INSTITUT FÜR INFORMATIK
OF LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Index Set Mappings for Multidimensional Views on PGAS Data Domains

Marius Herget

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Index Set Mappings for Multidimensional Views on PGAS Data Domains

Marius Herget

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Fuchs
Abgabetermin:	Mittwoch, 19.08.2020

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 19.8.2020

Marius Herget

"Putting people of all Shapes, Sizes, Colors. Putting them on stage together and presenting them as equals, another critic might have even called it a celebration of humanity."

– James Gordon Bennett, *The Greatest Showman*

Abstract

The Partitioned Global Address Space (PGAS) programming model represents distributed memory as global shared memory space. The DASH library implements C++ standard library concepts based on PGAS and provides additional containers and algorithms that are common in High Performance Computing (HPC) applications.
This thesis examines especially sparse matrices and whether state-of-the-art conceptual approaches to these data storage can be abstracted to improve adaptation of domain-specific computational intent to the underlying hardware. Therefore, an intensive systematic analysis of state-of-the-art distribution and storage of sparse data is done. Based on these findings a given property classification system of general dense data distributions is extended by sparse properties.
Afterward, a universal vocabulary of a domain decomposition concept abstraction is developed. This four-layer concept introduces the following steps: (i) *Global Canonical Domain*, (ii) *Formatting*, (iii) *Decomposition*, and (iv) *Hardware*. In combination with the proposed abstraction, an index set algebra utilizing the resulting sparse data distribution system is presented. This concludes in a newly created Position Concept that is capable of referencing an element in the data domain as a multivariate position. The key feature is the ability to reference this element with arbitrary representations of the element's physical memory location.

Contents

1 Introduction

Although matrices and in more detail sparse matrices are elementary for science and industry, only a few projects implement modern and efficient solutions for calculating large sparse matrices on heterogeneous computing clusters. Besides, some implementations have been done for utilizing matrix calculations in HPC environments, in most cases, these are highly specialized and individualized for a small number of problems. In the modern world, where the trend goes to heterogeneous, complex data centers, and decentralized solutions (folding @ home), these applications do not have great portability. In the near future, a new formal description has to be established, which combines high-level approach portability, user-driven structure, and high compatibility.

This master thesis concentrates on the fact of how to distribute sparse data formats while enabling a user to define domain-specific computational intent. With the challenges of expressiveness, portability, and compatibility in mind, a formal foundation is developed by introducing a formal vocabulary for data decomposition of irregular, sparse, and hierarchical data domains. This is based on the devised index set mappings for multi-dimensional data domains and a new concept for referencing elements within the data domain in local and global domain space.

1.1 Data Distribution and Parallel Programming in PGAS

In a modern highly heterogenous environment Partitioned Global Address Space (PGAS) is a "programming model suited for shared and distributed memory parallel machines" [Alm11]. These machines consist of multiple processing units connected through a communication network. These units have attached local memory, which is available to all processors within the system. This distributed memory space can introduce private sections only available for the local processing unit. Although these system are highly connected different latency classes are available: (i) *affinity* (memory which is local with low latency), (ii) *tray-local* (memory positioned on the same processing system connected by high-bandwidth and reasonable low latency data lanes), (iii) *rack-local* (memory positioned on boards within the same rack), and (iv) *inter-rack* (memory only accessible through the communication network) [Alm11; KFS18]. These latencies are highly important for a fast calculation, and therefore the optimization of communication efficiency is crucial. Data distribution plays a huge role in

PGAS. Nevertheless, PGAS only determines between local and global memory. Hence it is not able to differentiate between tray-local, rack-local, and inter-rack locality out of the box [Für+14].

PGAS implementations often rely on one-sided message communication, regularly implementing as Remote Direct Memory Accesses (RDMA), whereby the remote CPU is not bothered with the transfer of data. The special feature of PGAS is establishing local coherency domains while having a non-coherent global domain. [Yel+07; FL10].

One approach to simple and effective data distribution is Vienna Fortran. FORTRAN is a general-purpose, compiled imperative programming language that was "designed to solve problems that can be expressed algebraically and used mainly in mathematics, science, and engineering" [For]. Although Fortran implemented the PGAS concept in 2008 with *Coarray Fortran (CAF)*, an early adopter of the theory was Vienna Fortran by Chapman, Fahringer, and Zima [CFZ94]. The language extension is based on the Single Program, Multiple Data (SPMD) concept, which enables parallelization by simultaneously computing instructions on different parts of the data domain. Their data distribution schema is, therefore, similar to PGAS: (i) the data domain is partitioned, (ii) each data array is mapped to a processor, (iii) the processor owns this data, while it is stored in its local storage, (iv) the work is distributed to the processors, (v) each processor performs the computation which is assigned to its owned data. This *owner computes pardigm* is supported by a message-passing functionality to enable non-local memory access. During the whole schema, the programmer is in charge of the selection of the data distribution. This extends to defining communication methods, logical processor structure, and dynamically distribution. [CFZ94].
CAF is current implication of the PGAS concept within Fortran. It was initially proposed in 1998 by Numrich and Reid [NR98]. *Work* is distributed according to the SPMD model and is executed asynchronously by standard Fortran. Meanwhile, data distribution is up to the programmer. After its finale implementation in Fortran Standardization Mellor-Crummey et al. [MC+09] proposed further features, e.g., global pointers, safe synchronization, and collective communication.

Another implementation of PGAS can be found in Chapel [CCZ07] which goal is to provide a high-level abstraction while still providing specific low-level tier to improve control and or performance [CC11]. One special feature is that it allows "standard and user-defined data distributions that permit advanced users to specify their own global-view array implementations" [Lar+11]. These global-view arrays allow the program to apply natural operations on data that is potentially distributed on multiple units or nodes. The method of individualized data distribution is capable of specific layouts over nodes or specific memory. In Chapel this is called *Domain Map* [CC11].

Unified Parallel C (UPC) is another PGAS programming language for large scale parallel machines based in *ANSI C*. It follows C's philosophy and requires the programmer to program on a low level close to the hardware. UPC is implemented by different players (e.g., IBM, HP, and Verkley) and supports static and dynamic memory allocations. It is also based on the SPMD design idea. [BCA07; Hud]

This thesis concentrates on DASH which is a `C++` template library specialized on software for Exas-cale computing [Für+14]. This framework will be further explaned in section 2.3.

All in all, there are various ideas on how to distribute data. Nevertheless, the optimal distribution pattern is always dependent on the use case. Which suits best depends on different questions:

- How and how often does the algorithm/use case access remote memory?
- On which hardware does the algorithm runs? This highly depends on processes and the communication method (network topology).
- How powerful is the computation capacity, and how quickly can it access local data?
- How parallelizable is the algorithm? Is the developer able to split up tasks to run in close-by locality?

1.2 Problem Statement

HPC is facing the exascale barrier while moving towards a highly heterogeneous node level system configurations. Therefore, highly adaptive load balancing strategies and access times reduction methods are required to keep the overhead under control. This allows to exploit the diverse characteristics of heterogeneous computing and storage components. There are two possibilities for optimization: at compile-time and run-time. These specialized computing strategies are often applied to a scientific application. The correct approach is highly use-case specific. Hence, an identical domain-specific problem is algorithmically formulated in system-specific execution models like sequential execution or data-parallel access.

One particularly challenging aspect of this vast research domain is irregular, sparse, and hierarchical data domains. Sparse matrices are an essential, fundamental data structure to higher-level algorithms and applications like graphs, high dimensional feature extraction, and many others. One example is feature extraction within a language (Bag-of-Words model). The vocabulary of a language is saved as a very large corpus, and sentences are represented as a highly sparse vector representation of this corpus.

In contrast to regular, dense data domains that are already intensively researched, irregular, sparse domains introduce completely new challenges. Due to their nature, there are a lot of unoccupied elements which are often neglectable. Conclusive load balancing is a huge, complex task. Therefore, this thesis addresses the following research question:

Can conceptual state-of-the-art approaches to sparse data storage be abstracted to improve adaptation of domain-specific computational intent to underlying hardware?

In extension, this results in the question of practicability:

Given the abstraction as postulated in the research question, how could a software architecture put this expressiveness into practice?

1.3 Objectives and Contributions

In brief, this work addresses the approach of a general concept to efficiently save, and distribute irregular data domains in highly heterogenous HPC systems. The objective is to build a formal description of index set mappings and extend a given abstraction with a property classification system of domain decomposition. This includes the following contributions:

- A systematic analysis of state-of-the-art distribution and storage of sparse data
- Extending a given property classification system of general dense data distributions by sparse properties
- An universal vocabulary (specification language) of a concept abstraction and index set algebra utilizing the resulting sparse data distribution system

The research question and the contributions will be mainly investigated based on sparse matrices and their attributes. It is important to emphasize that this refers to software implementation in general, independent from programming languages and library support. The research objective is not to develop a specific solution but to establish an expressive model. This concludes that the following tasks are out of scope for this thesis:

- Process the developed decomposition efficiently to the hardware (layouting)
- Identify and define the hardware structure and develop a competitive mapping of the data model to this structure
- A well-defined reference implementation in DASH

1.4 Structure of this thesis

The remainder of this thesis is structured in the following manner:

Chapter 2 Background and Related Work

Establishes the base knowledge of the research area. While defining, describing, and explaining the background of data domains and decomposition, DASH is briefly introduced. Afterward, the current state-of-the-art techniques and properties for domain decomposition and data operations on dense data domains are presented. In the final section, the currently available approaches are described.

Chapter 3 Approach

Introduces the fundamental theory about irregular domains and sparse matrices. Moreover, it discusses different sparse matrix storage formats and basic operations on sparse matrices. This leads to the conclusion of how the current standards need to be expanded to support domain decompositions on irregular domains. From this, a new abstraction and a general index set mapping concept is concluded. It also introduces a new, more capable reference and representation concept.

Chapter 4 Use Cases & Evaluation

Specifies the proposed abstraction and concepts by presenting various operations, mappings, optimizations, and examples.

Chapter 5 Summary & Conclusion

Revisiting the objective of a general concept for irregular (sparse) data domain decomposition in a summary of contributions and derived conclusions. Lastly, a recommendation for future work is presented.

2

Background and Related Work

This chapter reviews related work to identify challenges and opportunities of different data domains and the resulting consequences for a possible abstraction postulated in the research question. Hence, state-of-art approaches and concepts are explored.

2.1 Data Domains

As already mentioned in the first chapter, data distribution is a crucial ingredient for high efficient parallel programming. It determines how fast and easy data can be accessed and further processed. This section discusses how data can be logically organized to provide the foundation of a meaningful distribution.

A *data domain* describes a collection of values that are related somehow. Sometimes this is expressed in the form of "restrictions on larger sets of values" [Los01]. Within this thesis context, it is explained as an intrinsic form of data structuring and representation. Therefore, the values are not directly restricted, but the relationship within the data domain. These data domains are able to provide a basis for data decomposition and data spaces which are further explained in section 2.2.

There are various forms of data domains. The most common ones are now explained:

regular Regular data domains have a repetitive, arithmetic pattern.

irregular Irregular data domains have no persistent referencing concept and layout design, and therefore it is impossible to derive a general resolution concept to them (from a general arithmetic expression). Hence, each irregular data domain has to be individually interpreted with the help of a secondary mapping.

k-dimensional Data domains can be one- ($k = 1$) or multidimensional ($k > 1$).

hierarchical Hierarchical data domains are "data [that] are organized into records that are recursively composed of other records" [Hai09]. Thereby, some are independent, and some concepts depend on higher-level domains. This can be visualized, e.g., in tree structures.

dynamic Dynamic data domains are in the most cased irregular data domain where data changes during runtime. This requires a dynamic index lookup and might require on-the-fly address space reallocation.

sparse Within sparse data domains, the majority of elements are unpopulated - returning an unallocated default value, usually 0 or *null*. Only populated, non-zero elements are allocated in physical memory.

2.2 Domain Decomposition

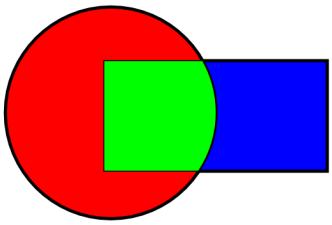


Figure 2.1: Schwarz's original problem [Con20]

As previously mentioned, the correct distribution of data can make or break an efficient parallel program. Hermann Schwarz (1843-1921) ist often associated as the inventor of domain decomposition methods. In his 1869 paper "Ueber einige Abbildungsaufgaben.", he introduces an alternating method to solve a Partial Differential Equation (PDE) set on a complex domain composed of the overlapping union of the structure. The original problem is shown in fig. 2.1. The developed mathematical basis is nowadays one of the most important aspect for scientific computing. [GR06]

In modern HPC, there are two essential problems to tackle: the number of operations and the required memory. Gosselet and Rey [GR06] introduces three criteria for the decomposition of an simulation PDE: (i) create independent problems which can run on independent processors, (ii) amount of synchronization points, and (iii) quantity of data which has to be traded during these points. The authors result in these since, in their observations, inter-processor communication is one of the most time-consuming aspects in HPC. To fulfill these criteria optimally, the authors consider the three great classes of mathematical decomposition as the basis of solving it: operator splitting, function-space decomposition, and domain decomposition. They conclude that only the last class ensures "that independent computations are limited to small quantities and that the data to exchange is limited to the interface (or small overlap) between subdomains, which is always one-to-one exchange of small amount of data" [GR06].

According to Smith, Bjorstad, and Gropp [SBG04], domain decomposition for a PDE can be generally defined in three ways depending on the context:

Parallel Computing: The process of distributing data from a computational model among multiple inter-connected computing units with distributed memory. Hence data has to be spread, but it does not need to influence the numerical solution methods. This can also be called *Data Decomposition*.

Asymptotic Analysis: The separation of the physical domain into areas that can be solved with different equations. In this context, communication is used to handle the various conditions of the data domain.

Preconditioning: The concept of dividing the solution of a large linear system into smaller subproblems.

These definitions are independent of each other and can arise in one single program in various aspects. Therefore, all three have to be considered when developing a custom domain decomposition model. The main goal is to develop a concept to solve a problem on large-scale computer networks in the most efficient way. Hence, there are various ways to enable this. Nevertheless, all of them share some common classes and properties which describe and distinguish different approaches. Those will be partly defined and explained in section 2.5.

2.3 DASH

DASH is a compiler free PGAS *C++* library approach with a SPMD model in mind. While other HPC programming models (e.g., Chapel) rely on a predetermined parallel programming structure, DASH only relies on predefined units. A unit is an individual participant of computing infrastructure which can vary from a CPU, a thread to an image. Those units are organized in teams that are dynamic ordered sets of units that regulate Remote Memory Access (RMA), syncing, and communication within the program. The base structure is a root team with all units defined. During runtime, different subsets can be build to form a hierarchy topology. Therefore, a programmer can customize his structure, e.g., in the form of the designated hardware topology or specific algorithm design. DASH runs in his own environments DAsh RunTime (DART). Communication is implemented as a one-sided Message Passing Interface (MPI). One advantage of this template library is that it is built similar to the common *C++* Standard Template Library (STL). Hence, the learning curve is quite easy. [FFK16]

DASH was developed with exascale computing within science in mind. Therefore, it enables different key features:

Hierarchical Locality One interesting aspect of most PGAS models is the differentiation between local and global data. DASH takes this approach one step further and implements a hierarchical approach. The base is the already mentioned hierarchical team construct. Carrying this idea on, the memory concept directly connects to the specific units, and therefore, different layers of memory locality can be manufactured. Hence, pointers and iterators can be specified at different levels of locality (e.g., within the unit's memory, within the teams distributed memory and global memory). [Für+14]

Data-Driven Science One important aspect of DASH development is the idea to construct code for data and the desired algorithms. Therefore, the hierarchical locality is used to adapt, especially heterogeneous infrastructure, to an efficient solution. The developer can derive based on his data different memory patterns. This basic approach concentrates on the best answer to the problem and not the easiest developing path.

Multidimensional Data DASH supports n-dimensional data containers with efficient access methods. Furthermore, it provides various methods of working with the multidimensional data out of the box (e.g., slicing, block regions, and sub-matrix views). One key aspect is the n-dimensional data distribution capability which ranges from *block* and *cyclic* to *tiled* and *column-/row major* distributions. [FFK16; FF16a; DT20]

DASH-optimized Algorithms DASH is intentionally designed to be as close as possible to the *C++* STL algorithms. Therefore, common equivalents are already available (e.g. *fill*, *for_each*, *max_element*). These are complements of the STL versions. Within different latency classes are available: (i) project global range to local range, (ii) apply STL algorithm on the local range, and (iii) if necessary combine the results. Nevertheless, DASH data structures are also capable of being used in the default STL algorithms. [Für+14]

Strong Scaling With exascale computing in mind, DASH sets great store on strong scalability. During initial benchmarking in 2014 by Fuerlinger, Fuchs, and Kowalewski [FFK16] the general algorithm *dash::min_element* scaled good up to 9800 cores of SuperMUC. In newer evaluations (2018) measured with the *Cowichan Problem*, DASH consistently achieves near to the best scaling results. Though in comparison to Chapel and Cilk, it does not struggle with memory issues. Due to appropriate data structures for the tested problem domain, DASH and its moderate to strong scaling is compatible with similar solutions in Go, TBB, Cilk, and Chapel. [Für+18]

Global Iterator/Pointer Concept DART introduces a virtual global memory space and global pointers/iterators to access it. Every unit is capable of computing a global pointer to

Clases	Numerical Matrix	Sparse Matrix	Gender	Adjacency Matrix	Social Security number
Ordered	Yes	Yes	No	No	Yes
Boundaries	Q	Q	Enumerated list ([Male, Female, Divers, Null])	Z^+	000-00-0000 - 999-99-9999
Dimension	2-dimensional	2-dimensional	1-dimensional	2-dimensional	1-dimensional
Position Concept	Coordinate tuple	e.g. *Sell-C-σ*	Index	Graph index tuple	None
Density	Dense	Sparse	Dense	Depends on graph	
Ontology	None	None	None	Spectal graph theory	None

Table 2.1: Example of Data Domain Classification

any location by simple arithmetic. [Für+14]

2.4 Classification of Data Domains

In section 2.1, different data domains have already been presented. Nevertheless, even those share some common attributes. In the following enumeration, some useful classes are shown which enable us to classify different data domains for further insights.

Ordered Are the elements within the data domain ordered or unordered.

Domain Boundaries Which value types are valid within the domain (e.g., absolute, nominal, continuous, ratio, or date and time).

Dimensions Is the data domain one- or multidimensional.

Resolution Concept Is there a global arithmetic pattern/expression to reference elements in the data domain.

Density Is the data domain dense or sparse.

Ontology Do the data domain has a concept to identify and show properties and relationships for its elements.

Hence applying these classes, data domains can be differentiated. Some example classifications are shown in table 2.1.

2.4.1 Index Sets

Index set theory is the concept of implementing a function x to deduce a set J to a set A [Hal17]. This can be mathematically express es $\{A_j\}_{j\in J}$, whereby A is the set in which its elements are indexed by the j values. Those form the index set J. [Mun00]

Index sets provide a resolution concept for regular and irregular data domain. Those indicate the order of elements within a data domain. There are different attributes which describe index sets:

Dimension An index set can be one- or multidimensional. Simple memory concepts can show this: There are different memory cells indexed by a simple address (e.g., $0x8020$). To address a specific byte/bit within the memory cell, an offset is provided. This concludes with a two-dimension index set.

Level This can be seen as hierarchical index sets: One index points to another set. This can be done on different levels of indices to perform a chain of sets. This is called index sets mapping and is further described in section 3.7.

Distinct An index set (or a part of it) can but does not have to be distinctive. Such index sets can therefore map in a *one-to-one* or *one-to-many* manner.

In programming index sets can be used to create *pointers*, *ranges* or *iterators*.

2.5 Classification of Domain Decomposition for Dense Domains

Domain decomposition is used to solve problems on large scale computer networks. This can be done with various concepts and approaches. Therefore, it is useful to define attributes to build different classes of domain decompositions. In the following section, we discuss the different steps to build efficient decomposition methods and discuss and explain attributes for each layer.

During the following classification, this thesis uses different terms to describe the different areas of data domains:

Partition The data domain is splitted into different partitions (e.g. *sorting scope* σ). These partitions may consist of one or various blocks.

Block Partitions are split into multiple blocks (e.g. *chunks* c) which consist of at least one group of elements.

Group of elements A group consist of a individual number of elements in a certain layout (e.g. *row* or *column*).

Element A element is a single entity of data.

The following properties are based on the work of Fuchs and Fürlinger [FF16b].

2.5.1 Partitioning Properties

When creating partitions or blocks to prepare smaller data packages for procesing units, different properties may apply:

rectangular Partition's or block's shape is constant in every single dimension, e.g. every row has identical size.

***k*-dimensional** The partitions/blocks have k amount of dimensions (one- or multidimensional).

uniform All partitions/blocks have the same shape.

balanced All partitions/blocks have the same number of (compressed, non-padded) elements.

sequential All elements in a block are organized sequential.

2.5.2 Mapping Properties

After partitioning, the blocks are mapped to the processing units. This mapping can fulfill various properties:

order preserving The order of elements, groups of elements, or blocks is not modified.

global/blocked sorting The blocks/groups of elements (e.g. *rows*) are globally or locally sorted ascending/descending (e.g. *size*).

balanced The number of assigned elements is identical for every unit.

Bounded balanced The number of assigned elements is within a lower or upper boundary for every unit.

uniform The number of groups of elements (e.g. *rows*) is identical for every unit.

somewhat uniform The number of groups of elements (e.g. *rows*) is similar for every unit.

multiple More than one block may be mapped to a unit.

adjacent All groups of elements in a block/partition are mapped to the same unit.

2.5.3 Layout Properties

When distributing the mapped, partitioned data structures to the processing units, they are written in a certain way to the unit's memory.

row-major Row major storage order.

col-major Column major storage order.

linear Local element order corresponds to a logical linearization within single blocks (tiled).

2.6 Dense Data Domain Opterations

So far, a lot of attributes and definitions of dense data domains have been explained. Nevertheless, in the real world, different operations are usually applied to these domains. In the following section, the most common operations are described with the help of the matrix ($2d\text{-}array$) concept.

2.6.1 Matrix-Vector Multiplication

Basic Linear Algebra Subprograms (BLAS) is the de facto standard specification that defines low-level routines for common linear algebra operations. Originated in Fortran the current specification has been published in the early 2000s.

Equation (2.1) is the defined general Matrix-Vector product equation by BLAS. $\vec{y}$ is the resulting vector, $\vec{x}$ the input vector and A the Matrix. α and β are scalars. Since usually the following values apply: $\alpha = 1; \beta = 0$ the simplified equation can be found in eq. (2.2). [For01]

$$\vec{y} = \quad \alpha \times A \times \vec{x} + \beta \times \vec{y} \tag{2.1}$$

$$\vec{y} = \quad A \times \vec{x} \tag{2.2}$$

$$y_i = \sum_{i=j}^{m} \alpha \times a_{ij} \times x_j + \beta \times y_i \tag{2.3}$$

$$y_i = \sum_{i=j}^{m} a_{ij} \times x_j \tag{2.4}$$

Equation (2.3) shows the calculation of each element y_i of the result vector. m is the number of columns of the matrix A. This can also be simplified (see eq. (2.4)).

2.6.2 Matrix-Matrix Multiplication

BLAS defines the dense Matrix-Matrix quite similar to Matrix-Vector product (see. eq. (2.5)) and, as seen in eq. (2.6), can be similiar simplified. [For01]

$$C = \alpha AB + \beta C \tag{2.5}$$

$$C = \quad AB \tag{2.6}$$

According to the standard matrix multiplication each row of A is elementwise multiplicated with each row of B. The sum of this multiplications form the new element in the resulting matrix C:

$$c_{ik} = \sum_{j=1}^{m} a_{ij} \cdot b_{jk} \tag{2.7}$$

$$m\text{: } len(A_{row}) \text{ or } len(B_{column})$$

2.6.3 Border Slicing

A straightforward operation is to reshape the matrix by slicing columns or rows. In this example, the simplest form of slicing is investigated: cutting away a border. Nevertheless, the same techniques can be done for slicing other elements as well. This may occur with a performance penalty.

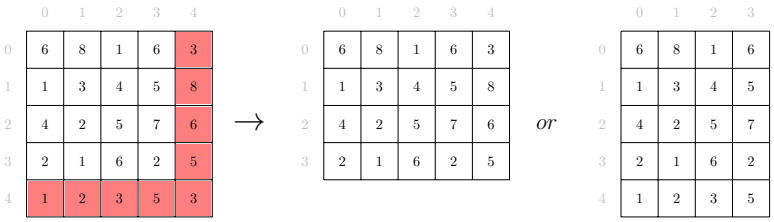


Figure 2.2: Border slicing with a dense matrix

The biggest pre-requirement is the storage concept. If the matrix is interpreted as *row-major*, it is easy to remove one row. This is equally true for *col-major* and removing a column. Hence, the more difficult task is to remove a column in a *row-major* layout concept. Then the storage needs to be manipulated by searching the correct column position in each row and removing the element for each row.

2.7 Related Work

In section 1.1 a brief overview over different approaches in HPC and domain decomposition have been presented. This section focusses on more detailed work done by different researchers in the field.

Chapel is, as already mentioned, a programming language focussed on high-level abstractions for data and task parallelism. The inventors *Cray Inc* claim productive, scalable, fast and portable characteristics [202b]. Hence, Chapel is organized in different distinct layers whereby high-level concepts are implemented by features on lower levels. These distinct layers can be viewed in fig. 2.3.

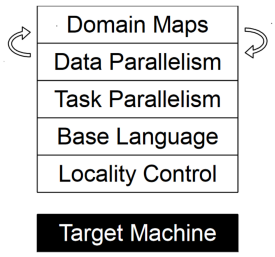


Figure 2.3: A notional diagram of Chapel's multiresolution design. [CC11]

One key feature is *Domain Maps*, which are defined as "recipes that instruct the compiler how to map the global view of a computation to the target locales' memory and processors" [202a]. Its primary purpose is to define data storage within local memory. Hence, an index mapping from global indices/elements to the locals is arranged. This includes implementing the correct layout of the data. Domain Maps are capable of running different operations (e.g. random access, iteration, and slicing). Chapel supports all common domain types for Domain Maps. Following its philosophy of high-level abstractions with the opportunity to implement low-level code on its own, there are some standard domain maps available. If these are not suitable for a specific problem, a user-defined domain map framework is provided. Therefore, multidimensional views with custom index mapping are possible to implement.[202a]

High Performance Fortran and **CAF** also support distribution of multidimensional views in different schemas. With Fortran 2008, an object-oriented design was introduced: the coarray parallel programming model. "An extension of the normal array syntax [in Fortran] to represent data decomposition plus an extension to the execution model to control parallel work distribution" [Num11] were added by implementing a SPMD concept. With the HPF-2 standard and Coarray, it is possible to develop distribution concepts for arbitrary mapping of data to processing units.

In addition, Mössbauer et al. [Mös+18] already presented a portable implementation of Coarrays in DASH.

Another programming language designed for effective HPC is **ZPL**. The key concept is to provide a programming language suitable for scientific and engineering computations and has the following main features: (i) array language, (ii) machine-independent, and (iii) implicitly parallel [Sny99]. ZPL implements regions that enable the compiler and user to define distributed array semantics. Nevertheless, Chamberlain et al. [Cha+00] concluded after an in-depth evaluation that ZPL is not suitable for dynamic and irregular problem domains. Therefore, it is not suitable for this thesis.

UPC has been already briefly introduced in section 1.1. The standard implementation of multidimensional data is just as linear arrays. Hence, data blocks are limited to one dimension, and new distributing methods need to be implemented. Some methods have already been proposed by El-Ghazawi et al. [EG+05] and Barton et al. [Bar+]. Since there are different implementations of UPC, some features are only available with one compiler. For example, IBM has its own IBM CL C compiler, which is optimized for the *POWER8*-architecture.

The UPC approach has been realized in *C++* with the UPC++ project by Zheng et al. [Zhe+14].

There are various approaches to multidimensional data decomposition in HPC. In some instances, the conditions need to be set precompile time.

The approach of DASH has, therefore, some advantages. It is an embedded extension for the current *C++* standard. Data decomposition and implementation can be run-time dependent. DASH is built on a well-defined mathematical concept, and its syntax is based on the STL algorithms. Nevertheless, it provides a high degree of customizability and is able to handle multidimensional and hierarchical data structures fine.

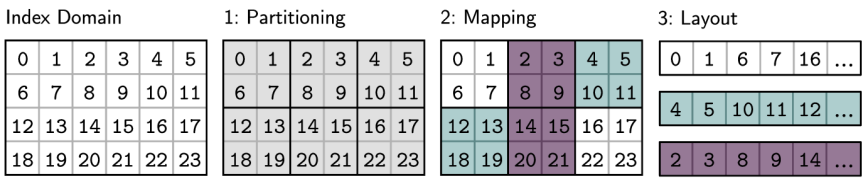


Figure 2.4: Example of partitioning, mapping, and layout in the distribution of a dense, two-dimensional array [FF16b]

A huge conceptual improvement has been proposded by Fuchs and Fürlinger [FF16b]: A general approach of decomposite dense data domains in PGAS environments. Their pattarn proposed the following layers for domain decomposition/distribution (see fig. 2.4): (i) Partitioning, (ii) Mapping, and (iii) (Memory) Layout. Within their work, there are two mapping layers: 1. from the index domain to the process, and 2. from process to local memory. The authors' work includes a wide variety of properties for each layer of decomposition. Nevertheless, their concept is not optimized for sparse data domains, since it takes no possible optimizations of storage, access patterns, or distribution in account.

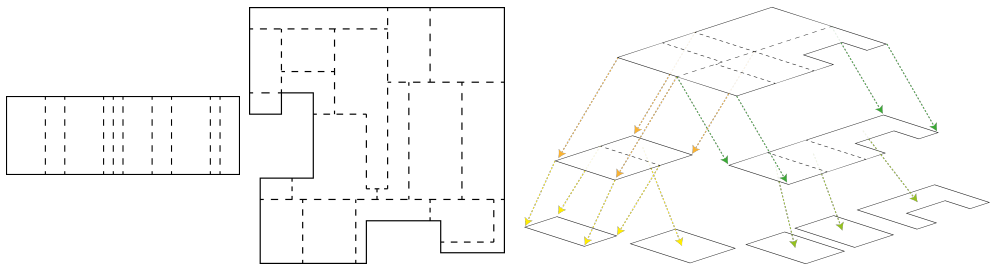
Programming Abstractions for Data Locality (PADAL) announced in recent years a shift from compute-centric to data-centric concepts for algorithms in HPC. Hence, data locality is the center of modern parallel programming. Within this context, this thesis follows the same idea. Data decomposition is the key to a good data locality and, therefore, an effective

access pattern. A detailed look into this new trend can be found in the work of Unat et al. [Una+17]. Within their publication the authors also acknowledge the work of Fuchs and Fürlinger [FF16b] and proposed a similar concept for data locality.

3 Approach

After establishing a general knowledge about the current concepts for other data domains, this section is dedicated to exploring irregular data domains. In this context, different storage formats for sparse data domains are presented. Based on these findings, a general abstraction is derived and presented.

3.1 Other Data Domains



(a) Onedimensional (b) Multidimensional (c) Hierarchical

Figure 3.1: Different data domains

As already described in section 2.1 irregular data domains do not have a general arithemtic expression as an resolution concept.
Nevertheless, irregular data domains come in various shapes and layouts. The simplest one is onedimensional. As can be seen in fig. 3.1a, there is no regularity or general consent on how the data is scattered. Another data domain class is k-dimensional, also known as multidimensional. Those domains are often used in different areas of science and technology and stretch from extensive simulations of atmosphere and ocean climate to image processing and structural dynamics [SS94]. In fig. 3.1b a 2-dimensional irregular domain is illustrated. The complexity compared to a onedimensional domain increases proportionally.
The last class, often associated with irregular domains, is hierarchical. Examples of these data structures are image processing, computational geometry, geographic data, or robotics

[Sam84; CR03]. Thereby, each element represents a whole new set of data and elements. Hence, each individual hierarchical partition can be structured completely differently.

All three classes often have a significant overlap. Therefore, it is essential to be able to store and do calculations on these data sets efficiently. Different domain decomposition concepts are able to achieve this.

3.1.1 Onedimensional

Figure 3.1a shows a simple onedimensional irregular data structure. In programming, this could be an irregular and dynamic data structure like a hash table, linked list, or queue [CR95]. One common resolution concept is an index table: This structure has a pointer for each block of data stored within an array. Therefore, the data blocks can be indexed with a simple integer and later referenced.

Nevertheless, onedimensional irregular data domains can be interpreted as a special case of a k-dimensional data domain ($k = 1$).

3.1.2 Multidimensional

A typical two-dimension regular data domain is a simple matrix. One-dimensional index sets often represent those. The implementation is like the following: (i) The rows/columns of the matrix are saved sequential into memory, and (ii) The index is just a numerical sequence $(0, 1, \ldots, n)$ indexing every value of the matrix. Nevertheless, the index set can be two dimensional as well: The first dimension marks the start of every row/column in memory. The second index set marks the offset within the row/column.

Irregular multidimensional data domains are more complex. One example is a sparse matrix. To perform element resolutions of such an irregular domain, different storage concepts have been developed and are further described in section 3.2. Other irregular domains are weighted graphs, analogous systems, or manifolds [Shu+12].

In direct comparison, multidimensional regular and irregular domains have some similar problems. Storage in modern systems is mostly one dimensional. Hence, k-dimensional domains need to adapt to unique storage designs. Nevertheless, regular matrices designs are mostly simpler to implement, e.g., with *linear allocation*. In this approach, a domain is laid out "linearly by a nested traversal of the axes in some predetermined order" [SS94]. In general for multidimensional data domains, Sarawagi and Stonebraker [SS94] proposed different ideas for efficent organisation: (i) chunking, (ii) reordering, (iii) redundancy, and (iv) partitioning. These approaches can be later seen in different storage formats and distributions.

3.1.3 Hierarchical

Hierarchical data domains can but do not have to be irregular domains. The general data structure just indicates that there are multiple levels of data organized in a hierarchical manner. While every level can has its unique distribution attributes and intentions, at partitioning and mapping these data domains often behave similarly to other data domains. Therefore, it is essential to emphasize that there can be a significant overhead when dereferencing or restructuring elements, but this can be avoided by applying individual domain decomposition techniques to every level of the hierarchy.

3.2 Sparse Matrices

Sparse matrices are a common and significant aspect of progress in many scientific and industrial areas. These vary from Computational Circuit Design, Linear Programming, or Partial Differential Equations to specific topics such as regional Water Transport problems or Graph Theory [RW72]. In recent years those areas expanded to quantum physics, fluid dynamics, or structural mechanics [Kre+14]. Often these areas work with large eigenvalue problems or linear equations. When looking more profound in these problems a lot of computing time is invested into a multiplication of large sparse matrices with dense vectors which are also known as Sparse Matrix-Vector Multiplication (spMV).

Hence especially in High Performance Computing (HPC), there is a high priority to optimizing storage and computing designs/algorithms for sparse matrices. In this section, we discuss different popular sparse matrix storage formats and how those developed over time. These examples are building up in difficulty and efficiency. While discussing the advantages and problems, general construction and storage patterns are described and visualized. This will result in an in-depth description of the - for this thesis chosen - format *Sell-C-σ* .

3.2.1 Popular Sparse Matrix Storage Formats

During time there have been various approaches for efficient sparse matrix storage models. The simplest storage format is Coordinate List (COO) which just saves the sparse matrix in three arrays: (i) *nonZeroValues*: essential values which are not zero, and (ii/iii) *column index* / (ii) *row index* (row_{ptr}): coordinates for each essential element of. It is quite clear that the performance of this design is in the most cases quite low. [MLA10]

$$\begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 \\ 0 & 0 & a_{23} & 0 & a_{25} \\ 0 & 0 & 0 & 0 & 0 \\ a_{41} & 0 & a_{43} & 0 & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{pmatrix} \rightarrow$$

ess_{vals}	a_{11}	a_{14}	a_{23}	a_{25}	a_{41}	a_{43}	a_{45}	a_{55}
col_{ind}	1	4	3	5	1	3	5	5

row_{ptr}	0	2	4	4	7

Figure 3.2: Example of an Compressed Row Storage (CRS) design

Compressed Row Storage (CRS)

One fairly basic sparse matrix design is CRS. The general concept is to save only necessary elements. Therefore, it uses a three memory space design: (i) *essential values* (ess_{vals}): data domain for non-zero values within the matrix, (ii) *column index* (col_{ind}): data domain for the column index of each value, and (iii) *row pointer* (row_{ptr}): reference pointer for locations in the *essential elements* data domain that start a row [Bar+94]. A reference example in the style of arrays can be found in fig. 3.2.

The basic concept of CRS leads to some disadvantages. Firstly, there is no assumption of data structure and access patterns. Therefore, it is quite inefficient to access. Besides, cache utilization is quite low.

CRS already saves a significant amount of storage space. It requires only $(2 * N_{nonZeroValues} + N_{rows} + 1)$ (3.1) storage locations [SUC05] instead of $(N_{nonZeroValues} * 3)$ (3.2) in COO.

An analogous concept to CRS is Compressed Column Storage (CCS), whereby, instead of rows, columns are stored. Hence CCS is the CRS format for A^T. Furthermore, there are different varients of CRS for different assumptions about the sparsity structure of the matrix (e.g. Compressed Diagonal Storage, or Block Compressed Row Storage) [Bar+94].

Jagged Diagonal Storage (JDS)

JDS is also an varition of CRS and similar to the Compressed Diagonal format. It is especially designed for parallel and vector processors. The general idea is build an CRS for every row (see fig. 3.3b) and afterwards ordering them in decreasing order according to the amount non-zero elements in each row (see fig. 3.3c). During every step the permutations are saved as an additional single dimension array. Afterwards the coloumns are stored sequential in one array. The final JDS format can be seen in fig. 3.3d whereby the semicolons indicate the end of a jagged diagonal. The result is quite similar to the CRS schema but the additional array $perm_{vector}$ is needed to establish the original order of the rows. Moreover, $start_{pos}$ indicates the index of the jagged diagonals in the ess_{vals} array. [Bar+94; ME04]

$$\begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 \\ 0 & 0 & a_{23} & 0 & a_{25} \\ 0 & 0 & 0 & 0 & 0 \\ a_{41} & 0 & a_{43} & 0 & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{pmatrix}$$

(a) Original Matrix

$$\rightarrow \begin{pmatrix} a_{11} & a_{14} & 0 & 0 & 0 \\ a_{23} & a_{25} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ a_{41} & a_{43} & a_{45} & 0 & 0 \\ a_{55} & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Step 1

$$\rightarrow \begin{pmatrix} a_{41} & a_{43} & a_{45} & 0 & 0 \\ a_{11} & a_{14} & 0 & 0 & 0 \\ a_{23} & a_{25} & 0 & 0 & 0 \\ a_{55} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \rightarrow$$

(c) Step 2

ess_{vals}	a_{41}	a_{11}	a_{23}	$a_{55};$	a_{43}	a_{14}	$a_{25};$	a_{45}
col_{ind}	1	1	3	5	3	4	5	5

$start_{pos}$	0	4	7

$perm_{vector}$	4	1	2	5	3

(d) Step 3

Figure 3.3: Example of an Jagged Diagonal Storage (JDS) design

The storage space required by JDS is $(2 * N_{nonZeroValues} + N_{rows} + N_{jaggedDiagonals})$ (3.3) [SUC05] storage locations. Advantages and disadvantages of JDS are discussed in [Saa89].

ELLPACK

$$\begin{pmatrix} a_{11} & 0 & a_{13} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & a_{24} & 0 & 0 & 0 & a_{28} & 0 \\ 0 & a_{32} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{44} & 0 & a_{46} & 0 & 0 & 0 \\ 0 & 0 & a_{53} & 0 & 0 & 0 & 0 & a_{58} & a_{59} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_{69} \\ a_{71} & 0 & a_{73} & a_{74} & 0 & 0 & a_{77} & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{85} & 0 & 0 & a_{88} & 0 \\ a_{91} & a_{92} & 0 & a_{94} & 0 & 0 & 0 & a_{98} & 0 \end{pmatrix}$$

(a) Original Matrix

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ a_{21} & a_{24} & a_{28} & 0 \\ a_{32} & 0 & 0 & 0 \\ a_{44} & a_{46} & 0 & 0 \\ a_{53} & a_{58} & a_{59} & 0 \\ a_{69} & 0 & 0 & 0 \\ a_{71} & a_{73} & a_{74} & a_{77} \\ a_{85} & a_{88} & 0 & 0 \\ a_{91} & a_{92} & a_{94} & a_{98} \end{pmatrix}$$

(b) ELLPACK

$$\begin{pmatrix} a_{71} & a_{73} & a_{74} & a_{77} \\ a_{91} & a_{92} & a_{94} & a_{98} \\ a_{21} & a_{24} & a_{28} & 0 \\ \hline a_{53} & a_{58} & a_{59} & \\ a_{11} & a_{13} & 0 & \\ a_{44} & a_{46} & 0 & \\ \hline a_{85} & a_{88} & & \\ a_{32} & 0 & & \\ a_{69} & 0 & & \end{pmatrix}$$

(c) Sliced ELLPACK

Figure 3.4: Comparision of ELLPACK and Sliced ELLPACK designs

ELLPACK is a high-level system used for solving elliptic boundary value problems which

is mainly used as an extension to Fortran. Nevertheless, it delivers several problem solving modules whose theories can also be used in other languages. Therefore, the ELLPACK sparse matrix design is the main topic of ELLPACK in this thesis. The ELLPACK format and its variations is mainly known for its optimized performance on GPUs [Liu+13].

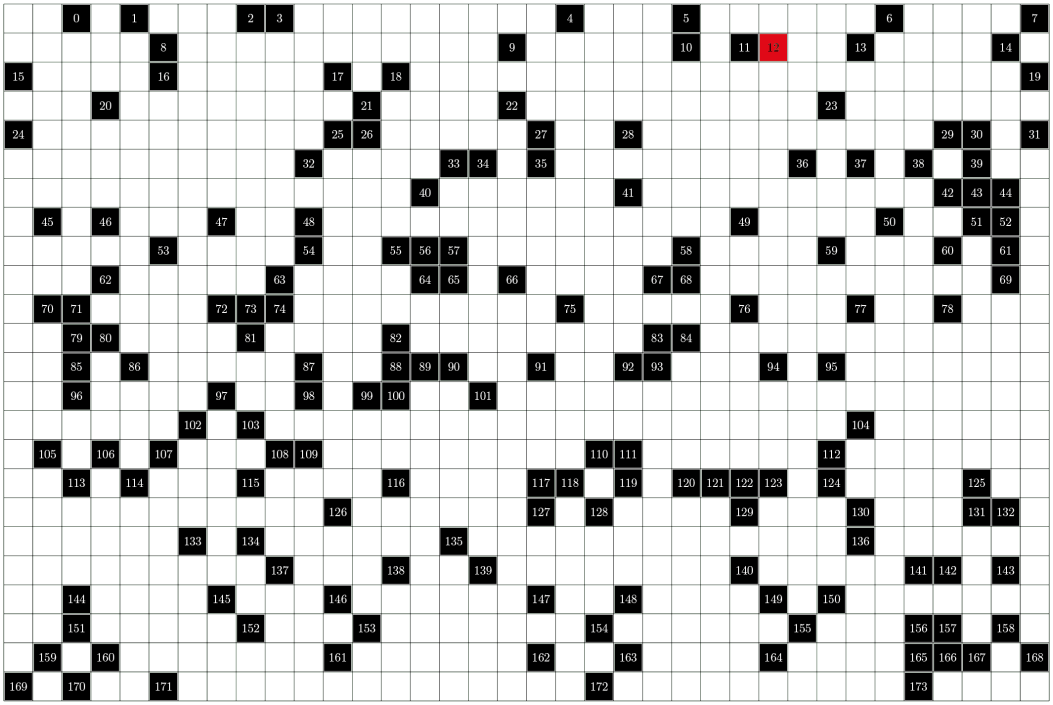
Similar to JDS within ELLPACK, the essential elements of a matrix are stored as a dense array in column-major order. Nevertheless, ELLPACK does not order the rows but introduces a *zero-padding*. In this concept, zeros are stored if necessary to obtain the same row length over the whole matrix (see fig. 3.4b). While this layout enables easy row-wise thread parallelization and more effective memory access, it introduces the problem of inefficiency when not all rows are similar in their length. Therefore, Monakov, Lokhmotov, and Avetisyan [MLA10] proposed an improved format called *Sliced ELLPACK*.
Sliced ELLPACK follows the concept of dividing the ELLPACK matrix in equally-sized *chunks* with C rows per chunks. Each of these slices is stored in a sperate ELLPACK matrix: they are zero-padded within the chunk and afterward stored consecutively in column-major order. This significantly reduces the storage overhead required and improves data locality. One further enhancement is the *Matrix Reordering*. This further advanced storage overhead by rearranging the rows descending by their amount of essential elements. A sample implementation of both aspects is shown in fig. 3.4c.

ELLPACK uses $(\max(N_{elementsInOneRow}) * N_{rows} * 2)$ (3.4) storage locations. Since Sliced ELLPACK is parametrized its storage space is not as easy to calculate:

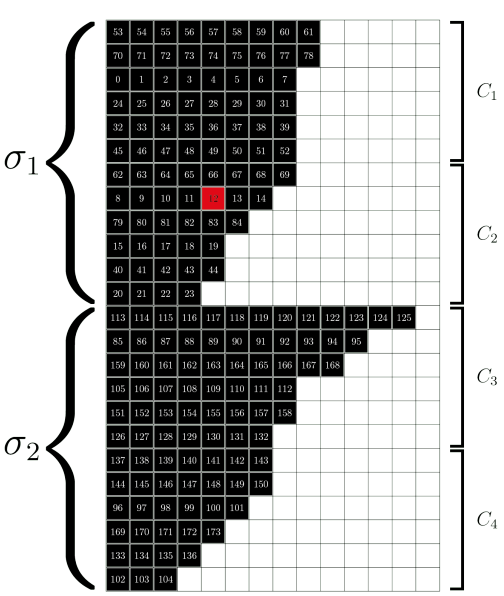
$$\sum_{chunk_{index}=0}^{\max(chunk_{index})} (\max(N_{elementsInOneRow}(chunk_{index})) * N_{rows}(chunk_{index}) * 2) + N_{rows} \quad (3.5)$$

3.2.2 *Sell-C-σ*

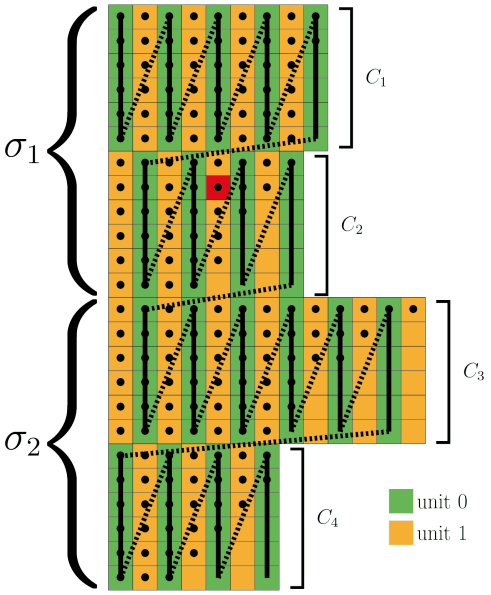
The *Sell-C-σ* data format was firstly introduced by Kreutzer et al. [Kre+14] in 2014 and is a "SIMD-friendly data format which combines long-standing ideas from General Purpose Graphics Processing Units (GPGPUs) and vector computer programming" [Kre+14]. It is derived from the already mentioned Sliced ELLPACK. Hence it is similar build as a single storage format and can be used in highly heterogeneous computer clusters. One differentiation is that *Sell-C-σ* (as its name suggests) can be optimized by two parameters: C and σ. [Kre+14]



(a) Original Matrix (36x24)



(b) Densification, Row ordering within σ, and chunk seperation



(c) Final visual layout with sequential storage layout (for unit 0)

Figure 3.5: *Sell-C-σ* reference design: *Sell*-6-12

The general structure is Sliced ELLPACK combined with SIMD vectorization. Hence, *Sell-C-σ* has some significant improvements. The transformation steps are like the following:

1. The original matrix is cut into equally-sized chunks of C rows, and all non-zero elements are removed.

2. Rows are reordered descending based on their lengths (number of non-zero elements) within the sorting scope σ. Usually, σ is a multiple of C, and therefore, rows are often interchanged between consecutive chunks. This leads to new chunks which can be seen in fig. 3.5b

3. The rows are zero-padded to match the length of its longest row within the chunk.

4. All elements within the chunk are stored in column-major order consecutively. All chunks are stored sequentially.

Parameters Optimization

Sell-C-σ has two parameters. In order to optimize those Kreutzer et al. [Kre+14] introduces the *chunk occupancy* β value which indicates the ratio between essential values (N_{NZ}) and total amount of stored values:

$$\beta = \frac{N_{NZ}}{\sum_{i=0}^{N_C} C * cl[i]} \qquad (3.6) \qquad cl[i] = \max_{k=iC}^{(i+1)C-1} rowLen[k] \qquad (3.7)$$

The worst-case scenario in *Sell-C-σ* is signaled by a minimum value of β, whereby one can find a maximum data transfer overhead. In contrast, the best case is $\beta = 1$, whereby no zero elements need to be stored within the whole *Sell-C-σ* matrix. This chunk occupancy indicator can be optimized by choosing an adequate sorting scope σ. Nevertheless, a high β value is not always the best case. For example: When there are N rows within a matrix, $\sigma = N$ (global sorting) can lead to a high β value, but there will be high access pattern penalties during computations. Besides, locality can be destroyed.

Kreutzer et al. [Kre+14] introduces a middle way with their *rule of thumb* of $\sigma = k * C$. In most cases, their studies reveal a "good enough" β by obtaining a good access structure. Reordering the rows adds computing effort but since it has only to be done once in preprocessing the relative overhead itself usually can be overlooked (especially in large matrices).

The parameters can be set to match the standard ELLPACK design ($C = N_{Rows};\ \sigma = 1$) or CRS ($C = 1;\ \sigma = 1$).

In conclusion, *Sell-C-σ* is the currently most advanced algorithm for efficiently storing and accessing sparse matrices. It makes use of three of the four proposed techniques proposed by Sarawagi and Stonebraker [SS94] (see section 3.1.2).

3.2.3 Sparse Matrix Operations

Section 2.6 presented some basic operations for dense data domains. These are now adopted to support sparse matrices and their various storage formats. While all formats could be used to implement those, this exploration concentrates primarily on *Sell-C-σ* .

Sparse Matrix-Vector Multiplication (spMV)

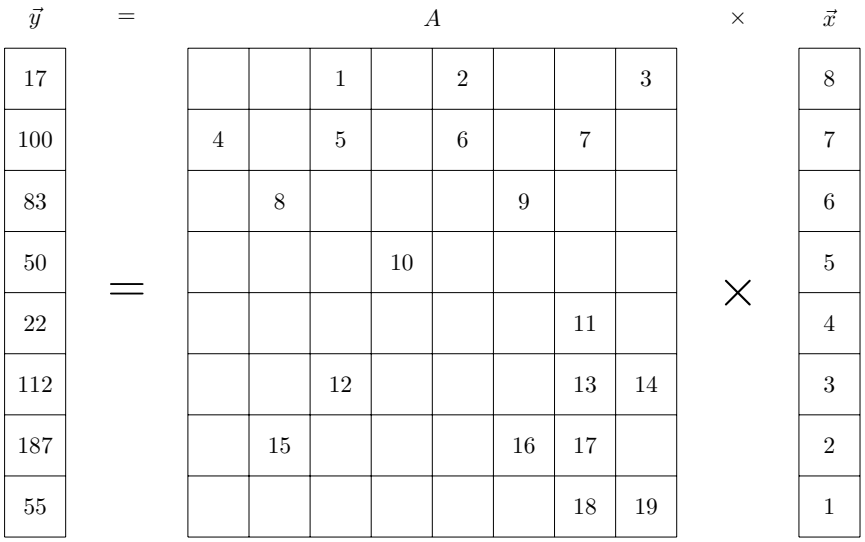


Figure 3.7: Sparse Matrix-Vector Multiplication

The Sparse Matrix-Vector Multiplication (spMV) operation is defined by BLAS exactly the same as for dense operations (see section 2.6.1). Therefore, eqs. (2.1) to (2.4) still hold.

Nevertheless, this process can be streamlined by not multiplying the zero values. For example the *Sell-C-σ* storage concept can achieve spMV (example in fig. 3.8) by the following steps:

1. Build *Sell-C-σ* for A.
2. Each row of $SCS(A)$ is multiplied elemtwise with the corresponding element of $\vec{x}$. Thereby, the column index (CI) of a_{ik} determines the element index of $\vec{x}$:

$$y_i = \sum_{k=1}^{len(a_i)} (a_{ik} \cdot x_{CI(a_{ik})}) \tag{3.8}$$

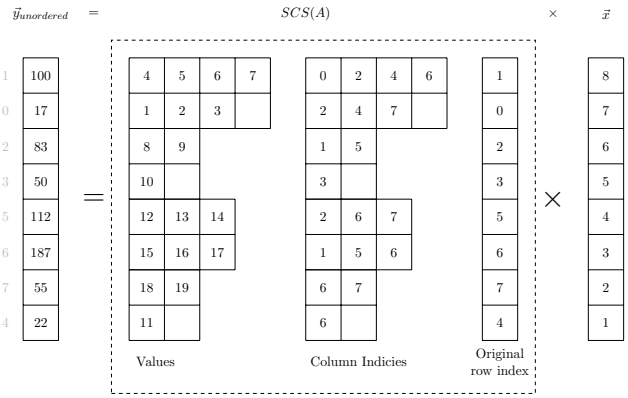


Figure 3.8: *Sell-C-σ* -Vector Multiplication without reordering
($C = 2; \sigma = 4$)

3. The resulting vector $\vec{y}$ needs to be reordered: The original row index ($orRow$) of A indicates thereby the final row index of $\vec{y}$.

A more detailed performance analysis and evaluation can be found in the master thesis of Ecker [Eck16].

Sparse Matrix-Sparse Matrix Multiplication (spMspMM)

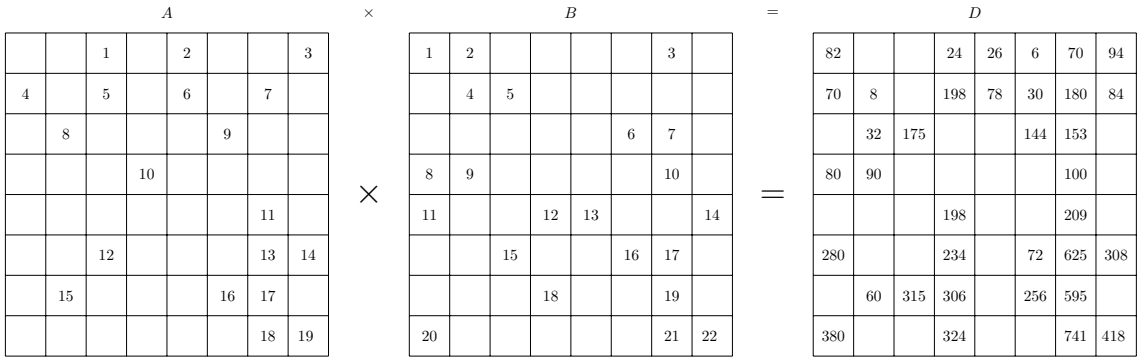


Figure 3.9: Sparse Matrix-Sparse Matrix Multiplication

An unmodified sparse matrix can be multiplied according to a dense matrix (see section 2.6.2). Nevertheless, there are many unnecessary computations (one of the multiplicated values is 0). Figure 3.10 is an example of an $8x8$ matrix multiplication. Within this, there are already 463 unnecessary computations. Only 49 multiplications are necessary to construct the resulting matrix. This overhead of roughly 90% is just for an $8x8$ matrix. Therefore, a more optimized solution is developed: Sparse Matrix-Sparse Matrix Multiplication (spMspMM) with the *Sell-C-σ* storage format.

The *Sell-C-σ* spMspMM for $A \times B = D$ consists of the following steps:

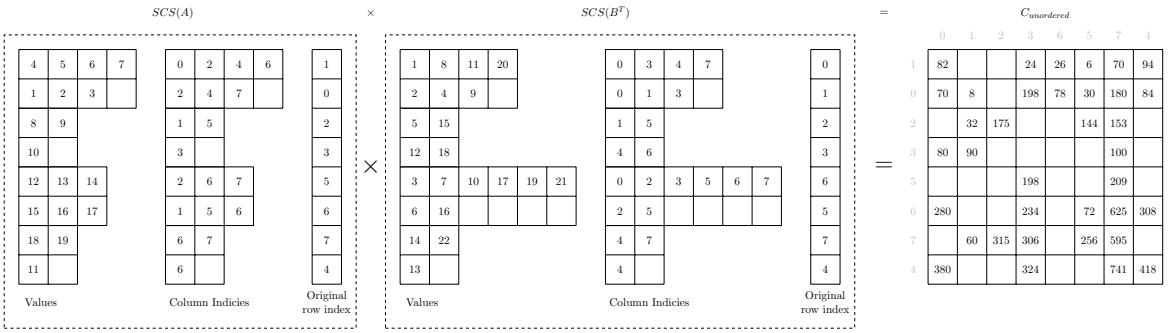


Figure 3.10: *Sell-C-σ* Multiplication without reordering
($C = 2; \sigma = 4$)

1. Transpose B.

2. Build *Sell-C-σ* for A and B^T. The parameters C and σ can be different.

3. Multiplicate each row of $SCS(A)$ with each row of $SCS(B^T)$. The elements of each row are only multiplicated when the column indicies (CI) match:

$$d_{ij} = \sum_{k=1}^{len(a_i)} \{(a_{ik}b_{jl}) \mid \forall l \in len(b_j) \ \wedge \ CI(a_{ik}) == CI(b_{il})\} \tag{3.9}$$

4. The resulting matrix D needs to be reordered: The original row index ($orRow(A)$) of A indicates thereby the row index of D, and the original row index of B^T indicates the column index of D.

For this approach, only the 49 necessary computations are done. However, there is a significant overhead of comparisons, reordering, and preparation of the second matrix. In a quick python implementation, a speedup of up to 3 was achieved (compared: manual dense matrix-matrix multiplication and *Sell-C-σ* -*Sell-C-σ* multiplication). Nevertheless, this process is not deeply researched so far and therefore, a decent performance increase can be expected in the future.

There are also some other already optimized concept for spMspMM e.g. Dalton, Olson, and Bell [DOB15], Buluç and Gilbert [BG12], and Liu and Vinter [LV14].

Slicing

An unmodified sparse matrix can be sliced in the same way as a dense matrix (described in section 2.6.3). Slicing a sparse matrix formatted into the *Sell-C-σ* design is quite a challenge. As can be seen in fig. 3.11 the following steps need to be taken into account:

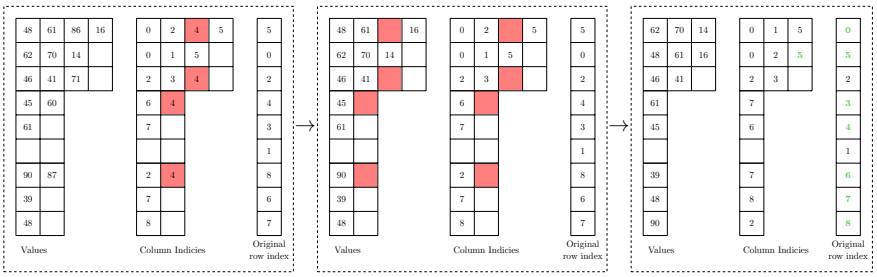


Figure 3.11: Slicing a column (4) within the matrix in a *Sell-C-σ* format

1. Iterate over *Column Indices* to identify every item which is a member of the to-be removed column.
2. Removing the element and the resp. entry in the *Column Indices*.
3. Rebuild each chunk with a modified entry. This includes rebuilding the chunk length, reorder the rows within the chunks (*Original row index*), and rebuilding the *Column Indices*.
4. Reorder chunks within scopes if necessary.

The example shown in fig. 3.11 only needs to be modified by the first three steps.

Slicing a row introduces new challenges as well:

1. Delete the row.
2. Rebuild the chunk with the modification and all subsequent chunks.
3. Reorder chunks within scopes if necessary.

3.3 Summary of Analysis of Irregular Decompositions

During the first half of this chapter, we discussed different data domains and their specific challenges. Especially, sparse matrices have an enormous speedup potential. In summary, there are different similarities which apply on sparse matrices:

- Irregular domains are often complex, non-trivial data structures with, e.g., unnecessary data, metadata overhead, and or non-displayable concepts.
- Irregular domains need to be simplified and reorganized to achieve a meaningful and efficient storage base. Therefore, temporary layers of restricted storage/computation formats need to be implemented.

- Each irregular domain class has a unique access pattern for different use cases. Hence, it is important to plan this beforehand.

- In most cases, during decomposition of irregular data domains, the user needs to assess whether storage size or access time should be optimized.

- Since irregular domains do not have a persistent resolution concept and layout design, the access concept is implicity.

- The indexing and referencing concept can or must be on multiple levels with different mapping interconnecting the different layers.

- Introducing storage formats for irregular domains can add huge complexity and overhead while applying operations.

3.4 Methodology

During the last chapters, different storage formats and operations have been developed and presented. Especially while exploring different operations on irregular domains, it has been differentiated in which challenges are given for specific formats and in more detail sparse domains. These experiences are used to extend and challenge the current state of the art methods presented in chapter 2.

The methodology of these results consisted of the following steps:

1. Understanding different concepts and ideas for regular data domains.

2. Exploring and defining one-/multidimensional and hierarchical irregular data domains to build up a theoretical base.

3. An in-depth analysis and review of sparse matrices and different storage concepts.

4. Applying operations on these sparse domains to find new attributes and properties which distinguish different concepts.

5. Combining analysis and experimental operations to general concepts.

In the following sections, this methodology is the base for new or modified concepts and ideas.

3.5 Classification of Domain Decomposition for Irregular Domains

Domain decomposition can be split up in different steps/layers. Firstly, a global canonical domain can be alternated in its shape, layout, or dimensions. This step is also called *Formatting*. Afterward, three additional steps are needed. *Partioning* is the method to split up the global, formatted view (index set) into an arbitrary number of logical blocks. After that, *Mapping* is used to distribute these blocks to available processing units. The final step *Layouting* is to arrange elements in a unit's physical memory.

Within the aspired abstraction model (further discussed in section 3.6) domain decomposition is splitted up in three general steps: *Formatting*, *Distribution* (partioning and mapping) and *Hardware* (layouting). For simplification partitioning and mapping is combined to *Distribution*.

A domain decomposition can be structured hierarchically. A partition or block may consist of a completely different domain decomposition concept. Therefore, it is possible that, for example, half of the partitions have a *row-major*, and the other half have a *column-major* layout in the memory. This induction can have an arbitrary number of layers, each with individual domain decomposition.

3.5.1 Formatting Properties

With the introduction of an additional layer within the concept, there are new properties:

Compression Data is compressed in some form by removing data or metadata.

Lossless Data is altered without losing information. Hence, it can be recreated at all times.

Shape Shape of the data structure is alternated.

Layout Layout of the data structure is altered.

Dense The domain is dense. Hence, no additional formatting step is required.

Latency aware Layout improves access latency for a specified use case.

Memory aware Layout improves memory access for a specified use case.

Metadata-dependent Group of elements/blocks/partitions need additional metadata (e.g. Column Indices or Row Pointer) to be interpreted.

These properties can depend on each other. For example, *Sell-C-σ* is a compressing, lossless storage format for sparse matrices, which alters the shape and layout of the data. In comparison, a submatrix of a dense matrix is compressing, lossy formatting, which alters the layout of the canonical data.

3.5.2 Partitioning and Mapping Properties

In section 2.5.1, some partitioning properties have been presented. After the investigation in the last section, the *balanced* property is redefined to enable a better representation:

k-regular balanced Workload for every class has the same number of elements (values). Mapping targets are associated with exactly one of k capacity classes with identical data volume in each class.

Symmetric balanced All partitions/blocks have an identical number of elements (non-zeros and possible padding elements).

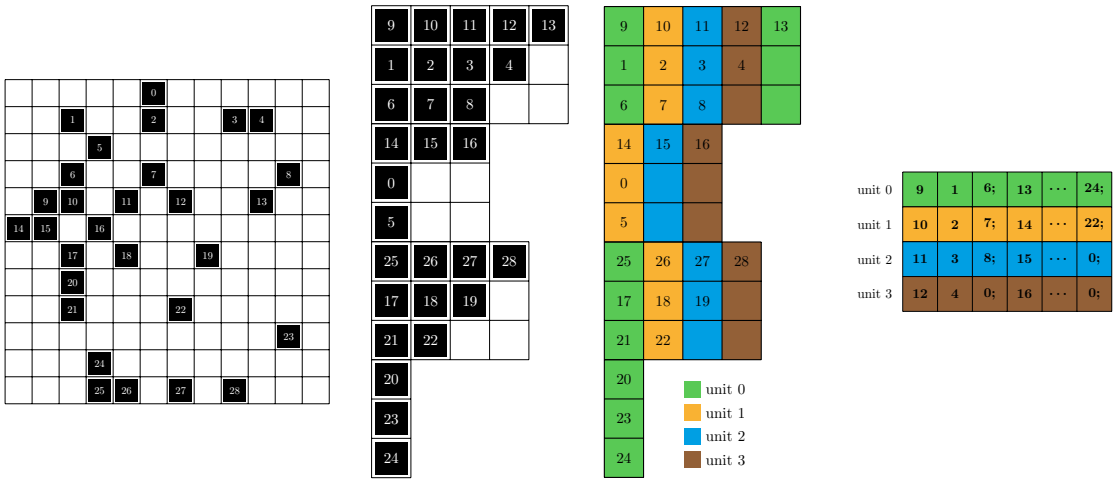
This amendment is vital since some storage formats introduce padding elements which can - depending on the algorithm/storage concept - be sometimes ignored.

In addition to the already in section 2.5.2 presented mapping properties, one newly attribute is added:

Conservitive compression Mapping of compressed partitioning patterns is lossless.

Nevertheless, the previously given mapping, partitioning, and layout properties still hold (see sections 2.5.1 to 2.5.3).

3.6 Abstraction



(a) Global Canonical Domain
Sparse matrix

(b) Formatting
Sell-C-σ
(Simplified)

(c) Decomposition
4 Processing Units

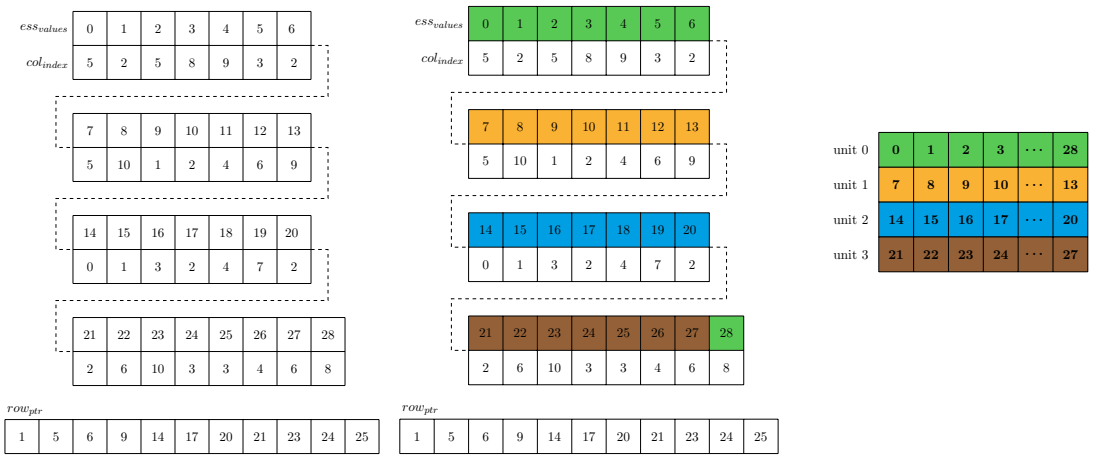
(d) Hardware

Figure 3.12: Fundamental four layer abstraction
(12×12 sparse matrix, *Sell*-3-6)

The last sections have shown different classification and storage concepts of data domains, domain decomposition, index sets, mapping, and layout. Combining all these findings, we propose a four-layer abstraction for multidimensional views on PGAS data domains. This concept is based on the concepts of Fuchs and Fürlinger [FF16b] and Unat et al. [Una+17] presented in section 2.7.

As seen in fig. 3.12, the highest level layer is the *Canonical View*, which displays the user's data domain. The second layer is the *Formating* layer. During this step, a more efficient view of the data domain is developed. These can be implemented as storage concepts, densification, or subviews. The third layer is *Decomposition*, whereby the formatted data domain is prepared for parallel access and distribution to the processing units and memory. The fourth layer is the *Hardware* realization. This layer symbolizes the successive steps to bring the data to the different hardware units by layouting the data within memory.
The layers are interconnected by index set mappings (further described in section 3.7) to maintain accessibility and computation methods. The different concepts for the first three layers will be discussed in chapter 3, and a reference implementation for sparse matrices is shown in chapter 4. The fourth layer is not within the scope of this thesis.

During the abstraction of irregular domains, one must assume that every step increases the complexity of the implementation. Each index set mapping needs to be carefully mod-



(b) Formatting CRS (c) Decomposition 4 Processing Units (d) Hardware

Figure 3.13: Example Abstraction with CRS

eled and enforced. Since each mapping adds an additional layer, each operator for these implementations is an entanglement of new dependencies. This will be deeper analyzed in chapter 4.

As already mentioned this concept is an extension of Fuchs and Fürlinger [FF16b] and Unat et al. [Una+17]. The contribution to this concept is the *Formatting* layer which enables the user to dissolve the irregularity and, afterwards, to use the regular *Partitioning-Mapping-Layout* pattern (see figs. 3.12c and 3.12d). It is worth mentioning that our abstraction can also be applied on regular data domains since the *Formatting* step can be implemented as a simple $1:1$ mapping which can be later ignored at compile-/run-time.

Within this thesis, this can be perfectly applied to sparse matrices. *Formatting* eliminates the bulk of sparse elements. Especially *Sell-C-σ* is an exceptional example since it has decent compression, and, besides, the concept provides a perfect base for efficient partitioning and mapping. It creates rectangular (optimized) chunks, which can be easily further processed with dense matrices techniques. This is possible, as the dense pattern properties provided by Fuchs and Fürlinger [FF16b] can be effortlessly applied to these rectangles.

Figure 3.12 is showing an example of this abstractation with *Sell-C-σ* . The same global canonincal domain (fig. 3.12a) is used to generate another example with CRS which is shown in fig. 3.13. In both examples after *Formatting* dense pattern properties can be applied. Nevertheless, *Sell-C-σ* is a 2-dimensional and CRS is 1-dimensional storage concept.

The first one has the following properties:

Formatting Compression, lossless, shape, layout, metadata-dependent

Partioning Rectangular, 2-dimensional, sequential, workload balanced, memory balanced

Mapping Blocked and partitioned sorted (row/chunk-length), somewhat uniform, multiple

Layout Column-major, linear

The CRS example has the following properties:

Formatting Compression, lossless, shape, layout, metadata-dependent

Partioning 1-dimensional, sequential, workload balanced, memory balanced

Mapping Order preservive, somewhat uniform, multiple

Layout Row-major, linear

One interesting observation is that although both examples have the same formatting properties, those describe different storage formats. This is due to the combination with the rest of the properties, which determines distinctive partitioning and mapping. Nevertheless, these two examples show that the presented abstraction can be used on divergent concepts. Furthermore, the same concept can differ in properties for specific use cases. For example, if the *Sell-C-σ* format never has to restore the original column or row order, *metadata-dependent* can be removed. This deduces that *Column Indices* and *Orignal row index* are not built. However, this concludes that it is no longer *lossless*.

All in all, the presented abstraction is capable of fulfilling all the required criteria to handle dense, hierarchical, sparse, and other data domains. The properties determine different concepts and architectural decisions. Therefore, a variety of domain decomposition and algorithms can be applied to data represented with this abstraction. Within an implementation of domain decomposition, the best possible outcome is achieved by providing the most and best-detailed properties. Meanwhile, an algorithm requires only the bare-bone necessary properties. The beneficial consequences for both are examined in chapter 4.

3.7 Index Set Mappings

When performing formatting, partitioning, decomposition, or memory layout on datasets, new value sets are built, which can differ from the old in element position or order. Hence, a concept needs to be developed to perform mappings between these sets to keep up the information flow. Index set mappings do this.

An index set I is defined as $\{A_i\}_{i \in I}$. To perform a mapping between the sets A_1 and A_2, a function needs be developed, which builds the index sets of both partitions to one another: $X(I_1) \rightarrow I_2$. Hence, one can predict and follow the different position of a value a within those sets.

A mapping can be one- or bidirectional. A one-way mapping only concludes that the position of the values in set A_1 can be tracked to their position in A_2. The other way around is not possible. This can be due to various reasons: information is lost on the mapping (e.g., reordering rows without saving their original position), or elements are deleted, e.g., due to compression.

A bidirectional mapping is defined as $X(I_1) \rightarrow I_2 \wedge I_1 \leftarrow Y(I_2)$. This can be done within sparse matrices storage concepts by memorizing the a and y positions of the original element. Since only unnecessary information (*zero*) are lost in CRS and others, these can be easily filled in afterward.

In summary, an index set is a set which contains objects that enumerates over objects in another set. An index set mapping is bijective.

3.7.1 Position concept

Consequently, from the index set concept, a position algebra/concept can be formulated. The *Position Concept* is used to determine the position of an element during each layer of abstraction. It provides the necessary algebra and function to access elements. Mostly this is closely related to the level of the index set. In general, there are different position levels which are generally needed:

Canonical Domain Position This is the position the user references to.

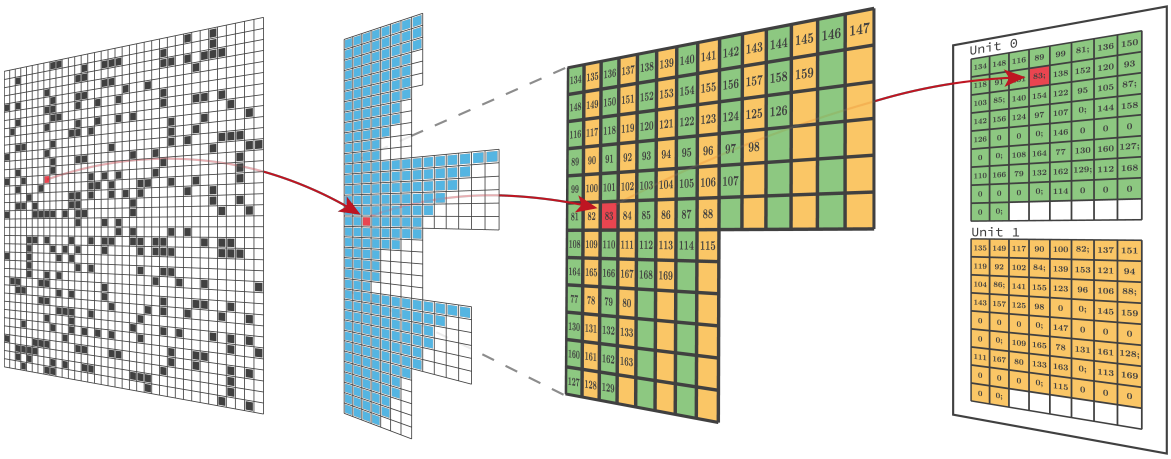
Global Position Global position is within the layer of formatting. The position determines where a value within the optimized data structure is for all distributed nodes.

Local Position Local position refers to a subset of the global domain, which is available at a layer of parallelization. This can be in a DASH team within a processing unit or a

specific local data domain.

Memory Position The memory position is the exact position of a value within the distributed memory.

There are no general rules for the format from a position. Nevertheless, it should be expressed as minimalistic as possible. That can differ from a 2-dimensional position for a matrix (x, y) or for *Sell-C-σ* as $(\sigma, C, \text{row}, \text{offset})$. Thereby, the implementation can vary highly. For *Sell-C-σ* , it can be implemented as a multidimensional quadruple or as a multilevel hierarchical tree structure.



(a) Global Canonical Domain (b) Formatting: *Sell-C-σ* (c) Decomposition: 2 Processing Units (Chunk 2 and 3) (d) Hardware: Memory View

Figure 3.14: *Sell-C-σ* reference index set mapping

An example index set mapping design with *Sell-C-σ* is given in fig. 3.14. The global canonical domain is a sparse matrix. The example follows the element $a = 83$ which has the index $I_{Global}(a) = (7, 13)$. Within the formatting step, *Sell-C-σ* is applied and a is now at position $I_{Formatting}(a) = (1, 0, 5, 2)$. The general index position is $(\sigma_{of\ a}, C_{chunk\ where\ a\ is}, \text{row}, \text{offset})$. In fig. 3.14c, the formatted data structure is partitioned in column-major order and linked to its corresponding processing unit. Since we have 2 units in this example the new position is now $I_{Decomposition}(a) = (0, 11)$ whereby 83 is distributed to unit 0 and its offset within the data is 11. The last mapping is to the memory at the processing unit. Therefore, the start of the memory block (memory address) and its offset within the block is used: $I_{Memory}(a) = (0\text{x}86540, 2)$.

3.8 Expressiveness

> *"As so often happens in computer science,*
> *we're willing to sacrifice efficiency for generality."*
> – Pedro Domingos, [Dom15]

The goal of this thesis is to extend an existing formal framework of dense domain decomposition by hierarchical and sparse data domain properties.

Therefore, an in-depth analysis of state-of-the-art sparse formats has been carried out. A vital feature of a formal framework is its expressiveness. This is achieved by developing a vocabulary to describe various formats with different properties. Those properties are the minimal ones required to define all characteristics of each format. The developed abstraction places these formats and properties in a higher-level approach to decompose the data domains in heterogenous computing clusters.

Nevertheless, this introduces a new problem: easy and efficient dereferencing of elements. Hence, the position concept has been formulated. This diminishes the complexity of the problem to a minimal level. While the advantages and a more detailed specification of positions are discussed in chapter 4, the general idea achieves a higher degree of expressiveness. An implementation is no longer dependent on specific implementations for local and global functions (e.g. views). This is accomplished and carried out by positions. Furthermore, the concept is still capable of the same features as the pointer and iterator concept.

Both concepts are combined into a formal language that reaches the set goal. It defines the necessary expressions, vocabulary, and structure to reduce the complexity to a minimum. This minimal language enables users to express their desired domain-specific intent while allowing arbitrary extension and new features.

4

Use Cases & Evaluation

In the last section, a general abstraction for irregular, sparse data domains has been presented. This knowledge is now utilized to explore different use-cases and opportunities to establish software architecture and examples. This includes a specified position concept and different approaches to make use of the advantages.

4.1 Position Concept

In this section, we explore some design examples for a position concept combined with the abstraction model shown in section 3.6.

Firstly, it is important to emphasize that the position concept model is an algebra, which implies the common pointer model. Nevertheless, there is a key difference:

While the pointer concept is integrated as a direct reference of an element as a scalar within the byte-addressable memory space, a position is multivariate and references an element with arbitrary representations of the element's physical memory location.

Putting it more clearly, a position can be pictured as a geographic location. For example, the entry door of the LMU main building is the specific location. Nevertheless, there are countless ways to describe where it is and or how a person can arrive. The simplest method is to give an address with the specific house number. Another description is to go out of the north left entry from the railway station University. Other ways are: (i) latitude and longitude coordinates, (ii) specific amount of meters, direction, and starting point, or (iii) what3words (coordinates based on words). All descriptions are valid to find the exact geographic location. The same idea applies to positions. An element with a physical memory location can be referenced in multiple ways, for example: (i) global/local coordinates, (ii) offset, (iii) index, or (iv) memory address.

Hence, the position concept's purpose is to achieve a higher degree of generality than the Global PGAS Pointer/Iterator concept. As a result, a position is hierarchically ignostic. It is not necessary or possible to recognize from the outside whether a position is global or local. Another key difference is the space where both concepts operate. The pointer concept is operating within the address space. The position concept is in the higher-level element

space while still being capable of operating in the address space (position implies pointer). Nevertheless, the position has access to the value's container concept and all corresponding patterns.
Besides, the position concept can be extended to support regions of elements (also called views). This can be done by combining two positions as start and end indicators.

Concepts definitions do not include instantiation, the original parameters used to define a position is not part of the concept. Any valid representation of an element is also a valid instantiation of a position, which again is compatible/acceptable with operations on any position instantiated by whatever means.

Further, it is crucial that a position does not need to be well defined since it can also imply unit/block/partitions/region-specific expressions. This implies possible dependencies on local parameters.

4.1.1 Operations

The position concept introduces a new set of symbols with its own vocabulary. Operations supplement this. Those are functions/actions which produce an output of symbols based on a logic and an input of symbols. Thus, operations map a set of valid symbols to a different set of valid symbols.

Nevertheless, attributes for those actions need to be postulated:

1. Each operation has to have at least one inverse operation.
2. Each operation has to have a corresponding neutral operation.
3. Each operation has to be combinable if the same operation is applied to the same position.

With these attributes, operations, symbols, and vocabulary a new algebra with its own arithmetic is proposed.

Various operators are available for the position concept. It is important to emphasize that each operation adds an overhead. Depending on the operation, the storage concept, and the domain, these are enormous or minuscule. In section 4.2, some operations with some storage concepts are examined. A few valid operations for positions are presented in table 4.1.

As already stated, each operation has an inverse and a neutral operation. An inverse element reverses the procedure/action, e.g., *global()* and *local()*. A neutral operator does not affect the outcome. Lastly, some operations can be combined with each other, e.g., twice the operation. Since those operation effects can be exploited for further operations, this is further explored in section 4.1.4.

Furthermore, one key advantage of a position is that an address resolution is to be performed first when an arithmetic expression is executed on a position's value. This is discussed in more detail in section 4.1.5.

Operation	Returns	Description
`global()`	Global Position	Expresses in position in the global domain
`local()`	Local Position	Expresses in position in the local domain
`offset()`	Offset in global/local domain	Expresses a position as an offset in the global/local domain
`address()`	Memory adress (scalar)	Expresses a position as a memory address in physical memory
`index()`	Index in global/local domain	Expresses a position as an index
`value()`	Element's value	Resolves the position
`shift<d>(v)`	Position	Shifts position in dimension `d` by `v`
`block()`	Region position	The whole local block of the current postion is selected
`sub<d>(v)`	Region position	Builds a subset of the current position in dimension `d` by `v`
`expand<d>(v)`	Region position	Expands region position in dimension `d` by `v`
`join(pos)`	Region position	Combines a position with another position `pos`
`split<d>(v)`	Two (region) postions	Splits a position into two positions in dimension `d` at `v`

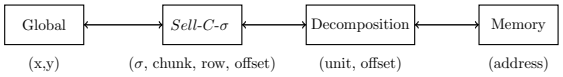
Table 4.1: Extract of Position Operations

4.1.2 Index Set Mappings

As stated in section 3.7, each layer of our abstraction is interconnected by index set mappings. These are based on value sets containing the index/position in layer A and there respective new index/position in layer B. The specific implementation differs from language to language and from concept to concept. We propose *lookup tables*, which saves the position/coordinate

of an element in A and maps those to an abstract position/coordinate in B.

Each conversion from *global* to *local* position includes diverse index set mappings. Hereby, a *local* position is defined as the position in the *Decomposition* layer. Figure 4.1 shows an example of such a mapping from global to memory with *Sell-C-σ* .



(a) *Sell-C-σ* index set mappings

Global	*Sell-C-σ*
...	...
(1,5)	(0,0,3,0)
(1,7)	(0,0,3,1)
...	...

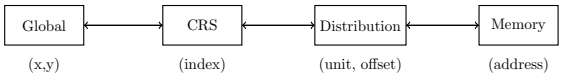
Sell-C-σ	Decomp.
...	...
(0,0,3,0)	(0,0)
(0,0,3,1)	(1,0)
...	...

Decomp.	Memory
...	...
(0,0)	0x24571
(1,0)	0x34864
...	...

(b) Example Lookup Tables

Figure 4.1: Index set mapping use-case for *Sell-C-σ*

Another case is illustrated in fig. 4.2. The CRS storage concept is quite different to *Sell-C-σ* . Nevertheless, both mappings only differ in the form of how the coordinate is saved. In both cases, *lookup tables* provided the necessary mapping capabilities.



(a) CRS index set mappings

Global	CRS
...	...
(1,5)	(5)
(1,7)	(6)
...	...

CRS	Decomp.
...	...
(5)	(1,3)
(6)	(0,2)
...	...

Decomp.	Memory
...	...
(1,3)	0x62158
(0,2)	0x15479
...	...

(b) Example Lookup Tables

Figure 4.2: Index set mapping use-case for CRS

If there is a need for the context of a storage concept, the implementation can simply look in the saved metadata of the storage. This is, e.g., for the column indices in *Sell-C-σ* or the row pointer in CRS. Though, all can be calculated or identified by the provided coordinates.

4.1.3 Composability

All these operations and mappings are composable. For example, in *C++* as pipe (|). This concludes to a constant folding at compile time and can be viewed in listing 1. The global position is created. Afterward, a region is created, and it is converted to a local position. Listing 2 shows the resulting code of the compiler. Nothing is calculated, but the variables are saved in static memory. The whole construct is then first resolved when the position is accessed. It is also possible that it grows when more operations are applied later.

In more detail, representation operations are idempotent which means that a object a fulfils the following property: $a \circ a = a$. In the context of positions, this is expressed in `local(LocalPosition) = LocalPosition` also listened in a code example in listing 3.

```
dash::Array<N> g_array;
// Global Position
auto pos = g_array.begin()
              | sub<1>(2, 20)
              | local();
```

Listing 1: Operations on position

```
LocalView<SubView<1, LocalPosition<2>>(
           LocalView<
              SubView<1, LocalPosition<2>
            >(pos, 2, 20))
```

Listing 2: Generated code

```
template <class PositionT> local(LocalPosition<PositionT> && lp) -> auto {
  return lp;
}
```

Listing 3: Idempotent local position conversion

4.1.4 Compile and Runtime Optimizations

The advantages of composability can be exploited at different occurrences within an application. The simplest is at compile-time when resolutions cancel out before accessing an element.

The following expressions show the most straightforward streamlining possibilities:

$$\texttt{local(local(pos))} \rightarrow \texttt{local(pos)} \tag{4.1}$$

$$\texttt{global(global(pos))} \rightarrow \texttt{global(pos)} \tag{4.2}$$

$$\texttt{local(global(local(pos)))} \rightarrow \texttt{local(pos)} \tag{4.3}$$

$$\texttt{global(local(global(pos)))} \rightarrow \texttt{global(pos)} \tag{4.4}$$

$$\texttt{shift<0>(shift<0>(pos, 10), 11)} \rightarrow \texttt{shift<0>(pos,21)} \tag{4.5}$$

$$\texttt{shift<0>(shift<0>(pos, 1), -1)} \rightarrow \texttt{pos} \tag{4.6}$$

$$\texttt{sub<0>(expand<0>(pos, 1), -1)} \rightarrow \texttt{pos} \tag{4.7}$$

Nevertheless, this can also be achieved at runtime, but this adds computing overhead. Therefore, it is recommended to optimize as much as possible at compile-time.

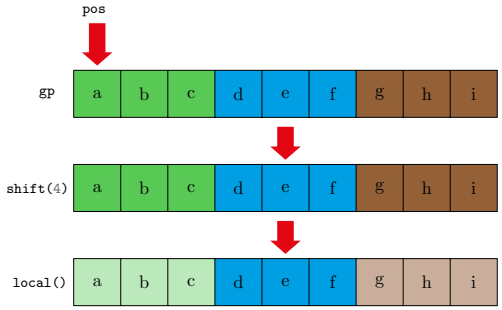
4.1.5 Address resolution

Since operators can be composed as a chain, a position is lazy binding. This means that a memory address of a position is only resolved when an arithmetic expression or read/write is applied to the element.

In combination with compile- and runtime optimizations this can be used to eliminate unnecessary overhead. This is illustrated in fig. 4.3. Figure 4.3a shows an exaggerated example of operations applied to an position. These can be folded down: (i) `global()` (line 7) cancels out `local()` (line 5), (ii) `index()` (lines 6 & 9) are not resolved since they are not used, and (iii) `shift(5)` (line 4) and `shift(-1)` (line 8) can be combined to `shift(4)` . The only necessary operation to receive an local position is `local()` (line 10). The optimized result can be found in section 4.1.5. Since all operation parameters are known this is done at compile-time.

```
1  dash::Array<N> g_array;
2  // Global position
3  auto pos = g_array.begin()
4                | shift<0>(5)
5                | local()
6                | index()
7                | global()
8                | shift<0>(-1)
9                | index()
10               | local();
```

(a) Original code



(b) Global canonincal data with indicatied position `pos`

```
1  dash::Array<N> g_array;
2  auto pos = g_array.begin()
3                | shift<0>(4)
4                | local();
```

(c) Optimized code (compile-time)

```
1  dash::Array<N> g_array;
2  auto pos = g_array.begin()
3                | shift<0>(4)
4                | local();
5  *pos = 'z';
```

(d) Code with memory access

Figure 4.3: Optimizations with the Position Concept

Figure 4.3b shows a simple memory space in the global domain. The elements are split into three balanced partitions for three units. As initialized firstly `pos` is a global position at the beginning of the data array. Afterward, it is shifted to the fourth position in the canonical domain (`shift(4)`). The last operation converts the global position to a local

position. As positions are multivariate, this process can be described, for example, with indexes: `pos` is at the global index 0. After shifting, it is at global index 4. Lastly, it is saved as a local position with the local index 1 (unit 1).

Nevertheless, `pos` is never actually resolved since there has been no access to the element. Figure 4.3d changes this, since it adds a write operation to the code (line 5).

4.1.6 Traits

Traits are a type class that provides a uniform interface to the properties of *Position* types. These describe different attributes of a position and allow them to implement algorithms only in terms of positions.

We propose the following `position_traits` :

Type	Definition	Example
`position_type`	The position type	
`value_type`	Type of referenced element value	`int`
`representation_type`	Type of arbitrary representation	`coordinate`
`difference_type`	Type resulting from subtracting two positions of type `pos`	`posdiff_t`
`is_global`	Indicates wheter `pos` is global or local	`true`
`balanced_class`	Indicates the class introduced by the k-regular balanced property (see section 3.5.2)	`2`

Table 4.2: Proposed position traits

4.2 Examples

This thesis explored different aspects of storage, distribution, and access concepts for irregular, sparse PGAS data domains. Additionally, in the last sections, the position concept and proposed implementations have been provided. All these insights are now put into practicability by introducing and examining different example implementations. These are expressed in *C++* in context of the *DASH* parallel programming framework. All codes examples are simplified for improved readability.

4.2.1 Shift Position

As already described in section 4.1, a position is able to depend on local parameters/variables. Therefore, one useful implementation is a shift position. This position can be passed, e.g., to every unit in the heterogeneous computing structure, and each one will resolute the position based on their local data. In this example, the position starts at their local beginning of data and shifts the position by 3 elements in the first dimension. This is exemplified in listing 4.

```
1  template <class PositionT>
2  class ShiftPosition
3    : public Position<PositionT>
4  {
5    private:
6      LocalPosition<PositionT> pos;
7    public:
8      shiftPosition(LocalPosition<PositionT> _pos)
9        : pos(_pos)
10     {}
11
12     auto operator* () {
13       return *(pos | shift<0>(3));
14     }
15     ...
16 }
```

Listing 4: Simple Shifted Position

This simple idea can be arbitrarily expended. In section 3.5.2 new properties have been introduced. k-regular balanced introduces different classes of balanced partitions. In an application different classes can be introduced for e.g. different computing units (e.g. host cpu or accelerators). Hence a `shiftPosition` can be implemented which shifts differently for every class. Therefore, the presented `position_traits` are used. This is exemplified in listing 5.

```
template <class PositionT>
class ShiftPosition
  : public Position<PositionT>
  enable_if< position_traits<PositionT>::balanced_class > 0 >
{
  private:
    LocalPosition<PositionT> pos;
    int balanced_class;
  public:
    shiftPosition(LocalPosition<PositionT> _pos)
      : pos(_pos)
    {
      balanced_class = dash::pattern_of(pos).balanced_class;
    }

    auto operator* () {
      return *(pos | shift<0>( 3 * balanced_class ));
    }
    ...
}
```

Listing 5: Shifted Position with different implementations for partition classes

Both `shiftPosition` are still capable of the general position concept. These examples show, in an elegant way, the expressive power of this concept. The new position is a modification of any other valid position. The arbitrary representation is not touched, yet it is widening the possibilities of referencing elements.

4.2.2 *Sell-C-σ* Position

One key advantage of positions is the ability to represent arbitrary concepts. Hence, it is useful to introduce a position for specific storage concepts. In this example, it is a position that is able to reference the values and column indices in *Sell-C-σ* . Hence, two internal positions are needed. We are assuming that *Sell-C-σ* is implemented in a way that it can return two positions to the start of the values and column indices memory position. Both are stored sequentially in storage. Furthermore, a `SellCSigmaHelper` is available, which is capable of calculating the original row of a value based on the value's position. Besides, the last element of the *Sell-C-σ* format is given. This can vary from the overall last element to the last element of the local distribution. This is exemplified in listing 6.

```
template <class PositionT>
class ScsPosition
{
  private:
    LocalPosition<PositionT> pos_value;
    LocalPosition<PositionT> ci_value;
    GlobalPosition<int64> row;
    GlobalPosition<PositionT> last_in_scs;
  public:
    current
  public:
    scsPosition(LocalPosition<PositionT> _pos_value,
                LocalPosition<PositionT> _ci_value,
                GlobalPosition<PositionT> _last)
      : pos_value(_pos_value)
      , ci_value(_ci_value)
      , last_in_scs(_last)
    {
      row = SellCSigmaHelper.getOriginalRow(pos_value);
    }

  auto operator++() {
    pos_value = pos_value | shift<0>(1);
    ci_value = ci_value | shift<0>(1);
    row = SellCSigmaHelper.getOriginalRow(pos_value);
    return *this;
  }

  auto value() {
    return *pos_value;
  }

  auto colum_index() {
    return *ci_value;
  }

  auto orignal_row() {
    return *row;
  }

  bool isLast() {
    return (last_in_scs == pos_value);
  }
  ...
}
```

Listing 6: Position for *Sell-C-σ*

In addition, wie introduce specific operations for `ScsPosition` :

Operation	Return	Definition
`colum_index()`	Column Index	Resolve the original column index of the *Sell-C-σ* element
`orignal_row()`	Row Index	Resolve the original row index of the *Sell-C-σ* element
`isLast()`	Boolean	Determines whether the position is the last within the *Sell-C-σ* object

Table 4.3: *Sell-C-σ* position operations

4.2.3 Sparse Matrix-Vector Multiplication (spMV)

Section 3.2.3 discussed a spMV with the storage concept *Sell-C-σ* . This example is now expanded to work with the presented abstraction and the position concept. On behalf of simplicity, we assume the sparse matrix is already stored as *Sell-C-σ* and distributed to DASH units. A copy of the vector $\vec{x}$ is stored on every unit's memory. For this example, the `ScsPosition` introduced in listing 6 is used to access values and metadata of the saved sparse matrix in *Sell-C-σ* format.

During the algorithm, each unit calculates its local partitions of the *Sell-C-σ* sparse matrix. Since each element of the input vector $\vec{x}$ is potentially accessed, a local copy is recommended. After all, units have finished their designated work, a `dash::all_reduce` is applied to combine the local results. The resulting `global_result_vector` is the final result of the Sparse Matrix-Vector Multiplication (spMV).

Details and specifications on the implementation with DASH can be found in section 2.3 and the linked references.

```
template <class T>
std::vector<T> spMVM(
    const std::vector<T> input_vector,
    const sellCSigma<T> matrix)
{
  dash::array<T> local_result_vector(std::size(input_vector), 0);
  auto current = ScsPosition<T>(std::begin(matrix)) | local();
  do {
    local_result_vector.at(current | orignal_row()) =
        (*current)
      * input_vector.at(( current | colum_index() ));
    current++;
  } while(!current.isLast())
  dash::Team::All().barrier();

  std::vector<T> global_result_vector(std::size(input_vector), 0);
  dash::all_reduce(
      std::begin(local_result_vector), std::end(local_result_vector),
      std::begin(global_result_vector), std::end(global_result_vector),
      std::plus<T>()
  );

  return global_result_vector;
}
```

Listing 7: spMV with position concept, DASH and *Sell-C-σ*

5 Summary & Conclusion

This thesis explored the research question of whether it is possible to abstract conceptual state-of-the-art approaches to sparse data storage to improve adaptation of domain-specific computational intent to underlying hardware. Hence, chapter 2 established the base knowledge to understand data domains and their classifications. In more detail, different data domains decomposition steps have been presented, and their properties are explained. Furthermore, crucial related work has been presented. Notably, the work of Fuchs and Fürlinger [FF16b] is the base for an exploration of a similar concept for sparse data domains.
In chapter 3, the basic knowledge is extended by introducing other data domains and their challenges. An in-depth look is emphasized on different popular sparse matrix storage formats, whereby *Sell-C-σ* is identified as one of the leading designs. Based on these insights, the main contribution is presented: Classification of irregular domain decomposition for sparse matrices. Thereby, it is concluded that the concept of Fuchs and Fürlinger [FF16b] needs to be extended by another layer called *Formatting*, which introduces new properties. The resulting abstraction is capable of handling all challenges of irregular, sparse data domains while still being applicable to regular domains. This extension comes with the challenge of more complicated mappings and calculations. Therefore, a new concept to address resolution is presented: the *Position Concept*.
The new abstraction and especially the presented *Position Concept* are explored in chapter 4 by applying different software practices to them. This practical approach examines the various challenges and opportunities in using the concept and the resulting algebra.

5.1 Revisiting the objective

Sparse matrices are the main objective of this thesis. In section 3.2.1 COO, CRS, CCS, JDS, ELLPACK, and *Sell-C-σ* have been analyzed. While all concepts have their own advantages and drawbacks, *Sell-C-σ* is the leading one for most applications.

Furthermore, as a state-of-the-art property classification system, the work of Fuchs and Fürlinger [FF16b] has been chosen and extended by a generally applicable formatting layer. The resulting properties have been developed on the insights of the various sparse matrices

storage formats. This enables an entirely new area of approaches. Nevertheless, these properties can be extended for other use-cases as well.

One essential contribution is the algebra of the position concept. Although this can be applied to dense data domains, it is especially helpful for irregular domains since it can be arbitrarily extended. This is shown in listing 6 when developing a specialized position based on the *Sell-C-σ* storage format. The vital factor is that this design allows all the advantages of positions while enabling specialized operators for easier access to elements and metadata in *Sell-C-σ* . Furthermore, the position concept can be used to realize the index set mappings, e.g., by connecting different positions via *lookup tables*.

5.2 Conclusion

Coming back to the given main research question:

Can conceptual state-of-the-art approaches to sparse data storage be abstracted to improve adaptation of domain-specific computational intent to underlying hardware?

The presented abstraction has shown that it is perfectly possible to design a general, adaptive concept to support sparse data storage while expressing computational intent. The four-layer system is capable of interchangeable storage formats (*Formatting*) and an exhaustive mapping to the subsequent partitioning and mapping (*Decomposition*). With the given related work the abstraction can be expressed as enhanced software architecture.

In conclusion, the position concept implies all the features of the global pointer concept. However, it eliminates its limitations by providing an abstract representation and operation within the element space. This enables a user to express computational intent and be more versatile while implementing a solution. By explicitly stating different storage formats and decomposition patterns, an algorithm can be optimized for the underlying hardware.

5.3 Recommendations & Future Work

The results of this thesis apply to a large variety of problems in irregular domains. Nonetheless, there are many other formats and concepts to deal with those domains. As already mentioned, the developed abstraction, the position concept, and the index set algebra is designed to be extended with those in mine. It is also recommended to examine other use-cases like Graphs, which are not stored as sparse matrices. These should be realizable with the abstraction and new properties as well.

Another recommendation is the combination of the concepts with more information about the hardware. Fuchs and Fürlinger [FF18] presented a graph-based approach to provide logical hardware topologies and high-level operations for dynamic, distributed hardware locality during run-time. This representation of hardware can be used to uniformly and algorithm-specifically distribute the data and processing workload on the given hardware. Linking this with the presented properties, hardware topology can be exploited and logical structures individualized.

Furthermore, another relevant research topic can be execution policies within the given concepts. When optimizing algorithms, operations are often reordered for processing. This can be further improved when giving information about formatting, distribution, mapping, layout, and hardware capabilities to the compiler or run-time. These additional details represent different advantages for processing unit or domain-specific processing intent.

This thesis has not done any exhaustive implementation or performance benchmarking. Both aspects are recommended to be done shortly within the DASH framework. Through the general approach and the specific examples in chapter 4, an excellent base to implement other aspects has been presented. Mainly, the `positions_traits` can be used to implement specialized algorithms for GPUs, CPUs, and accelerators. For specific sparse matrices implementations, the given resources and literature should be further examined to receive the best performance possible.

List of Figures

List of Listings

Acronyms

BLAS Basic Linear Algebra Subprograms. 14, 27

CAF Coarray Fortran. 2, 16

CCS Compressed Column Storage. 22, 53

COO Coordinate List. 21, 22, 53

CRS Compressed Row Storage. i, 22, 26, 35–37, 44, 53

DART DAsh RunTime. 9, 10

GPGPUs General Purpose Graphics Processing Units. 24

HPC High Performance Computing. vii, 1, 3, 4, 8, 9, 15–17, 21

JDS Jagged Diagonal Storage. i, 22–24, 53

MPI Message Passing Interface. 9

PADAL Programming Abstractions for Data Locality. 17

PDE Partial Differential Equation. 8, 9

PGAS Partitioned Global Address Space. 1–3, 9, 10, 17, 34, 41, 47

RDMA Remote Direct Memory Accesses. 2

RMA Remote Memory Access. 9

SPMD Single Program, Multiple Data. 2, 3, 9, 16

spMspMM Sparse Matrix-Sparse Matrix Multiplication. 28, 29

spMV Sparse Matrix-Vector Multiplication. iii, x, 21, 27, 51, 52

STL Standard Template Library. 9, 10, 17

UPC Unified Parallel C. 3, 17

Bibliography

[202a] *Chapel: Domain Maps (Layouts and Distributions)*. [Online; accessed 18. Jun. 2020]. 2020. URL: https://chapel-lang.org/tutorials/SC11/SC11-6-DomainMaps.pdf.

[202b] *Chapel: Productive Parallel Programming*. [Online; accessed 18. Jun. 2020]. 2020. URL: https://chapel-lang.org.

[Alm11] George Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1539–1545. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_210. URL: https://doi.org/10.1007/978-0-387-09766-4_210.

[Bar+] Christopher Barton et al. "Multidimensional Blocking in UPC". In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, pp. 47–62. DOI: 10.1007/978-3-540-85261-2_4. URL: https://doi.org/10.1007%2F978-3-540-85261-2_4.

[Bar+94] Richard Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. DOI: 10.1137/1.9781611971538. URL: https://doi.org/10.1137%2F1.9781611971538.

[BCA07] Christopher Barton, Călin Caşcaval, and José Nelson Amaral. "A Characterization of Shared Data Access Patterns in UPC Programs". In: *Languages and Compilers for Parallel Computing*. Ed. by George Almási, Călin Caşcaval, and Peng Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 111–125. ISBN: 978-3-540-72521-3.

[BG12] Aydin Buluç and John R. Gilbert. "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments". In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C170–C191. DOI: 10.1137/110848244. URL: https://doi.org/10.1137%2F110848244.

[CC11] Bradford L. Chamberlain and Sung-Eun Choi. "Authoring User-Defined Domain Maps in Chapel". In: 2011.

[CCZ07] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. "Parallel programmability and the chapel language". In: *International Journal of High Performance Computing Applications* 21 (2007), pp. 291–312.

[CFZ94] Barbara M. Chapman, Thomas Fahringer, and Hans P. Zima. "Automatic support for data distribution on distributed memory multiprocessor systems". In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 184–199. ISBN: 978-3-540-48308-3.

[Cha+00] Bradford Chamberlain et al. "ZPL: A Machine Independent Programming Language for Parallel Computers." In: *IEEE Trans. Software Eng.* 26 (Mar. 2000), pp. 197–211. DOI: 10.1109/32.842947.

[CR03] I. F. Cruz and A. Rajendran. "Semantic data integration in hierarchical domains". In: *IEEE Intelligent Systems* 18.2 (2003), pp. 66–73.

[CR95] Bertrand Le Cun and Catherine Roucairol. "Concurrent Data Structures for Tree Search Algorithms". In: *Parallel Algorithms for Irregular Problems: State of the Art*. Ed. by Afonso Ferreira and José D. P. Rolim. Boston, MA: Springer US, 1995, pp. 135–155. ISBN: 978-1-4757-6130-6. DOI: 10.1007/978-1-4757-6130-6_7. URL: https://doi.org/10.1007/978-1-4757-6130-6_7.

[DOB15] Steven Dalton, Luke Olson, and Nathan Bell. "Optimizing Sparse Matrix—Matrix Multiplication for the GPU". In: *ACM Trans. Math. Softw.* 41.4 (Oct. 2015). ISSN: 0098-3500. DOI: 10.1145/2699470. URL: https://doi.org/10.1145/2699470.

[Dom15] Pedro Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. New York: Basic Books, 2015. ISBN: 978-0-465-06570-7.

[DT20] Dash-Team. *NArray - DASH*. [Online; accessed 5. Jun. 2020]. 2020. URL: https://dash.readthedocs.io/en/latest/Containers/NArray/#the-dash-multidimensional-array-narray.

[Eck16] Jan Philipp Ecker. "Requirement Analysis and Realization of Efficient SparseMatrix-Vector Multiplications on Different ProcessorArchitectures". [Online; accessed 18. Jul. 2020]. MA thesis. HochschuleBonn-Rhein-Sieg - Computer Science Department, Nov. 2016. URL: https://berrendorf.inf.h-brs.de/lehre/abschlussarbeiten/arbeiten/2016_Master_Thesis_Ecker.pdf.

[EG+05] T. El-Ghazawi et al. *UPC: Distributed Shared Memory Programming*. Wiley Series on Parallel and Distributed Computing. Wiley, 2005. ISBN: 9780471478379. URL: https://books.google.de/books?id=n4pknjxmh7EC.

[FF16a] Tobias Fuchs and Karl Fürlinger. "A Multidimensional Distributed Array Abstraction for PGAS". In: Dec. 2016. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0150.

[FF16b] Tobias Fuchs and Karl Fürlinger. "Expressing and Exploiting Multi-Dimensional Locality in DASH". In: vol. 113. Jan. 2016, pp. 341–359. ISBN: 978-3-319-40526-1. DOI: 10.1007/978-3-319-40528-5_15.

[FF18] Tobias Fuchs and Karl Fürlinger. "Runtime Support for Distributed Dynamic Locality". In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by Dora B. Heras et al. Cham: Springer International Publishing, 2018, pp. 167–178. ISBN: 978-3-319-75178-8.

[FFK16] K. Fuerlinger, T. Fuchs, and R. Kowalewski. "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms". In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Dec. 2016, pp. 983–990. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140.

[FL10] H. Froning and H. Litz. "Efficient hardware support for the Partitioned Global Address Space". In: *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 2010, pp. 1–6.

[For] *FORTRAN*. [Online; accessed 20. May 2020]. American Heritage Dictionary of the English Language (5 ed.). The Free Dictionary. 2011. URL: https://www.thefreedictionary.com/FORTRAN.

[For01] BLAS Technical Forum. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*. Aug. 2001. URL: https://www.netlib.org/blas/blast-forum.

[Für+14] Karl Fürlinger et al. "DASH: Data Structures and Algorithms with Support for Hierarchical Locality". In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by Luís Lopes et al. Cham: Springer International Publishing, 2014, pp. 542–552. ISBN: 978-3-319-14313-2.

[Für+18] Karl Fürlinger et al. "Investigating the Performance and Productivity of DASH Using the Cowichan Problems". In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. Tokyo, Japan, Jan. 2018.

[GR06] Pierre Gosselet and Christian Rey. "Non-overlapping domain decomposition methods in structural mechanics". In: *Archives of Computational Methods in Engineering* 13.4 (2006), pp. 515–572. DOI: 10.1007/bf02905857. URL: https://doi.org/10.1007%2Fbf02905857.

[Hai09] Jean-Luc Hainaut. "Hierarchical Data Model". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1294–1300. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_189. URL: https://doi.org/10.1007/978-0-387-39940-9_189.

[Hal17] P.R. Halmos. *Naive Set Theory*. Dover Books on Mathematics. Dover Publications, 2017. ISBN: 9780486821153. URL: https://books.google.de/books?id=bWu9DgAAQBAJ.

[Hud] David E. Hudak. *Introduction to the Partitioned Global Address Space (PGAS) Programming Model*. [Online; accessed 18. Jun. 2020]. URL: https://www.osc.edu/sites/osc.edu/files/staff_files/dhudak/pgas-tutorial.pdf.

[KFS18] Christoforos Kachris, Babak Falsafi, and Dimitrios Soudris. *Hardware Accelerators in Data Centers*. Springer, Cham, 2018. ISBN: 978-3-319-92791-6. DOI: 10.1007/978-3-319-92792-3.

[Kre+14] Moritz Kreutzer et al. "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units". In: *SIAM Journal on Scientific Computing* 36.5 (2014), C401–C423. ISSN: 1095-7197. DOI: 10.1137/130930352. URL: http://dx.doi.org/10.1137/130930352.

[Lar+11] Rafael Larrosa et al. "A First Implementation of Parallel IO in Chapel for Block Data Distribution". In: *PARCO*. 2011.

[Liu+13] Xing Liu et al. "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors". In: June 2013, pp. 273–282. ISBN: 9781450321303. DOI: 10.1145/2464996.2465013.

[Los01] D. Loshin. *Enterprise Knowledge Management: The Data Quality Approach*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2001. ISBN: 9780124558403. URL: https://books.google.de/books?id=3BXTfCtR8zsC.

[LV14] W. Liu and B. Vinter. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 370–381.

[MC+09] John Mellor-Crummey et al. "A New Vision for Coarray Fortran". In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS '09. Ashburn, Virginia, USA: Association for Computing Machinery, 2009. ISBN: 9781605588360. DOI: 10.1145/1809961.1809969. URL: https://doi.org/10.1145/1809961.1809969.

[ME04] Euripides Montagne and Anand Ekambaram. "An optimal storage format for sparse matrices". In: *Information Processing Letters* 90.2 (2004), pp. 87–92. DOI: 10.1016/j.ipl.2004.01.014. URL: https://doi.org/10.1016%2Fj.ipl.2004.01.014.

[MLA10] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures". In: Jan. 2010, pp. 111–125. DOI: 10.1007/978-3-642-11515-8_10.

[Mun00] James Munkres. *Topology*. Upper Saddle River, NJ: Prentice Hall, Inc, 2000. ISBN: 9780131816299.

[Mös+18] Felix Mössbauer et al. "A Portable Multidimensional Coarray for C++". In: *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2018)*. Cambridge, UK, Mar. 2018.

[NR98] Robert W. Numrich and John Reid. "Co-Array Fortran for Parallel Programming". In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), 1–31. ISSN: 1061-7264. DOI: 10.1145/289918.289920. URL: https://doi.org/10.1145/289918.289920.

[Num11] Robert W. Numrich. "Coarray Fortran". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 304–310. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_477. URL: https://doi.org/10.1007/978-0-387-09766-4_477.

[RW72] Donald J. Rose and Ralph A. Willoughby, eds. *Sparse Matrices and their Applications*. Springer US, 1972. DOI: 10.1007/978-1-4615-8675-3. URL: https://doi.org/10.1007/978-1-4615-8675-3.

[Saa89] Youcef Saad. "Krylov subspace methods on supercomputers". In: *Siam Journal on Scientific and Statistical Computing* 10 (1989), pp. 1200–1232. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19890017045.pdf.

[Sam84] Hanan Samet. "The Quadtree and Related Hierarchical Data Structures". In: *ACM Comput. Surv.* 16.2 (June 1984), 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: https://doi.org/10.1145/356924.356930.

[SBG04] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 2004. ISBN: 9780521602860. URL: https://books.google.de/books?id=dxwRLu1dBioC.

[Sch69] H. A. Schwarz A. Schwarz. "Ueber einige Abbildungsaufgaben." In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1869.70 (1869), pp. 105–120. DOI: 10.1515/crll.1869.70.105. URL: https://doi.org/10.1515%2Fcrll.1869.70.105.

[Shu+12] David I Shuman et al. "The Emerging Field of Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and Other Irregular Domains". In: (2012). DOI: 10.1109/MSP.2012.2235192. eprint: arXiv:1211.0053.

[Sny99] L. Snyder. *A Programmer's Guide to ZPL*. Scientific and engineering computation. MIT Press, 1999. ISBN: 9780262692175. URL: https://books.google.de/books?id=kfAcHfLHlacC.

[SS94] S. Sarawagi and M. Stonebraker. "Efficient organization of large multidimensional arrays". In: *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. 1994, pp. 328–336.

[SUC05] R. Shahnaz, A. Usman, and I. R. Chughtai. "Review of Storage Techniques for Sparse Matrices". In: *2005 Pakistan Section Multitopic Conference*. 2005, pp. 1–7.

[Yel+07] Katherine Yelick et al. "Productivity and performance using partitioned global address space languages". In: Jan. 2007, pp. 24–32. DOI: 10.1145/1278177.1278183.

[Con20] Contributors to Wikimedia projects. *Ddm original logo - Schwarz alternating method - Wikipedia*. [Online; accessed 10. Jun. 2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Schwarz_alternating_method&oldid=959441699.

[Una+17] D. Unat et al. "Trends in Data Locality Abstractions for HPC Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020.

[Zhe+14] Y. Zheng et al. "UPC++: A PGAS Extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1105–1114.