation*/;
$V(e_0, e_1, ..., e_{p-1}) \leftarrow V(\mathbf{e}) \leftarrow ComputeVolume(\mathbf{e})$;
/*Parallel Simplex Volume Maximization */;
for $x \leftarrow 0$ **down to** $\mathbb{U}_1^* - 1$ **do**
  for $y \leftarrow 0$ **down to** χ_i **do**
    for $iter \leftarrow 0$ **down to** p **do**
      $V_c(\mathbf{e}) \leftarrow V(InsertReplace(\mathbf{e},\ iter,\ f(x,y)))$;
      if $V_c(\boldsymbol{e}) > V(\boldsymbol{e})$ **then**
        $\mathbf{e} \leftarrow InsertReplace(\mathbf{e},\ iter,\ f(x,y))$;
        $V(\mathbf{e}) \leftarrow V_c(\mathbf{e})$;
/*Collective Determine the Largest $V(\boldsymbol{e})$*/;
$V_a(\mathbf{e}) \leftarrow ComputeLargestVol(V(\mathbf{e}))$;
/* $\boldsymbol{e}$ is the Endmembers */;
for $j \leftarrow 0$ **down to** p **do**
  $\tau[j] \leftarrow \mathbf{e}[j]$;
return τ

An appropriate selection of initial endmembers (using *RandomSelect*() in Algorithm 10) is critical to produce correct results and also to speed up convergence.

9.3.3 Parallel Spatial/Spectral Endmember Extraction

The two basic morphological operations in the AMEE algorithm are erosion and dilation. Let $f(x, y)$ denote a pixel vector at the spatial coordinate (x, y) of a hyperspectral data cube $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$. The erosion of f by structuring element (sometimes called kernel) K is the *minimum* pixel vector in the spatial neighbourhood of $f(x, y)$ defined by K. This is represented as :

$$(f \ominus K)(x, y) = \min_{(s,t)\in k}\{f(x + s, y + t) - k(s, t)\} \tag{9.3.4}$$

where $\ominus$ is the erosion operator and $k(s, t)$ denotes the weight associated to the different elements of the kernel. The minimum is determined by using distance measures from the centroid of the kernel. This can be done using either *cumulative distance* or simple *distance measures* to the centroid:

$$Cumulative\ Distance : D(f(x, y), K) = \sum_s \sum_t dist(f(x, y), k(s, t)), \forall, s, t \in K$$

$$Distance\ to\ Centroid : D(f(x, y), K) = dist(f(x, y), C_k), C_k = \frac{1}{\mathbb{U}_2^*} \sum_s \sum_t k(s, t), \forall, s, t \in K$$

The dilation of f by K on the other hand can be expressed as the *maximum* pixel vector. This can be expressed as:

$$(f \oplus K)(x, y) = Max_{(s,t)\in k}\{f(x - s, y - t) + k(s, t)\} \tag{9.3.5}$$

where $\oplus$ is the dilation operator.

The kernel K is moved through all the pixels of f, calculating the minimum and maximum pixel at each K neighbourhood, using morphological erosion and dilation described above. At each pixel, MEI is calculated using

$$MEI(x,y) = SAD[(f \ominus K)(x,y), (f \oplus K)(x,y)] \tag{9.3.6}$$

SAD is the spectral angular distance computed as the consine of the vector dot product divided by the product of the norms. If we denote SAD by θ, then $SAD[x,y]$ can be defined as

$$\theta(x,y) = cos^{-1}\left(\frac{\sum_{i=1}^{\mathbb{U}_2^*} x_i y_i}{\sqrt{\sum_{i=1}^{\mathbb{U}_2^*} x_i^2} \times \sqrt{\sum_{i=1}^{\mathbb{U}_2^*} y_i^2}}\right) \tag{9.3.7}$$

This iterative process is continued until either the maximum threshold set is reached or the kernel goes through all the requisite pixels. The set $\mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$ is formed by selecting the pixels with highest association scores in MEI.

In the parallel implementation, we first initialize the global array by reading the hyperspectral data file using PEXTA utilities. We then reduce the spectral dimension by employing Algorithm 8. The local process dimensions of the hyperspectral data cube is retrieved into local buffers. Morphological erosion, dilation, spectral angular distance and MEI of each enclosed pixel (enclosed by K) are computed in parallel. The MEIs from processes are collectively merged and sorted and the first highest p MEIs are selected as $\mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$.

Algorithm 11: Parallel AMEE using GA.

input : Hyperspectral Data File f or $A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*]$, I_{max}, p, K
output : $\tau[p] : \mathbf{E} = \{\mathbf{e}\}_{j=1}^{p}$
begin

Initialise:
$PEXTA\_Init$;
$A[\mathbb{U}_0^*][\mathbb{U}_1^*][\mathbb{U}_2^*] \leftarrow PEXTA\_Read(f)$;
$sad \leftarrow 0$;
$c \leftarrow 0$;
$d_n \leftarrow 0$;
$d_d \leftarrow 0$;
$A_r \leftarrow ReduceDim(A)$;
$Q^i \leftarrow GetLocalDim(A_r, rank)$;

/*Parallel Morphological Erosion and Dilation */;
for $x \leftarrow 0$ **down to** $\mathbb{U}_1^* - 1$ **do**
  for $y \leftarrow 0$ **down to** χ_i **do**
    $x_i \leftarrow (f \ominus K)(x, y)$ $y_i \leftarrow (f \oplus K)(x, y)$

    /*SAD Computation */;
    for $\lambda \leftarrow 0$ **down to** $\mathbb{U}_2^* - 1$ **do**
      $d_n \leftarrow d_n + x_{i\lambda} \times y_{i\lambda}$;
      $d_d \leftarrow d_d + Sqrt(x_{i\lambda}^2) \times Sqrt(y_{i\lambda}^2)$;

    $sad \leftarrow cos^{-1} \dfrac{d_n}{d_d}$;
    $mei[c] \leftarrow sad$;
    $c \leftarrow c + 1$;

/*Collective Merge and Sort of MEIs */;
$mei_{ms} \leftarrow MergeSort(mei)$;

/*Selecting the first p MEIs */;
for $i \leftarrow 0$ **down to** p **do**
  $\tau[i] \leftarrow mei_i$;

return τ

9.4 Experimental Results

This section provides experiments conducted to determine effectiveness of the parallel algorithms in terms of performance gains and accuracy in the analysis of real

hyperspectral datasets. Then, we provide an overview of the hyperspectral image datasets used. A detailed computational cost–performance analysis of the parallel algorithms is then presented.

9.4.1 Parallel Computing Platform

All experiments in this section were conducted in a cluster (called ZA-Wits-Core Cluster) The cluster consists of 16 machines, with a total of 190 cores. Each node has a minimum of 16GB of RAM and a maximum of 256GB. The cluster has four Intel storage servers of roughly 120TB of storage. The cluster runs Scientific Linux 6.3 and EMI-3. It runs Maui/Torque to control local access, as well as a UI machine for grid access.

9.4.2 Hyperspectral Datasets

We used three different hyperspectral datasets for our experimental validations. The datasets are from NASA AVIRIS flight sensors and NASA EO-1 satellite (Hyperion sensor) acquisitions. All imagery datasets have extensive ground-truth information available, thus allowing us to validate the performance of parallel algorithms in terms of performance and accuracy.

9.4.2.1 Botswana Okavango Delta Dataset

This is a sequence of data over the Okavango Delta, Botswana in $2001 - 2004$. The EO-1 sensor acquires the dataset at 30m pixel resolution over a 7.7 km strip in 242 bands covering the $400 - 2500nm$ portion of the spectrum in $10nm$

windows. The data was already preprocessed by University of Texas at Austin (UT) Center for Space Research. Noisy bands that cover water absorptions features were removed, leaving 145 bands. The candidate feature bands included $10-55, 82-97, 102-119, 134-164$ and $187-220$nm. The ground truth consisted of 14 identified classes representing in seasonal swamps, occasional swamps, and drier woodlands located in the distal portion of the Delta. The ground truth observations are represented in Table 9.1 and an image of the Okavango Delta is presented in Figure 9.4.1.

No.	Class	Number of Samples
1	Water	270(8.31%)
2	Hippo Grass	101(3.09%)
3	Floodplain Grasses 1	251(7.74%)
4	Floodplain Grasses 2	215(6.63%)
5	Reeds	269(8.27%)
6	Riparian	269(8.27%)
7	Firescar	259(7.98%)
8	Island Interior	203(6.26%)
9	Acacia Woodlands	314(9.67%)
10	Acacia Shrublands	248(7.65%)
11	Acacia Grasslands	305(9.38%)
12	Short Mopane	181(5.56%)
13	Mixed Mopane	268(8.27%)
14	Exposed Soils	95(2.92%)

Table 9.1: Ground-truth Information for Botswana Hyperion Data

Classes 3 and 4 are both floodplain grasses that are seasonally flooded but differ in their hydroperiod (the amount of time flooding). Classes 9 through 11 represent different mixtures of acacia woodlands, shrublands, and grasslands and are named according to the dominant class.

Figure 9.4.1: An image of Okavango Delta, Botswana

9.4.2.2 AVIRIS Kennedy Space Center Dataset

This dataset spans over the Kennedy Space Center (KSC), Florida. It is a well known hyperspectral dataset used in hyperspectral imaging literature. It has 224 bands of 10nm width ranging from $400 - 2500$nm with spatial resolution of $18m$. The dataset was preprocessed by removing water absorption and low SNR bands, leaving 176 bands for the analysis. For classification purposes, 13 classes representing the various land cover types that occur in this environment were defined for the site (see Table 9.2).

9.4.2.3 AVIRIS Salinas Valley Dataset

The hyperspectral dataset for this scene was collected over the Valley of Salinas in Southern California. The area covered comprises of 512 lines by 217 samples with 224 spectral bands. Preprocessing involved the removal of water absorption and

No.	Class	Number of Samples
1	Scrub	761(14.6%)
2	Willow Swamp	243(4.66%)
3	Cabbage Palm Hammock	256(4.92%)
4	Cabbage Palm/Oak Hammock	252(4.84%)
5	Slash Pine	161(3.07%)
6	Oak/Broadleaf Hammock	229(4.38%)
7	Hardwood Swamp	105(2.00%)
8	Graminoid Marsh	431(8.27%)
9	Spartina Marsh	520(9.99%)
10	Cattail Marsh	404(7.76%)
11	Salt Marsh	419(8.04%)
12	Mud Flats	503(9.66%)
13	Water	927(17.8%)

Table 9.2: Ground-truth Information for Kennedy Space Center AVIRIS

noisy bands. This resulted in 186 spectral bands ranging from 0.4 to 2.5μm with nominal spectral resolution of 10nm and 16-bit radiometric resolution. The dataset include vegetables, bare soils and vineyard felds. Salinas ground-truth contains 16 classes presented in Table 9.3. Figure 9.4.2(a) shows a false color composition of the scene and Figure 9.4.2(b) shows the available ground-truth regions for this scene, which cover about two thirds of the entire Salinas scene.

9.4.3 Experiments Conducted

We conducted a number of comparative experimental analysis of the different three endmember extraction techniques for hyperspectral image classification. The main criteria for our analysis are accuracy and processing time performance. The accuracy for each experiment was measured by the ratio of endmember abundance of each ground-truth class in each dataset to the actual ground-truth percentage. In order to validate the experimental results provided by other implementations, we used

No.	Class	Number of Samples
1	Brocoli Green Weeds 1	2009(3.71%)
2	Brocoli Green Weeds 2	3726(6.88%)
3	Fallow	1976(3.65%)
4	Fallow Rough Plow	1394(2.58%)
5	Fallow Smooth	2678(4.95%)
6	Stubble	3959(7.31%)
7	Celery	3579(6.61%)
8	Grapes Untrained	11271(20.82%)
9	Soil Vinyard Develop	6203(11.46%)
10	Corn Senesced Green Weeds	3278(6.06%)
11	Lettuce Romaine 4wk	1068(1.97%)
12	Lettuce Romaine 5wk	1927(3.56%)
13	Lettuce Romaine 6wk	916(1.69%)
14	LLettuce Romaine 7wk	1070(1.98%)
15	Vinyard Untrained	7268(13.43%)
16	Vinyard Vertical Trellis	1807(3.34%)

Table 9.3: Ground-truth classes for the Salinas scene

ground-truth measurements that is usually associated with the freely available dataset from AVIRIS or Hyperion sensors as our benchmark.

Several different combinations were tested for the considered parallel PGAS (with PEXTA) implementation and and MPI (with parallel HDF5). In order to assess the classification accuracy of our endmember extraction implementation, we performed the following experiments:

1. The ground-truth dataset and the original hyperspectral data were fed to ENVI software package to generate class labels for each pixel. This was used as baseline for our comparative analysis.

2. We then run our parallel implementation of PPI, *N*-FINDR and AMEE using their original configurations, i.e., using a dimension reduction technique to reduce the dimensionality of the input data, thus obtaining a set of spectral

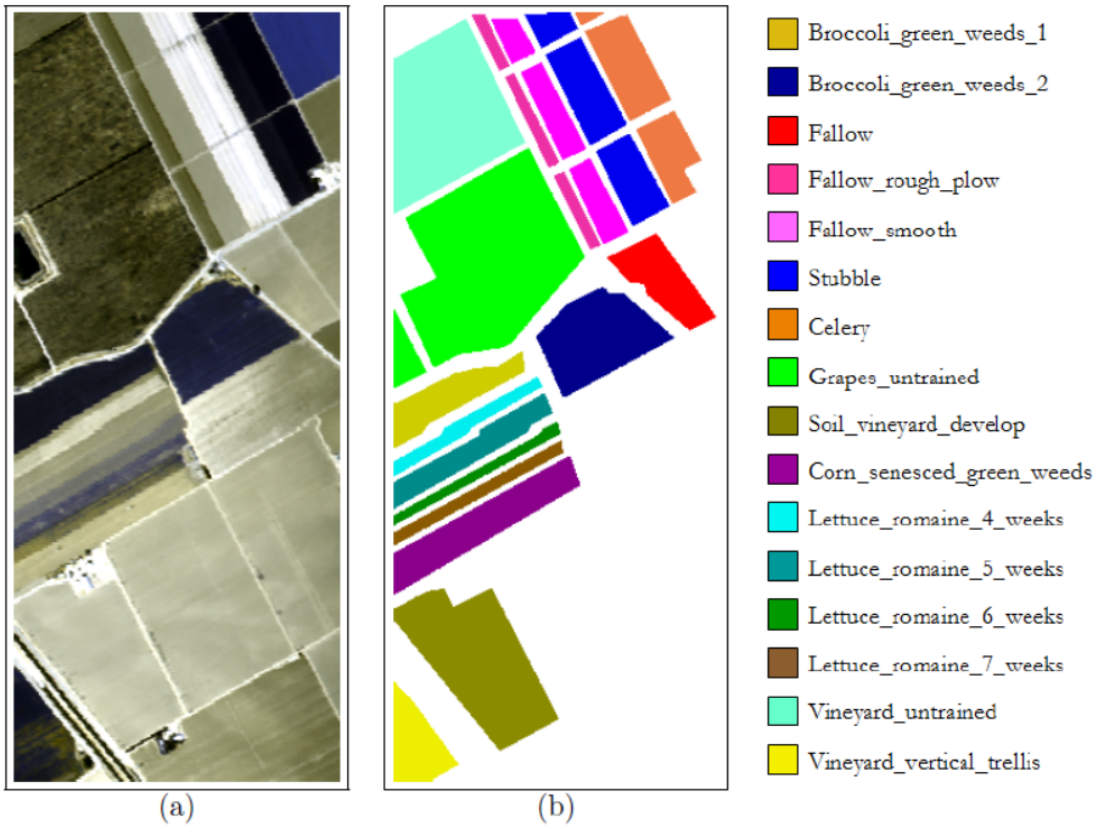


Figure 9.4.2: AVIRIS hyperspectral image comprising several agricultural fields in Salinas Valley

endmembers in all cases.

3. We compute the abundance estimation of each endmember using $\Phi = \{\Phi\}_{j=1}^{p}$ which is then used to compute the accuracy of abundance match.

4. The computational time for each algorithm implementation is then compared with the parallel HDF5 implementation.

9.4.4 Analysis and Discussion of Results

In Table 9.4 and 9.5, we report the individual and overall endmember extraction accuracies produced by the parallel PPI, *N*-FINDR and AMEE implementations respectively, after using the same algorithm configurations for the Botswana Okavango Delta and AVIRIS Salinas Valley datasets. We utilize and modified the C source code found in [68] for parallel HDF5 with MPI implementations whiles the GA implementation is based on Algorithms 9, 10 and 11. We utilize 160 processes in both experiments.

	GA with PEXTA			MPI with PHDF5		
	PPI	*N*-FINDR	AMEE	PPI	*N*-FINDR	AMEE
Processing Time(s)	= 25.21	= 29.53	= 26.34	= 28.11	= 35.81	= 30.42
Classes						
Water	242	307	278	235	300	278
Hippo	76	73	93	76	54	109
Floodplain Grs. 1	273	213	241	229	212	260
Floodplain Grs. 2	243	247	205	239	167	224
Reeds	293	307	279	293	303	280
Riparian	298	300	277	289	309	279
Firescar	280	295	266	223	301	270
Island Interior	179	231	195	237	235	214
Acacia Wd. lands	295	282	322	281	277	325
Acacia Sb. lands	224	280	255	225	286	259
Acacia Grs. lands	324	331	314	284	345	316
Short Mopane	201	207	171	146	211	192
Mixed Mopane	297	297	259	231	298	276
Exposed Soils	113	122	88	60	144	105
Overall Accuracy(%)	90.11	86.82	96.43	87.9	83.94	95.84

Table 9.4: Per-Class and overall percentages of correctly classified pixels in the Botswana Hyperion Dataset

In both implementations, AMEE produced higher classification accuracies than PPI and *N*-Findr. This establishes the fact that incorporating spatial information

to endmember extraction increases accuracy. Tables 9.4 and 9.5 also report the execution times measured for the different algorithms using 20 processes of the ZA-Wits-Core Cluster. Processing times increase as the sizes of the dataset and the individual classes increase.

	GA with PEXTA			MPI with PHDF5		
	PPI	*N*-FINDR	AMEE	PPI	*N*-FINDR	AMEE
Processing Time(s)	= 18.54	= 40.21	= 33.79	= 51.49	= 100.63	= 77.55
Classes						
Brocoli Green Wds. 1	1737	2405	2101	2296	2383	2124
Brocoli Green Wds. 2	960	1028	827	1105	1127	863
Fallow	2227	1611	2067	2353	2415	2125
Fallow Rough Plow	1645	1109	1308	1672	965	1521
Fallow Smooth	2884	2315	2582	2942	2300	2822
Stubble	4243	3560	4053	4249	3572	4063
Celery	3816	3329	3474	3938	3180	3721
Grapes Untrained	11005	10874	11176	11557	10921	11418
Soil Vinyard Develop	5965	6550	6119	6555	6622	6326
Corn Sns. Green Wds.	3058	3614	3189	2975	3651	3425
Lettuce Romaine 4wk	1322	1420	985	807	1545	1184
Lettuce Romaine 5wk	2164	2238	2021	1660	2282	2033
Lettuce Romaine 6wk	1189	1202	997	557	1347	1022
LLettuce Romaine 7wk	1338	763	1168	1374	1451	1205
Vinyard Untrained	7505	6883	7368	7646	7701	7406
Vinyard Vertical Trellis	2112	1493	1913	2182	2213	1956
Overall Accuracy(%)	92.11	89.45	97.08	89.99	87.42	95.92

Table 9.5: Per-Class and overall percentages of correctly classified pixels in the AVIRIS Salinas Valley Dataset

We investigate the scaling of parallel endmember extraction implementations. Figure 9.4.3 illustrates their speedup relative to serial implementations. The total number of processes on the ZA-Wits-Core Cluster is 190, so we used an interval of 20, utilizing 180 processes. Results shows that in the MPI implementations, although the complexity of the eigenvector calculations is related to the number of spectra used in the problem, the split and merge are also highly dependent on the inherent

complexity of the input data. In the GA implementation, we can observe that the PPI and AMEE algorithms result in closer-to-linear speedups. The resulting

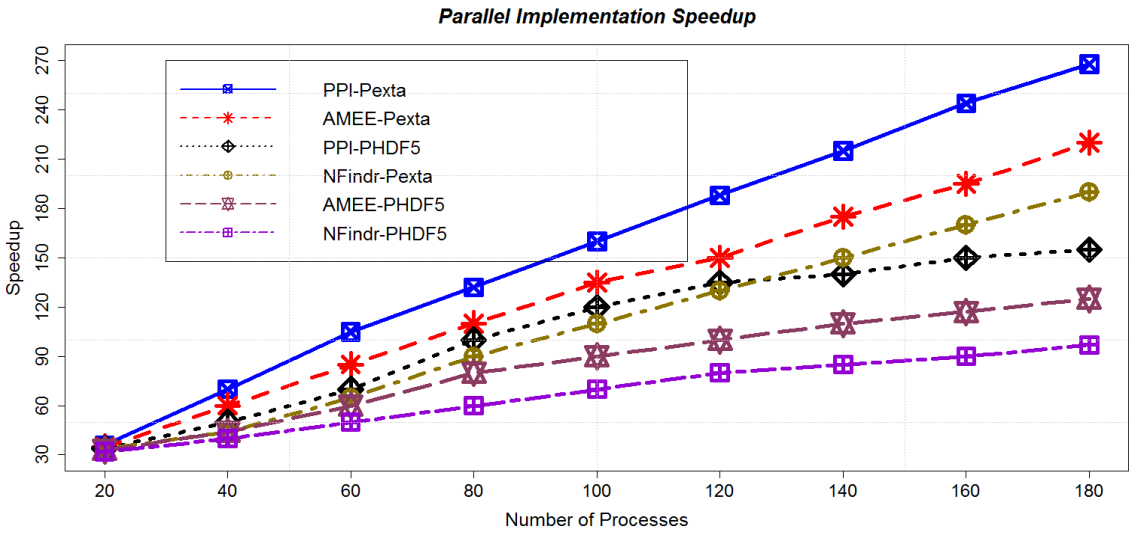


Figure 9.4.3: Speedup of Parallel Implementations

execution times for these algorithms for the experiment in Figure 9.4.3 are illustrated in Figure 9.4.4.

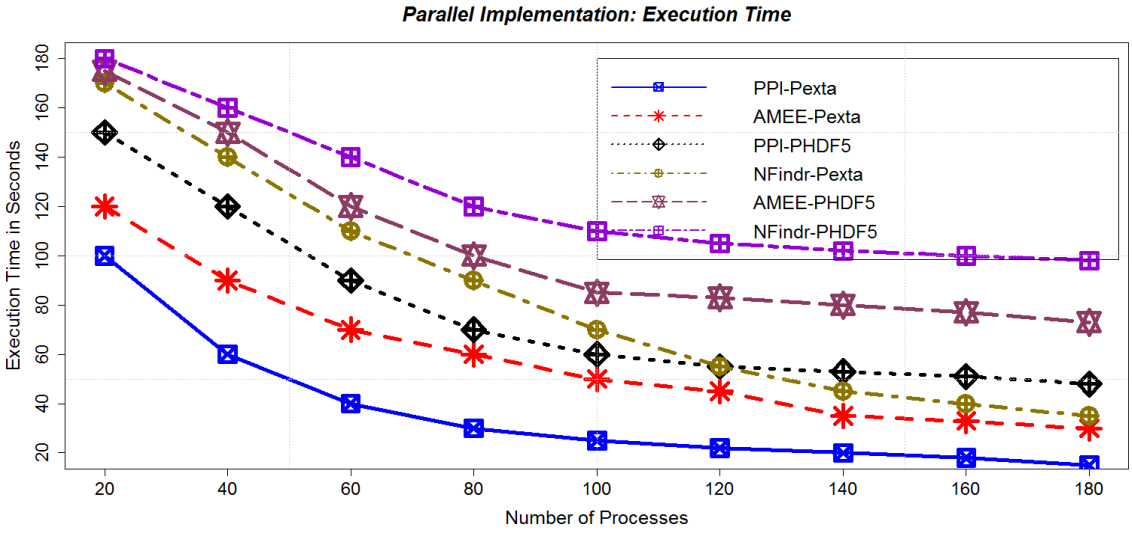


Figure 9.4.4: Execution Times of Parallel Implementations

10

Conclusion and Future Directions

This chapter concludes the thesis and gives roadmap for future work

Contents

10.1 Summary of Research Contributions

This thesis provides the implementation of the extendible chunked dense arrays using a computed access mapping function of the array elements. The mapping function of the extendible array allows for arbitrary extension of the array in any dimension without reallocation of already allocated array blocks or chunks. Two schemes have been presented: an extendible array that is entirely resident in memory and one that resides on disk.

The organisation of extendible arrays using such a mapping function is highly appropriate for most scientific datasets where the model of the data is perceived to be in the form of large array files. Such a property is suitable for high speed and parallel processing using the Global Array Toolkit. The disk resident extendible chunk array scheme is implemented to conform to the various programming models within the Global Array Toolkit for easy integration. The scheme can replace the Disk Resident Array in the GA, since the GA does not support extendibility. We refer to this scheme as the parallel extendible chunked dense arrays (PEXTA).

10.1.1 Chunked Extendible Arrays

In an effort to reduce the computational cost during interruptible expansions of extendible array, we provide chunking techniques that decomposes the array elements into smaller arrays called chunks or blocks. Chunking provides an improved performance in arrary elements access operations.

Chunking the array gives some additional advantages. Firstly, it gives contiguous

storage allocations for the elements of the chunks. Secondly, when arrays are allocated onto secondary storage, I/O can be made in multiples of the chunk size. The allocation is done in chunks as opposed to the single index extension.

The entry into the respective axial-vectors of extendible arrays is based the chunks instead of the single elements. This implies that accessing an element given its k-dimensional indices requires two translations of indices and two mapping schemes. The input indices are first translated into chunk indices. The chunk indices is used to retrieve the required chunk. With the base address of the chunk, the address of input indices is only a displacement within the chunk. This can be done by using a row-major sequence order (or column major). If the chunk size is 2^n where $n \geq 2$, then the Z-sequence order (also called the Morton sequence order) or Peano-Hilbert scan order can be used.

10.1.2 Axial Vectors as Memory Resident O_2-Tree

The search through axial-vectors is normally done by simple binary search or interpolation search. To improve the search performance, we introduced a new structure called O_2-Tree for maintaining the axial-vectors in memory. The O_2-Tree is an augmented Red-Black Tree with data records stored only at the leaf nodes. We store the axial vectors of expansions as records of the leaf nodes.

The internal nodes only stores separator keys, that allow all searches to terminate at the leaf-node. At this terminal leaf node, the requisite axial vector is identified by performing sequential or binary search. During traversal of the tree, on arrival at an internal node, a left branch is taken if the search key is less than the key at that

node; otherwise a right branch is taken. Every other property of the Red-Black tree is preserved through a standard left and right rotation.

Every expansion of a dimension results in an update of the corresponding O_2-Tree. If an application has already constructed the memory-resident O_2-Tree, then the linear address of an element of the corresponding array file, given its k-dimensional coordinates, will be computed using the computed mapping function.

The height of the O_2-Tree can be considerably reduced by keeping multiple axial-vectors at a leaf-node. In chunked extendible array files, this is even reduced further as the size of axial-vectors is significantly reduced.

10.1.3 Block Allocation Schemes and Access Schemes

The problem of determining optimal chunk size and shape has been extensively addressed in literature. Arrays chunks are usually stored on disk as I/O blocks. Chunking is usually parameterized by chunk size and chunk shape. The chunk size is the number of elements that can be contained in a chunk. The chunk shape is the length of the chunk in the direction of each dimension. The optimization goal is to find the optimal chunk size and optimal chunk shape so that the number of accessed disk blocks or I/O blocks per query is minimized. The chunk shape is chosen such that it minimizes the average number of accessed blocks. Large chunk size may cause unwanted data to be read for a query whiles small chunk size requires more accesses to retrieve all chunks needed for a query. Determining an optimal chunk size in an array file in the context of HDF5 is left for future work.

Another factor that may influence the performance of data element access is the allocation scheme within the chunks. We explore the use space-filling curves within the chunks. One advantage of using space-filling curves is that it reduces the number of dimensions to one, while preserving neighbourhood relations. We implemented the Hilbert curve within the chunks. Two algorithms were utilized, the Tree Representation of Hilbert curve and the Butz algorithm. Although there was not much significant performance difference between the use of Hilbert curve row-major sequence order, Hilbert curve is sometimes applicable in some applications that require neighbourhood preservation. It is not clear if the Hilbert space filling curve gives the minimum average distance of pairs of cells. This require further future work.

10.1.4 Mapping Functions for HDF5 and NetCDF

Scientific datasets are usually stored in scientific data formats such as NetCDF, HDF5, FITS etc. These file formats provide scientist the ability to store and retrieve multidimensional arrays, which are the building block of scientific data exploration. One of the challenges that is common to almost all scientific data formats is that their interfaces do not support dynamic extensions of array bounds during computation.

In this thesis, we have introduced the two most widely used scientific file formats namely NetCDF and HDF5, and have provided mapping techniques for translating datasets in these formats onto the chunked extendible array file structure. The mapping implementations provide routines for reading in and writing out multidimensional dataset for these formats. Applications may continue to use the HDF5 and NetCDF interface directly. However to enable extendibility in multiple dimensions,

datasets in these formats need to be read into our data model. Once the application is done with extensions, data can be written out into these data formats. The application to FITS files is yet to be explored.

10.1.5 PEXTA and its Applications

The message passing programming model is frequently used because of its portability but some problems or applications are too complex to be programmed with it in order to maintain balanced computations and avoid redundant computations. Although shared memory programming model is not portable and lacks full control over interprocess data transfer, it simplifies coding and very easy to code in. PGAS programming model combines the good features of both models leading to simple coding and efficient execution.

We envisage that PGAS models will be the predominant model for exascale computing. We chose to implement Parallel Extendible Chunked Dense Array for the Global array toolkit. The GA provides global memory address space that is logically partitioned and a portion of it is local to each process or thread.

Parallel Extendible Chunked Dense Array (PEXTA) for GA provides the necessary APIs for explicit transfer between the memory resident global array and its secondary storage counterpart but also allows the persistent array to be extended on any dimension without compromising on the access time of an element or sub-array elements. It serves as a drop-in API for the Disk Resident Array (DRA) of GA. The DRA library extends the GA NUMA programming model to disk. PEXTA

enables the GA persistant files to extended without interfering datasets in currently in memory or session.

We finally applied the PEXTA I/O routines to hyperspectral image processing. This processing involves implementing algorithms that automatically identifies pure spectral signatures (or endmembers). During hypectractral data aquisition, hundreds or even thousands of measurements (at multiple wavelength channels) are collected for the same area on the surface of the Earth. Recently, there has been the quest for real time analysis and rapid inspection of massive data archives and extraction of information. This has introduced the the use of HPC technologies, such as parallel and distributed computing in the processing hyperspectral imagery datasets. However, the problem of concurrently tiling of swathes and spectral bands to already allocated data cube while processing is in session has not been addressed in the scientific community. We use the routines of PEXTA to address this problem. We tested PEXTA and GA on the most frequently used endmember extraction algorithms namely PPI, N-Findr and AMEE.

PEXTA I/O is also suitable for Multi-dimensional On-Line Analytical Processing (MOLAP) used in data warehousing. In MOLAP, the basic data representation is a multi-dimensional array where each dimension represents discrete value attribute. For example, the number of items QTY sold by a company that sells many different items in different stores on any particular day, can be captured as a multi-dimensional array *QTY(ItemId, StoreId, Day). ItemId* and *StorId* denote unique identifiers for an item and a store respectively. If each cell is defined by a vector of $\langle ItemId,\ StoreId,\ Day\rangle$, then the value stored in that cell represents the quantity (QTY) sold. Extendibility

of the dimension representing *ItemId* is necessary if new items are sold by the company. Similarly, the dimension representing *StoreId* will be required to extend if the company increases its number of stores.

10.2 Future Directions

A number of activities have been identified for future work. To improve I/O performance of PEXTA, we intend to implement an in-memory resident cache that stores session data. This in-memory caching system will improve performance by holding frequently-requested chunks in memory, reducing the need for access requests to get that data from disk files.

Our chunked extendible array allocation scheme currently do not support shrinking. This means that array bounds can only be extended or increased. To support shrinking, the order of dimensional bounds extensions need to be stored as part of the entries within the axial-vectors. When the dimensional bounds are reduced, some data elements will be discarded. This will require some repacking of the remaining data elements. Some leaf nodes of the the O_2-Tree will either be deleted or their content will be updated. This will normally results in an update and rebalancing of the O_2-Tree. Techniques for such dimensional bound shrinking has not yet been investigated in the literature.

Other activities outlined for future work include (1) the determination of an optimal chunk size for HDF5 array files, (2) empirical and experimental investigations to determine which space filling curve produce the minimum sequence distance for near

neighbourhood search, (3) incorporating our storage allocation scheme into other files formats such FITS and (4) collaborating with the Global Arrays Programming Models group at Pacific Northwest National Laboratories (PNNL) to incorporate the PEXTA APIs as one of the I/O libraries of GA.

A

Appendix

A.1 Uniform Extendible Array

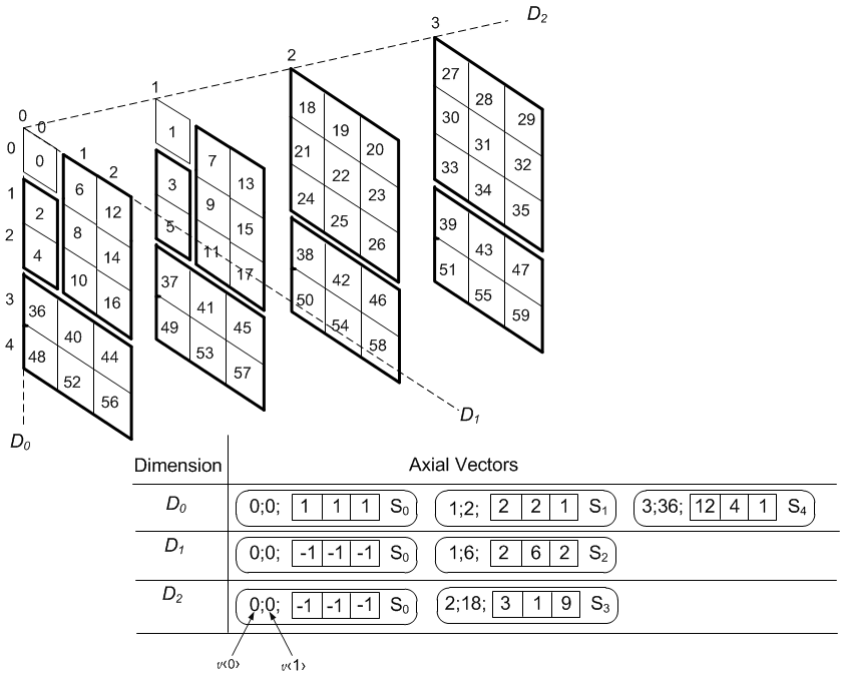


Figure A.1.1: Uniform Extendible Array with Linear Expansion : $\theta = 2$

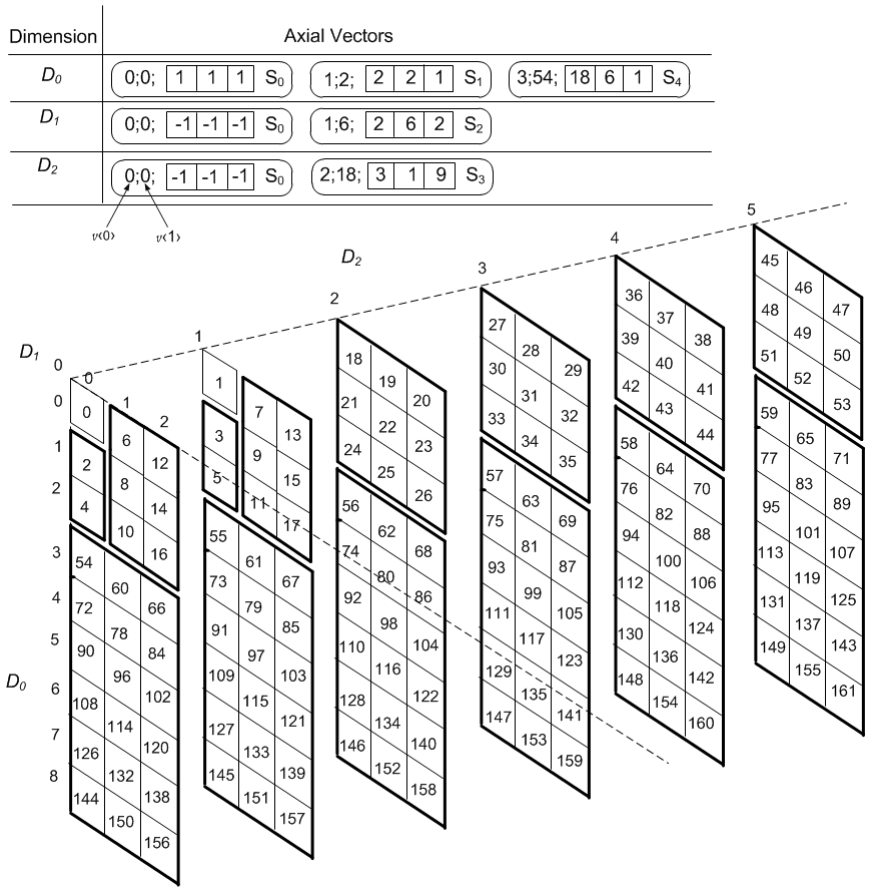


Figure A.1.2: Uniform Extendible Array with Exponential Expansion : $e = 2$

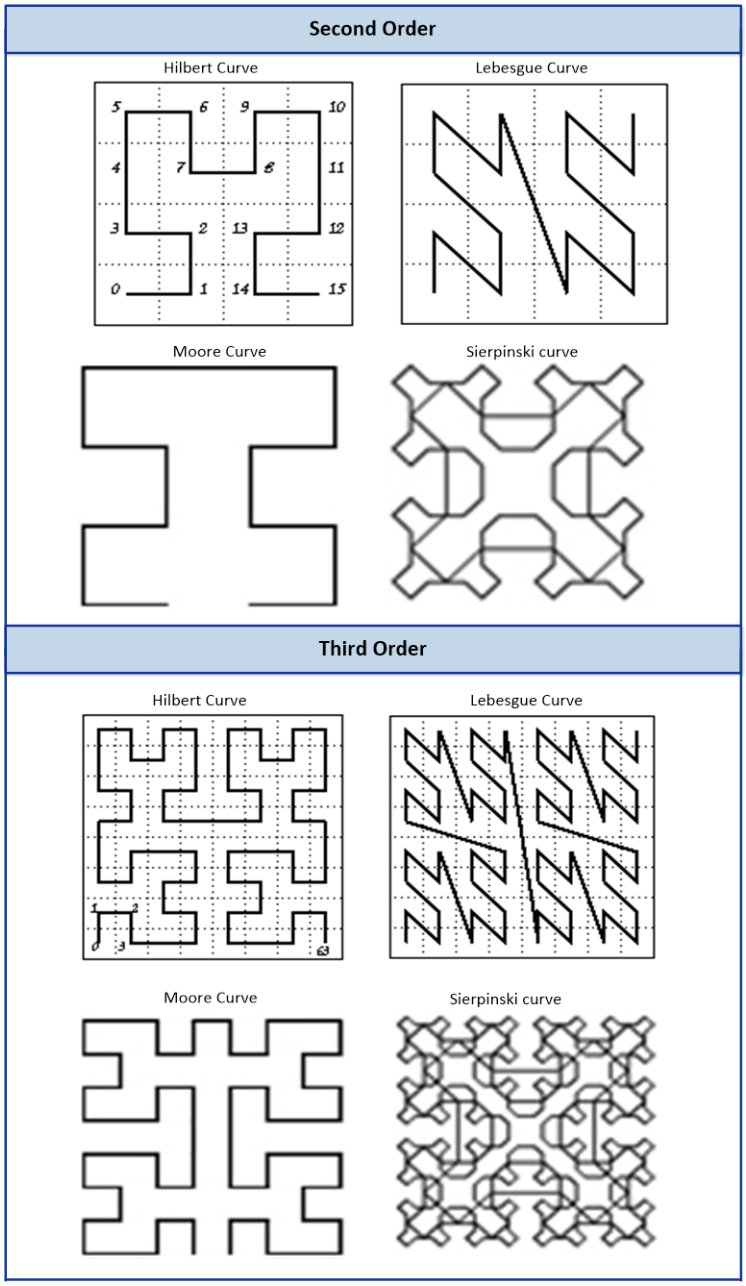


Figure A.2.1: Second and Third Order of Space Filling Curves

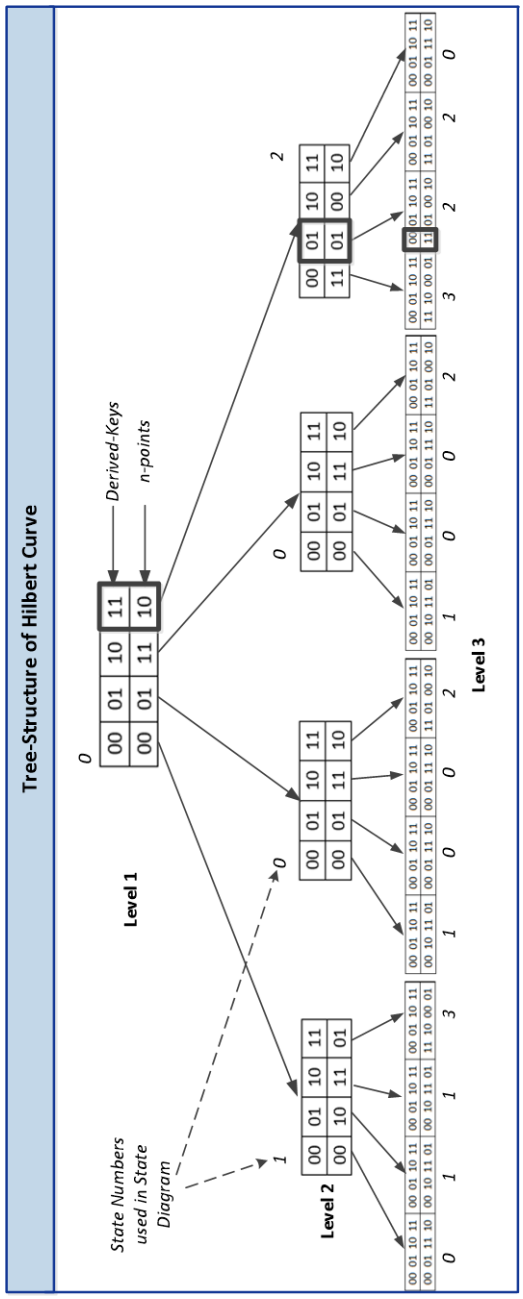


Figure A.2.2: Third Order Tree Repreeantation of Hilbert Curve in $2D$

References

[1] V. Choi, J. Soumagne, and Q. Koziol, "Extending HDF5 Datasets: Enhancements to the Chunk Indexing Methods."

[2] K.-T. Lim, S. D. Maier, O. Ratzesberger, and S. Zdonik, "Requirements for Science Data Bases and SciDB." Citeseer.

[3] E. Otoo and T. Merrett, "A storage scheme for extendible arrays," *Computing*, vol. 31, pp. 1–9, 1983, 10.1007/BF02247933. [Online]. Available: http://dx.doi.org/10.1007/BF02247933

[4] E. J. Otoo and D. Rotem, "A storage scheme for multi-dimensional databases using extendible array files," *Proceedings of the Third STDBM Workshop on Spatio-Temporal Database Management*, pp. 67–74, 2006.

[5] A. L. Rosenberg, "Allocating storage for extendible arrays," *J. ACM*, vol. 21, pp. 652–670, October 1974. [Online]. Available: http://doi.acm.org/10.1145/321850.321861

[6] A. L. Rosenberg and L. J. Stockmeyer, "Hashing schemes for extendible arrays," *J. ACM*, vol. 24, pp. 199–221, April 1977.

[7] S. M. M. Ahsan and K. M. A. Hasan, "An implementation scheme for multidimensional extendable array operations and its evaluation," *ICIEIS 2011, Part III, CCIS 253*, pp. 136 –150, 2011.

[8] K. A. Hasan, K. Islam, M. Islam, and T. Tsuji, "An extendible data structure for handling large multidimensional data sets," in *Computers and Information Technology, 2009. ICCIT'09. 12th International Conference on.* IEEE, 2009, pp. 669–674.

[9] S. Joannou and R. Raman, "An empirical evaluation of extendible arrays," in *International Symposium on Experimental Algorithms.* Springer, 2011, pp. 447–458.

[10] J. K. Iliffe, "The use of the genie system in numerical calculation," *Annual Review in Automatic Programming*, vol. 2, pp. 1–28, 1961.

[11] D. Ohene-Kwofie, E. J. Otoo, and G. Nimako, "O2-Tree: A Fast Memory Resident Index for Nosql Data-Store," in *CSE '12 Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering*, 2012, pp. 50–57.

[12] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the Message-Passing Interface," in *European Conference on Parallel Processing.* Springer, 1996, pp. 128–135.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[14] J. Nieplocha, R. J. Harrison, and Littlefield, "Global arrays: a portable shared-memory programming model for distributed memory computers," in *Proceedings*

of the 1994 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, 1994, pp. 340–349.

[15] J. Nieplocha and I. Foster, "Disk resident arrays: An array-oriented i/o library for out-of-core computations," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*. IEEE, 1996, pp. 196–204.

[16] G. Nimako, E. J. Otoo, and D. Ohene-Kwofie, "Chunked extendible dense arrays for scientific data storage." in *ICPP Workshops*, 2012, pp. 38–47.

[17] E. J. Otoo, G. Nimako, and D. Ohene-Kwofie, "Using chunked extendible array for physical storage of scientific datasets." in *SC Companion*, 2012, pp. 1315–1321.

[18] J. Boardman, "Automating Spectral Unmixing of AVIRIS Data using Convex Geometry Concepts," in *Sum. of the 4th Annual JPLAirborne Geosci. Workshop, JPL Pub.*, 1993, pp. 11–14.

[19] M. E. Winter, "N-FINDR: An Algorithm for Fast Autonomous Spectral End-member Determination in Hyperspectral Data," in *Proc. of the SPIE Conf. on Imag. Spectrom*, 1999, p. 266–275.

[20] S. Sanchez, G. Martin, and A. Plaza, "Parallel implementation of the N-FINDR endmember extraction algorithm on commodity graphics processing units," in *Geoscience and Remote Sensing Symposium (IGARSS), 2010 IEEE International*, 2010, pp. 955 – 958.

[21] A. Mercerand and A. Rosenfeld, "An Array grammar Programming System," *Commun. ACM*, vol. 5, pp. 299–305, May 1973.

[22] J. Earley, "Towards an understanding of data structures," *Commun. ACM*, pp. 617–627, 1971.

[23] K. A. Hasan, K. Islam, M. Islam, and T. Tsuji, "An extendible data structure for handling large multidimensional data sets," in *Computers and Information Technology, 2009. ICCIT'09. 12th International Conference on.* IEEE, 2009, pp. 669–674.

[24] D. Rotem and J. L. Zhao, "Extendible arrays for statistical databases and olap applications," *Proceedings of the Eighth International Conference on Scientific and Statistical Database Management*, pp. 108–117, 1996. [Online]. Available: http://dx.doi.org/10.1109/SSDM.1996.506053

[25] J. Plaza, D. Valencia, P. A., and C.-I. Chang, "Parallel Implementation of Endmember Extraction Algorithms From Hyperspectral Data," in *IEEE Geoscience and Remote Sensing Letters*, vol. 3, 2006, pp. 940–943.

[26] J. Plaza, R. Pérez, A. Plaza, P. Martínez, and D. Valencia, "Parallel morphological/neural processing of hyperspectral images using heterogeneous and homogeneous platforms," in *Cluster Computing*, vol. 11, 2007, pp. 17–32.

[27] J. Plaza, D. Valencia, and P. A., "An experimental comparison of parallel algorithms for hyperspectral analysis using heterogeneous and homogeneous networks of workstations," in *Parallel Computing*, vol. 34, 2005, p. 92–114.

[28] P. R. Marpu, S. Bernabe, P. A., and J. A. Benediktsson, "A new parallel tool for classification of remotely sensed imagery," in *Computers and Geosciences*, vol. 46, 2012, p. 208–218.

[29] S. Sánchez and A. Plaza, "Fast determination of the number of endmembers for real-time hyperspectral unmixing on GPUs," in *Real-Time Image Processing*, vol. 9, 2012, pp. 397–405.

[30] J. M. P. Nascimento, J. M. Bioucas-Dias, J. M. R. Alves, V. Silva, and A. Plaza, "Parallel Hyperspectral Unmixing on GPUs," in *Geoscience and Remote Sensing Letters, IEEE*, vol. 11, 2014, pp. 666–670.

[31] X. Wu, B. Huang, A. Plaza, Y. Li, and C. Wu, "Real-Time Implementation of the Pixel Purity Index Algorithm for Endmember Identification on GPUs," in *Geoscience and Remote Sensing Letters, IEEE*, vol. 11, 2014, pp. 666–670.

[32] C. Cormen, T. Stein, and R. Leiserson, C.and Rivest, *Introduction to Algorithms*, 3rd ed. Cambridge: Cambridge, Mass. MIT Press, 2009.

[33] E. J. Otoo, D. Rotem, and S. Seshadri, "Optimal chunking of large multidimensional arrays for data warehousing," in *DOLAP 2007, ACM 10th International Workshop on Data Warehousing and OLAP, Lisbon, Portugal, November 9, 2007, Proceedings*, 2007, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/1317331.1317337

[34] S. Sarawagi and M. Stonebraker, "Efficient organisation of large multidimensional arrays," in *Proceedings of the Tenth International Conference on Data*

Engineering,USA, 1994, pp. 328–336. [Online]. Available: http://dx.doi.org/10.1109/ICDE.1994.283048

[35] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Science & Business Media, 2012, vol. 9.

[36] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*.

[37] M. Bader, "Sierpinski curves," in *Space-Filling Curves*, ser. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2013, vol. 9, pp. 77–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31046-1_6

[38] C. H. Hamilton and A. Rau-Chaplin, "Compact hilbert indices for multi-dimensional data," in *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*. IEEE, 2007, pp. 139–146.

[39] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *ACM SIGMOD Record*, vol. 30, no. 1, pp. 19–24, 2001.

[40] A. R. Butz, "Alternative algorithm for Hilbert's space-filling curve," *IEEE Transactions on Computers*, no. 4, pp. 424–426, 1971.

[41] J. K. Lawder, "Calculation of mappings between one and n-dimensional values using the Hilbert's space-filling curve."

[42] D. DeFord and A. Kalyanaraman, "Empirical analysis of space-filling curves for scientific computing applications," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 170–179.

[43] P. Xu and S. Tirthapura, "A lower bound on proximity preservation by space filling curves," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 1295–1305.

[44] Unidata, *NetCDF: Network Common Data Form*. NetCDF Software Library and Utilities, 2012.

[45] T. H. Group, *HDF5: Hierachical Data Format, HDF5-1.8.8*. Illinois: HDF5 (Hierarchical Data Format 5) Software Library and Utilities, 2012.

[46] IAU, *Definition of the Flexible Image Transport System*. IAU FITS Working Groups, 2008.

[47] H. Group, "Parallel hdf5," 2012. [Online]. Available: ttp://www.hdfgroup.org/HDF5/PHDF5/

[48] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the Global Arrays PGAS model using MPI one-sided communication," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 739–750.

[49] J. Breitbart, "Dataflow-like synchronization in a pgas programming model," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 762–769.

[50] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[51] G. F. Lofstead, Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, R. A. Oldfield *et al.*, "Hello adios: The challenges and lessons of developing leadership class i/o frameworks." Sandia National Laboratories, Tech. Rep., 2012.

[52] M. Krishnan and J. Nieplocha, "SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International.* IEEE, 2004, p. 70.

[53] J. Le Moigne, W. J. Campbell, and R. F. Cromp, "An automated parallel image registration technique based on the correlation of wavelet features," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, no. 8, pp. 1849–1864, 2002.

[54] F. J. Seinstra, D. Koelma, and J.-M. Geusebroek, "A software architecture for user transparent parallel image processing," *Parallel computing*, vol. 28, no. 7, pp. 967–993, 2002.

[55] S. De Backer, P. Kempeneers, W. Debruyn, and P. Scheunders, "A band selection technique for spectral classification," *Geoscience and Remote Sensing Letters, IEEE*, vol. 2, no. 3, pp. 319–323, 2005.

[56] S. B. Serpico and L. Bruzzone, "A new search algorithm for feature selection in hyperspectral remote sensing images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 39, no. 7, pp. 1360–1367, 2001.

[57] T. El-Ghazawi, S. Kaewpijit, and J. L. Moigne, "Parallel and adaptive reduction of hyperspectral data to intrinsic dimensionality," in *Proceedings of the 3rd IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2001, p. 102.

[58] T. Achalakul and S. Taylor, "A distributed spectral-screening pct algorithm," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 373–384, 2003.

[59] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[60] I. Jolliffe, *Principal component analysis*. Wiley Online Library.

[61] P. Soille, *Morphological image analysis: principles and applications*. Springer Science & Business Media, 2013.

[62] A. J. Plaza, "Parallel processing of remotely sensed hyperspectral imagery: full-pixel versus mixed-pixel classification," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1539–1572, 2008.

[63] J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of aviris data," 1995. [Online]. Available: https://aviris.jpl.nasa.gov/proceedings/workshops/95_docs/7.PDF

[64] E. U. Guide, "Envi on-line software user's manual," *ITT Visual Information Solutions*, 2008.

[65] M. E. Winter, "N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data," in *SPIE's International Symposium on*

Optical Science, Engineering, and Instrumentation. International Society for Optics and Photonics, 1999, pp. 266–275.

[66] A. Plaza, P. Martinez, R. Pérez, and J. Plaza, "Spatial/spectral endmember extraction by multidimensional morphological operations," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, no. 9, pp. 2025–2041, 2002.

[67] R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 4, pp. 532–550, 1987.

[68] A. Plaza, J. Plaza, and H. Vegas, "Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems," *Journal of Signal Processing Systems*, vol. 61, no. 3, pp. 293–315, 2010.

[69] B. Meyer, "A constraint-based framework for diagrammatic reasoning," *Applied Artificial Intelligence*, vol. 14, pp. 327–344, 2000.

[70] T. H. Group, *HDF5 User's Guide.* Illinois: HDF5 (Hierarchical Data Format 5) Software Library and Utilities, 2006.

[71] T. Tsuji, M. Kuroda, and K. Higuchi, "History offset implementation scheme for large scale multidimensional data sets," in *Proceedings of the 2008 ACM symposium on Applied computing.* ACM, 2008, pp. 1021–1028.

[72] J. Earley, "Toward an understanding of data structures," *"Toward an understanding of data structures", Commun. ACM*, vol. 14, pp. 617–627, October 1971.

[73] R. Graham, R. Perriss, and A. Scarsbrook, *DICOM demystified: A review of digital file formats and their use in radiological practice, Clinical Radiology Review.* Cambridge, Mass. MIT Press, 2005.

[74] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," pp. 15–21, NOV 2003.

[75] A. Mercer and A. Rosenfeld, "An Array Grammar Programming System," *Commun. ACM*, vol. 5, pp. 299–305, May 1973.

[76] A. Milgram, D.and Rosenfeld, *"Array automata and array grammars"*, pp. 166–173, 1971.

[77] J. Nieplocha and I. Foster, "Disk resident arrays: An array-oriented i/o library for out-of-core computations," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the.* IEEE, 1996, pp. 196–204.

[78] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[79] E. J. Otoo and D. Rotem, "Parallel access of out-of-core dense extendible arrays," in *2007 IEEE International Conference on Cluster Computing.* IEEE, 2007, pp. 31–40.

[80] E. Otoo, *Parallel and Distributed Access of Dense Multidimensional*, pp. 1– 9.

[81] E. J. Otoo and D. Rotem, "Parallel access of out-of-core dense extendible arrays," in *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 31–40.

[82] A. L. Rosenberg, "Managing storage for extendible arrays (extended abstract)," *Sixth annual ACM symposium on Theory of computing*, pp. 297–302, 1974. [Online]. Available: http://doi.acm.org/10.1145/800119.803907

[83] T. A. Standish, *Data Structure Techniques*. Reading, Mass.: Addison-Wesley, 1980.

[84] K. M. Azharul Hasan and T. Tsuji, *Extendible Arrays for Multidimensional Databases: Concepts, Implementation and Evaluation*. Germany: LAP Lambert Academic Publising, 2011.

[85] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, Massachusetts: Addison Wesley Publishing Co., 1993.

[86] G. Peano, "On a curve which completely fills a planar region," *Mathematische Annalen*, vol. 36, no. 1, pp. 157–160, 1890.

[87] D. Hilbert, "Ueber die stetige abbildung einer line auf ein flächenstück," *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891.

[88] R. A. van de Geijn and J. Watts, "Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.