The Dissertation Committee for Martin Daniel Schatz

certifies that this is the approved version of the following dissertation:

Distributed Tensor Computations: Formalizing Distributions, Redistributions, and Algorithm Derivation

Committee:

Robert A. van de Geijn, Supervisor

Tamara G. Kolda, Co-Supervisor

John F. Stanton

Keshav Pingali

Jeff R. Hammond

Don S. Batory

Distributed Tensor Computations:
Formalizing Distributions, Redistributions, and Algorithm Derivation

by

Martin Daniel Schatz, B.S.C.S.; B.S.Ch.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2015

Dedicated to my mother.

Acknowledgments

Upon entering graduate school I knew that I wanted to conduct research related to both chemistry and computer science. Ultimately, I focused my efforts in the domain of tensor computations. It was due to Prof. Robert van de Geijn's previous experience, insights, and formal approach to the domain of high-performance dense linear algebra that I was able to gain intuition to extend key ideas to the domain of tensor computations. An initial theory was developed to express the intuition for distributed tensor computations thanks to the countless hours spent at the whiteboard discussing with Dr. Tze Meng Low. Once the initial theory was developed, Dr. Devin Matthews provided the practical application to make the theory meaningful. At this point, we could justify the madness and, thanks to Devin's unflappable demeanor, we understood the intricacies of the applications. It was then recognized that DxTer would be an invaluable tool to use for optimizing such applications. As DxTer is the product of Dr. Bryan Marker, we could start a true collaboration to test the developed theory. After having explained the theory, Bryan encoded the knowledge into DxTer, thereby giving us a way to test the theory. Without Bryan's tireless efforts, I would not have been able to finish this work as soon as I have. Unfortunately, only the handful of people mentioned understood the theory and notation. It is thanks to Dr.Tamara G. Kolda's meticulous eye for detail that the notation and theory has been refined into what it is today. Because of these efforts,

I would like to extend my deepest thanks to Robert, Tze Meng, Devin, Bryan, and Tammy.

In addition to the efforts associated with this dissertation, I am fortunate to have had both Robert and Tammy as my mentors. Not only have they guided me throughout my career as a graduate student, including preparing for the future and exposing me to different aspects of being a researcher, but they have also helped me how to better convey my ideas through writing, which is no simple task (as anyone who knows me can attest to). It is only through their constant encouragement and pressure to do better that I have been able to achieve all that I have. Once again, thank you.

Finally, I cannot go without acknowledging everyone I consider family (you know who you are). In particular, I would like to thank my brother Philip Schatz and my wife Erin Ballou. Philip always managed to do something zany, providing me needed reprieve from the stresses of this work, while at the same time (okay different time) showing me how fortunate I am to have him as a brother. And Erin, she is my rock. Whenever things looked impossible, Erin was there to make sure I knew that the impossible could be overcome. I have asked more from her than I could ever ask anyone, and I hope to be able to at least come close to making good on those loans. I am glad that I can finally begin repaying her for everything she has done for me during this long process. From the bottom of my heart, thank you all.

Martin Daniel Schatz

The University of Texas at Austin
December 2015

Distributed Tensor Computations:
Formalizing Distributions, Redistributions, and Algorithm Derivation

Publication No. ____________________

Martin Daniel Schatz, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Robert A. van de Geijn
Co-Supervisor: Tamara G. Kolda

A goal of computer science is to develop practical methods to automate tasks that are otherwise too complex or tedious to perform manually. Complex tasks can include determining a practical algorithm and creating the associated implementation for a given problem specification. Goal-oriented programming can make this systematic. Therefore, we can rely on automated tools to create implementations by expressing the task of creating implementations in terms of goal-oriented programming. To do so, pertinent knowledge must be encoded which requires a notation and language to define relevant abstractions.

This dissertation focuses on distributed-memory parallel tensor computations arising from computational chemistry. Specifically, we focus on applications based on the tensor contraction operation of dense, non-symmetric tensors. Creating an efficient algorithm for a given problem specification in this domain is complex; creating an optimized implementation of a developed algorithm is even more complex, tedious, and error-prone. To this end, we encode pertinent knowledge for distributed-memory parallel algorithms for tensor contractions of dense non-symmetric tensors. We do this by developing a notation for data distribution and redistribution that exposes a systematic procedure for deriving a family of algorithms for this operation for which efficient implementations exist.

We validate the developed ideas by implementing them in the Redistribution Operations and Tensor Expressions application programming interface (ROTE API) and encoding them into an automated system, DxTer, for systematically generating efficient implementations from problem specifications. Experiments performed on the IBM Blue Gene/Q and Cray XC30 architectures testing generated implementations for the spin-adapted coupled cluster singles and doubles method from computational chemistry demonstrate impact both in terms of performance and storage requirements.

Contents

Glossary of Notation

Here, we provide tables summarizing the various notation used throughout this document.

	Acronym	Interpretation
	BLAS	Basic Linear Algebra Subprograms
	CTF	Cyclops Tensor Framework
	TCE	Tensor Contraction Engine
	ATLAS	Automatically Tuned Linear Algebra Software
	ROTE	Redistribution Operations and Tensor Expressions
	MPI	Message-Passing Interface
	SIAL	Super Instruction Assembly Language
	ACES	Advanced Concepts in Electronic Structure
	DLTC	Dynamically Load-balanced Tensor Contractions
	DxT	Design by Transformation
	DxTer	Tool that implements Design by Transformation
	CCSD	Coupled Cluster Singles and Doubles

	Notation	Interpretation
Set related	$\mathcal{A}, \mathcal{B}, \ldots$	(Ordered) Sets
	$a_u, b_u, \ldots$	Element at index u of ordered set $\mathcal{A}$, $\mathcal{B}$,...
	$\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \ldots$	(Ordered) Sets of (ordered) sets
	$\lvert\mathcal{A}\rvert$	The cardinality of $\mathcal{A}$
	$\mathcal{A} \otimes \mathcal{B}$	The Cartesian product of $\mathcal{A}$ and $\mathcal{B}$
	$\mathcal{A} \sqcup \mathcal{B}$	The concatenation of $\mathcal{B}$ to $\mathcal{A}$
Tensor related	$\mathbf{A}, \mathbf{B}, \ldots$	Data tensors
	$M, M^{(\mathbf{A})}, M^{(\mathbf{B})}, \ldots$	Tensor order (of tensor $\mathbf{A}, \mathbf{B}, \ldots$)
	$\mathbf{G}$	Processing mesh
	N	Processing mesh order
	$\mathbf{P}$	Size of $\mathbf{G}$
	P_m	Mode-m dimension of $\mathbf{G}$
	$\mathbf{p}, \mathbf{q}$	Processes in $\mathbf{G}$
	$\iota, \eta, \kappa, \ldots$	Tensor mode labels for contraction operation
	$\widehat{\sum}, \widehat{\sum}_{\iota\eta}$	Local summation (over modes labeled ι, η)
	$\widetilde{\sum}, \widetilde{\sum}_{\iota\eta}$	Global summation (over modes labeled ι, η)
Index related	$\mathbf{I}, \mathbf{I}^{(\mathbf{A})}, \mathbf{I}^{(\mathbf{B})}, \ldots$	Size array (of tensor $\mathbf{A}$, $\mathbf{B}$, ...)
	$I_m, I_m^{(\mathbf{A})}, I_m^{(\mathbf{B})}, \ldots$	Mode-m dimension (of tensor $\mathbf{A}$, $\mathbf{B}$, ...)
	$\mathbf{i}, \mathbf{i}^{(\mathbf{A})}, \mathbf{i}^{(\mathbf{B})}, \ldots$	Multiindex (of tensor $\mathbf{A}$, $\mathbf{B}$, ...)
	$i_m, i_m^{(\mathbf{A})}, i_m^{(\mathbf{B})}, \ldots$	Mode-m index in multiindex (of tensor $\mathbf{A}$, $\mathbf{B}$, ...)
	$\mathcal{R}\left(I\right)$	The range associated with a dimension I
	$\mathrm{prod}\left(\mathbf{I}\right)$	The cumulative product of size array $\mathbf{I}$
	$\mathrm{multi2linear}\left(\mathbf{i}, \mathbf{I}\right)$	The linear index of $\mathbf{i}$ in $\mathbf{I}$

	Notation	Interpretation
Distribution related	$\boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{D}}^{(\mathbf{A})}, \boldsymbol{\mathcal{D}}^{(\mathbf{B})}, \ldots$	Tensor distribution (of tensor **A**, **B**, ...)
	$\overline{\boldsymbol{\mathcal{D}}}, \overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}, \overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}, \ldots$	"Final" tensor distributions
	$\mathcal{D}, \mathcal{D}^{(0)}, \mathcal{D}^{(1)}, \ldots$	Tensor mode distribution (of mode 0, mode 1, ...)
	$\widetilde{\mathcal{D}}, \widetilde{\mathcal{D}}^{(0)}, \widetilde{\mathcal{D}}^{(1)}, \ldots$	Comm. mode distribution (of mode 0, mode 1, ...)
	$\overline{\mathcal{D}}, \overline{\mathcal{D}}^{(0)}, \overline{\mathcal{D}}^{(1)}, \ldots$	"Final" tensor mode distributions
	$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}\right)$	Elements assigned to process **p** under $\boldsymbol{\mathcal{D}}$.
	$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}\right)$	Mode-m indices assigned to process **p** under $\mathcal{D}$.

Chapter 1

Introduction

For dense linear algebra, it has already been shown that carefully structured abstractions support the development of implementations that achieve high performance on distributed-memory architectures [18, 65, 86]. Tensor contractions, the generalization of matrix-matrix multiplication to higher-dimensional objects, are inherently more difficult operations to optimize due to the number of algorithmic variants that can, and as we demonstrate, need to be considered, making a full optimization daunting, even to an expert.

The thesis of this work is that a notation can be defined for data distributions and redistributions that exposes a systematic procedure for deriving parallel algorithms with high-performance implementations for the tensor contraction operation of dense non-symmetric tensors. In doing so, pertinent domain knowledge is consolidated into a formal language that facilitates the automatic generation of algorithms with high-performance implementations for individual and, more importantly, a series of tensor contractions. These techniques advance the state of the art in computer science by providing the foundation for encoding pertinent knowledge in the domain of distributed-memory parallel tensor computations. Additionally, these techniques advance the state of the art in computational chemistry as implementations for the spin-adapted coupled cluster singles and doubles method (CCSD) are developed that require at most half the available memory for workspace and attain performance that improves upon state of the art.

1.1 Motivation and Goals

The goal of any computational method designed for a distributed-memory architecture is to effectively utilize the available processing elements (processes) to collectively perform a computation. For the domain of dense linear algebra, both theoretical and practical approaches have been developed that achieve high performance on distributed-memory architectures while simultaneously reducing the storage needed for workspace [2, 18, 34, 49, 65, 79, 86, 87]. For each of these, the relationship between the data distribution and the developed algorithm must be considered in conjunction. This is necessary not only to ensure that data is efficiently distributed and redistributed among processes, but also to ensure that local computations performed by each process are efficient [73].

Tensors are multidimensional arrays and can be considered generalizations of matrices. The order of a tensor is the number of ways or dimensions that it represents. We say a tensor is higher-order if its order is greater than two. Tensor contractions are a generalization of matrix-matrix multiplication and are, for example, at the heart of methods in computational chemistry [7, 62, 66]. As we discuss in the next section, the challenge in developing efficient implementations for tensor contractions stems from the increased number of data distributions and algorithmic variants available.

The simplest approach to computing a tensor contraction is to rearrange the data, express the computation in terms of a matrix-matrix multiplication, and leverage previous work on parallelizing that simpler operation. Unfortunately, in doing so, the multiway structure of the operation is inherently lost, thereby reducing the opportunities for improving performance and/or limiting workspace requirements. Instead, in this work, we exploit the similarities between tensor contractions and matrix-matrix multiplication, allowing us to extend ideas used for matrix-matrix multiplication [70, 72, 80, 81].

Many important computations in quantum chemistry can be expressed as a long series of contractions [47, 62, 66]. For example, Figure 1.1 depicts the set of equations that define the CCSD application used as a motivating example in this work. We discuss how to interpret the depicted equations in Section 2.2 and Section 5.1. For now it is only necessary to understand that every instance of a summation in Figure 1.1 corresponds to a separate tensor contraction involving dense non-symmetric tensors. As the goal is to optimize the entire computation, an expert has to consider how to optimize all contractions in conjunction with one another in terms of both communication and computation. This quickly becomes daunting. The ideal solution is to automate this process; however, for automation to be possible, a well-defined language that expresses pertinent information must be designed.

With a well-defined language that expresses the relationship between data distribution, data redistribution, and the structure of the parallel algorithms used for computation, we can reason about optimizing an entire series of contractions as all steps involved are some combination of data redistribution or local computation.

$$
\begin{aligned}
W_{je}^{bm} &= (2w_{je}^{bm} - x_{ej}^{bm}) + \sum_{f}(2r_{fe}^{bm} - r_{ef}^{bm})t_j^f - \sum_{n}(2u_{je}^{nm} - u_{je}^{mn})t_n^b \\
&\quad + \sum_{fn}(2v_{nm}^{fe} - v_{mn}^{fe})(T_{jn}^{bf} + \frac{1}{2}T_{nj}^{bf} - \tau_{nj}^{bf}) \\
X_{ej}^{bm} &= x_{ej}^{bm} + \sum_{f} r_{ef}^{bm} t_j^f - \sum_{n} u_{je}^{mn} t_n^b - \sum_{fn} v_{mn}^{fe}(\tau_{nj}^{bf} - \frac{1}{2}T_{nj}^{bf}) \\
U_{ie}^{mn} &= u_{ie}^{mn} + \sum_{f} v_{mn}^{fe} t_i^f \\
Q_{ij}^{mn} &= q_{ij}^{mn} + (1 + \mathcal{P}_{nj}^{mi}) \sum_{e} u_{ie}^{mn} t_j^e + \sum_{ef} v_{mn}^{ef} \tau_{ij}^{ef} \\
P_{mb}^{ji} &= u_{mb}^{ji} + \sum_{ef} r_{ef}^{bm} \tau_{ij}^{ef} + \sum_{e} w_{ie}^{bm} t_j^e + \sum_{e} x_{ej}^{bm} t_i^e \\
H_e^m &= \sum_{fn}(2v_{mn}^{ef} - v_{nm}^{ef})t_n^f \\
F_e^a &= -\sum_{m} H_e^m t_m^a + \sum_{fm}(2r_{ef}^{am} - r_{fe}^{am})t_m^f - \sum_{fmn}(2v_{mn}^{ef} - v_{nm}^{ef})T_{mn}^{af} \\
G_i^m &= \sum_{e} H_e^m t_i^e + \sum_{en}(2u_{ie}^{mn} - u_{ie}^{nm})t_n^e + \sum_{efn}(2v_{mn}^{ef} - v_{nm}^{ef})T_{in}^{ef} \\
z_i^a &= -\sum_{m} G_i^m t_m^a - \sum_{emn}(2U_{ie}^{mn} - U_{ie}^{nm})T_{mn}^{ae} + \sum_{em}(2w_{ie}^{am} - x_{ei}^{am})t_m^e \\
&\quad + \sum_{em}(2T_{im}^{ae} - T_{mi}^{ae})H_e^m + \sum_{efm}(2r_{ef}^{am} - r_{fe}^{am})\tau_{im}^{ef} \\
Z_{ij}^{ab} &= v_{ij}^{ab} + \sum_{mn} Q_{ij}^{mn} \tau_{mn}^{ab} + \sum_{ef} y_{ef}^{ab} \tau_{ij}^{ef} + (1 + \mathcal{P}_{bj}^{ai}) \left\{ \sum_{e} r_{ab}^{ej} t_i^e \right. \\
&\quad - \sum_{m} P_{mb}^{ij} t_m^a + \sum_{e} F_e^a T_{ij}^{eb} - \sum_{m} G_i^m T_{mj}^{ab} + \frac{1}{2} \sum_{em} W_{je}^{bm}(2T_{im}^{ae} - T_{mi}^{ae}) \\
&\quad \left. -(\frac{1}{2} + \mathcal{P}_j^i) \sum_{em} X_{ej}^{bm} T_{mi}^{ae} \right\}
\end{aligned}
$$

Figure 1.1: Equations for a single iteration of the spin-adapted CCSD method based on the formulation from Scuseria, Scheiner, Lee, Rice, and Schaefer [76]. Following the notation in [76], both superscripts and subscripts of each tensor are used to represent labels assigned to modes (under some order). This notation differs from what is used in this dissertation. Each summation indicates a contraction. Details of the computation and how it relates to the theory developed in this dissertation are given in Chapter 5.

By associating costs with each operation defined in the developed language, an automated system can make intelligent decisions about how applications should be implemented. In this dissertation, we advance the state of the art towards achieving this goal of automation.

As we discuss in related works (Chapter 6), recent advances that provide efficient implementations for applications based on tensor computations are typically limited to considering a single contraction at a time. However, many of these works support tensor contractions involving sparse tensors or dense tensors with internal structure such as symmetry. When restricted to applications based on contractions of dense, non-symmetric tensors, this work differs in that it can optimize across a series of contractions when designing the algorithms and associated implementations. This is perhaps the most important difference of our work to that of other approaches.

1.2 Solution

We develop a notation that encodes a class of distributions of tensor data on a multidimensional processing mesh as well as a set of redistributions that are directly associated with various collective communications. With this notation, we expose a systematic approach for deriving families of high-performance algorithms for an arbitrary series of tensor contraction operations, thereby significantly reducing the number of choices needed to be made by an expert (human or mechanical).

By encoding the structure of the algorithms and the necessary communications using the same notation, decisions that normally would be made by an error-prone human can be automated. With automated tools creating the implementations, one can trust the correct optimizations to be applied wherever appropriate.

1.3 Background

Here we briefly discuss relevant history of parallel matrix-matrix multiplication as well as the Design-by-Transformation approach to software engineering (DxT). This provides the necessary background for the ideas developed in this document.

1.3.1 Parallel Matrix-Matrix Multiplication

Distributed-memory parallel algorithms for matrix-matrix multiplication, $\mathbf{C} = \mathbf{AB}$, have a long and rich history. Cannon's algorithm [15], the earliest such algorithm, is based on a block distribution of elements and point-to-point communications that iteratively cycle blocks of both input matrices, $\mathbf{A}$ and $\mathbf{B}$, among processes, computing contributions to the result, $\mathbf{C}$, at every iteration. Cannon's algorithm is sometimes referred to as the "roll-roll multiply" algorithm as blocks of both inputs $\mathbf{A}$ and $\mathbf{B}$ are cycled during the computation. Later, Fox et al. [27] developed an algorithm, sometimes called Fox's algorithm, that relies on the broadcast collective to communicate one of the two input matrices and a point-to-point communication for the other. For this reason, Fox's algorithm is sometimes referred to as the "broadcast-roll multiply" algorithm. Both algorithms target square processing meshes and are difficult to generalize to non-square configurations [19, 38, 39].

In the 1990s, algorithms were developed that improved upon both Cannon's and Fox's algorithms. Agarwal et al. [3], an algorithm is developed based on the all-gather collective communication[1] using a cyclic distribution of data. The scalable universal matrix multiplication algorithm (SUMMA) [87], developed in 1997, casts redistributions of *both* input operands in terms of broadcast collectives. One impor-

[1] communication involving groups of processes

tant contribution of [87] is the development of related algorithms for the different transpose variants of matrix-matrix multiplication, developing a step towards a family of algorithms. Gunnels et al. [30], a family of algorithms based on the idea of holding one operand "stationary" is developed based on broadcast, allgather, scatter, reduce-to-one, and gather-to-one collectives for communication. Creating families of algorithms allows selection of an algorithm that is most suitable for the problem of interest.

Following these previous works based on two-dimensional processing meshes, it was discovered that better theoretical and practical results could be achieved for matrix-matrix multiplication on a three-dimensional processing mesh [2, 41, 79]. The systematic derivation of such algorithms is encoded into a notation [73] and implemented into the high-performance library Elemental [65], based on an elementwise cyclic distribution.

1.3.2 Design-by-Transformation for Matrix Computations

Design-by-Transformation (DxT) is an approach to software engineering that creates implementations from a problem specification within a domain by systematically applying a series of transformations, gradually transforming the abstract representation of the computation into a concrete implementation [53, 55, 57]. As typically many transformations can be applied at any given step, a large space of possible implementations is created by this process.

DxTer is a prototype system that implements the ideas of DxT for an encoded domain. It can intelligently search the created space of implementations for the optimal, based on costs associated with each implementation. The details of how

DxTer performs this search for a given domain are out of the scope of this dissertation, but are detailed in [53, 58].

DxTer was applied to the Elemental library [65] for distributed-memory dense linear algebra [55, 57]. After encoding the notation and knowledge developed for Elemental, DxTer was able to recreate the optimizations hand-implemented by an expert. DxTer is related to, but differs greatly in approach, other automated code-generation projects such as SPIRAL [67], Built-To-Order BLAS [10], LGEN [82], AUGEM [89], and ATLAS [90]. A key difference is that the theory underlying DxT and DxTer does not rely on empirical testing or heuristics to determine the optimal (with respect to the encoded knowledge) implementation among the space of implementations[2]. Further, DxT and DxTer are domain agnostic, whereas many of the mentioned projects are domain specific. In Section 5.3, we discuss DxT and DxTer greater detail.

1.4 Contributions

In this dissertation, our goal is to generalize many of the insights from the domain of distributed-memory matrix computations to the domain of distributed-memory tensor computations. We provide the following contributions to state-of-the-art tensor-contraction methods:

- A concise notation for data distributions of arbitrary-order tensors on arbitrary-order processing meshes that formalizes data movement in terms of redistributions that can be cast in terms of collective communications.

[2] We mention here that DxTer does employ a heuristic for search, but this heuristic is not a fundamental feature of the search algorithm employed [56].

- A systematic approach to deriving algorithms for a single or a series of tensor contraction operations.
- A formalization of select transformations that enable high performance implementations.
- Development of the Redistribution Operations and Tensor Expressions (ROTE) C++ library that encodes the methods introduced in this document.
- A demonstration that our notation combined with the methods in DxTer [54, 55] leads to efficient implementations that improve upon the state of the art in tensor contractions, in some cases improving performance by fifty percent or more, while requiring significantly less storage to perform the same computations.

1.5 Outline of the Dissertation

Chapter 2 develops the notation for representing tensor data distribution and redistribution. The developed notation describes data of an arbitrary-order tensor distributed on an arbitrary-order processing mesh via an elemental-cyclic distribution. *As observed from work in the domain of distributed-memory dense linear algebra, many algorithms with high-performance implementations for matrix computations share the structure of performing redistributions, followed by local computations, followed by a global reduction (if necessary).* As the redistributions are implemented with collective communications, we develop our notation to capture this structure and extend it to tensor computations.

In Chapter 3, we show how to derive algorithms for the tensor contraction operation.

The method shows how families of algorithms can be systematically derived from the same problem specification.

In Chapter 4, we identify and formalize specialized transformations that improve performance, demonstrating the notation's extensibility and expressiveness. We begin with two transformations that exploit inherent structure within a class of redistributions to reduce the overall cost. We then shift focus to a more practical extension of the defined notation that can be used to reduce the time spent in local computation by eliminating unnecessary data movement.

In Chapter 5, we present performance results of implementations based on the ideas in this document and generated by DxTer for the spin-adapted CCSD method from computational chemistry. We show that upwards of a fifty percent improvement in performance can be achieved while achieving a factor three reduction in storage over other state-of-the-art methods.

In Chapter 6, we discuss related work. A key difference of this work include that it enables the optimization of a series of tensor contractions together across the required communications and local computations.

In Chapter 7, we end with concluding remarks and ideas for future research.

Chapter 2

Notation

In this chapter we introduce a notation for formalizing the distribution of arbitrary-order tensors on arbitrary-order processing meshes. We then relate redistributions of data in the defined notation to efficient collective communications. For reference, a glossary of all notation used in this document is given in the beginning of this document.

2.1 Preliminaries

2.1.1 Tensors

A tensor is an M-dimensional, or M-mode, array. The *order* of a tensor refers to the number of dimensions (also called ways or modes) represented by the tensor. The term *dimension* refers the length, or size, of a specific mode. We use boldface capital letters to refer to tensors ($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$). The order of a data tensor is denoted M. If necessary, a parenthesized superscript is used to differentiate between the tensors being considered; e.g., $M^{(\mathbf{A})}$ refers to the order of the tensor $\mathbf{A}$.

We use $\mathbf{I} = (I_0, \ldots, I_{M-1})$ to refer to the size of an order-M tensor. When referencing an element of the order-M tensor $\mathbf{A}$, we specify its location in $\mathbf{A}$ with an M-tuple, or *multiindex*, $\mathbf{i} = (i_0, \ldots, i_{M-1})$ with entries corresponding to the element's index in each mode of the tensor. Again, sizes and multiindices of a specific tensor are distinguished by a parenthesized superscript if necessary.

Example 1 incorporates the previously defined notation and shows the cases where each notation is used.

Example 1. *Consider the order-*2 *tensor* $\mathbf{A} \in \mathbb{R}^{2\times 4}$*. The size of* $\mathbf{A}$ *is denoted by* $\mathbf{I} = (I_0, I_1) = (2, 4)$*. Let* $\mathbf{A}$ *be defined with the entries according to*

$$\mathbf{A} = \begin{pmatrix} 0.5 & 2.5 & 4.5 & 6.5 \\ 1.5 & 3.5 & 5.5 & 7.5 \end{pmatrix}.$$

The value of the element at location $\mathbf{i} = (i_0, i_1) = (1, 2)$ *is* 5.5*. Hence,* $a_{\mathbf{i}} = 5.5$*.*

2.1.2 Processing Mesh

We use $\mathbf{G}$ to refer to the order-N processing mesh. We use $\mathbf{P} = (P_0, \ldots, P_{N-1})$ to denote the size of $\mathbf{G}$ and $\mathbf{p} = (p_0, \ldots, p_{N-1})$ to refer to a specific process.

2.1.3 Ordered Sets

We use capital script letters $(\mathcal{A}, \mathcal{B}, \ldots)$ to refer to sets, and boldface capital script letters $(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \ldots)$ to refer to sets of sets. Ordered sets, or tuples, are sets with an explicit order of elements. The cardinality of the set $\mathcal{A}$ is denoted $|\mathcal{A}|$. We denote the Cartesian product of the sets $\mathcal{A}$ and $\mathcal{B}$ as $\mathcal{A} \otimes \mathcal{B}$. We use braces to denote a set without order, and parentheses to indicate a set with order. See Example 2.

Example 2. *To define the unordered set $\mathcal{S}$ containing the elements 4,5, and 7, we say*

$$\mathcal{S} = \{4, 5, 7\}.$$

We could equivalently say $\mathcal{S} = \{4, 7, 5\}$, etc. The cardinality of $\mathcal{S}$ is $|\mathcal{S}| = 3$.
To define the ordered set $\mathcal{S}$ containing the same elements in, for example, decreasing order, we say

$$\mathcal{S} = (7, 5, 4).$$

The ordered set of all valid multiindices is called the *range* and is denoted by $\mathcal{R}$, as defined in Definition 1.

Definition 1. *The set of indices associated with a dimension $I \in \mathbb{N}$ is denoted $\mathcal{R}(I)$ where*

$$\mathcal{R}(I) = \begin{cases} \emptyset & \textit{if } I = 0 \\ (0, 1, \ldots, I-1) & \textit{otherwise.} \end{cases}$$

It is convenient to treat this as an ordered set. The set of all multiindices associated with a size array $\mathbf{I} = (I_0, \ldots, I_{M-1})$ is denoted $\mathcal{R}(\mathbf{I})$ where

$$\mathcal{R}(\mathbf{I}) = \mathcal{R}(I_0) \otimes \cdots \otimes \mathcal{R}(I_{M-1})$$

and $\mathcal{A} \otimes \mathcal{B}$ represents the Cartesian product of the sets $\mathcal{A}$ and $\mathcal{B}$.

When discussing redistributions and how to derive algorithms, it is useful to view ordered sets in terms of a prefix and suffix of elements. Definition 2 defines the concatenation operation on ordered sets; see also Example 3.

Definition 2. *The concatenation of two ordered sets is denoted with $\sqcup$; i.e.,*

$$\mathcal{A} \sqcup \mathcal{B} = \left(a_0, \ldots, a_{|\mathcal{A}|-1}, b_0, \ldots, b_{|\mathcal{B}|-1}\right).$$

Example 3. *Consider the ordered sets $\mathcal{A} = (3, 6, 2)$ and $\mathcal{B} = (4, 8)$. The elements of $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$ are*

$$\mathcal{C} = (c_0, c_1, c_2, c_3, c_4) = (a_0, a_1, a_2, b_0, b_1) = (3, 6, 2, 4, 8).$$

In order to select or filter entries from an ordered set, we introduce subset selection in Definition 3.

Definition 3. *Given an ordered set* $\mathcal{S} = \left(s_0, \ldots, s_{|\mathcal{S}|-1}\right)$ *and an ordered set* $\mathcal{F} = \left(f_0, \ldots, f_{|\mathcal{F}|-1}\right) \subseteq \mathcal{R}\left(|\mathcal{S}|\right)$ *specifying the entries of* $\mathcal{S}$ *to select, we denote the tuple with entries of* $\mathcal{S}$ *in the order specified by* $\mathcal{F}$ *as*

$$\mathcal{S}\left(\mathcal{F}\right) = \left(s_{f_0}, s_{f_1}, \ldots, s_{f_{|\mathcal{F}|-1}}\right).$$

Example 4. *Given an ordered set* $\mathcal{S} = (1, 5, 3, 6, -1)$ *and an ordered set* $\mathcal{F} = (2, 0, 4)$ *specifying the entries of* $\mathcal{S}$ *to select, then*

$$\mathcal{S}\left(\mathcal{F}\right) = \left(s_{f_0}, s_{f_1}, s_{f_2}\right) = \left(s_2, s_0, s_4\right) = (3, 1, -1).$$

2.1.4 Index Conversion

Describing and analyzing tensor distributions requires the mapping from a multiindex to the corresponding linear index. To give an idea of what this function represents, consider a matrix whose elements are stored somewhere in memory (linear array of addresses). In this case, mapping from a multiindex to a linear index corresponds to taking an element's location in the matrix and determining its offset in memory. This function depends on whether the matrix is stored via column- or row-major order. In this section, we generalize this notion to support the arbitrary mode ordering for storing a higher-order tensor.

Notation for the cumulative product is given in Definition 4.

Definition 4. *Given a size array* $\mathbf{I}$ *of an order-M tensor, the function* $\operatorname{prod}(\mathbf{I})$ *computes the cumulative product, i.e.,*

$$\operatorname{prod}(\mathbf{I}) = \begin{cases} 1 & \text{if } M = 0 \\ \displaystyle\prod_{\ell \in \mathcal{R}(M)} I_\ell & \text{otherwise.} \end{cases}$$

To specify the cumulative product considering only the entries of $\mathbf{I}$ *at locations specified by the set* $\mathcal{S}$*, we say*

$$\operatorname{prod}(\mathbf{I}, \mathcal{S}) = \operatorname{prod}(\mathbf{I}(\mathcal{S}))\,.$$

Definition 5 gives the function to convert multiindices to linear indices.

Definition 5. *Given an order-M multiindex $\mathbf{i}$ and corresponding size array $\mathbf{I}$, the function* $\operatorname{multi2linear}(\mathbf{i}, \mathbf{I})$ *converts the multiindex $\mathbf{i}$ to the corresponding linear index, i.e.,*

$$\operatorname{multi2linear}(\mathbf{i}, \mathbf{I}) = \begin{cases} 0 & \text{if } M = 0 \\ \displaystyle\sum_{\ell \in \mathcal{R}(M)} i_\ell \cdot \operatorname{prod}(\mathbf{I}, \mathcal{R}(\ell)) & \text{otherwise.} \end{cases}$$

To specify the linear index of $\mathbf{i}$ considering the entries of $\mathbf{i}$ at locations specified by the ordered set $\mathcal{S}$, we say

$$\operatorname{multi2linear}(\mathbf{i}, \mathbf{I}, \mathcal{S}) = \operatorname{multi2linear}(\mathbf{i}(\mathcal{S}), \mathbf{I}(\mathcal{S})).$$

Example 5 gives an example of using the multi2linear function convert multiindices of a matrix to their corresponding linear index.

Example 5. *Determine the linear index corresponding to the multiindex* $\mathbf{i} = (2,4)$ *of a matrix of size* $\mathbf{I} = (3,6)$ *based on a column-major and row-major ordering of elements.*

For a column-major ordering of elements, the associated filter is $\mathcal{F} = (0,1)$*.*

$$\begin{aligned}
\text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{F}\right) &= \text{multi2linear}\left(\mathbf{i}\left(\mathcal{F}\right), \mathbf{I}\left(\mathcal{F}\right)\right) \\
&= \text{multi2linear}\left(\left(2,4\right), \left(3,6\right)\right) \\
&= 2 + 4 \cdot (3) = 14.
\end{aligned}$$

For a row-major ordering of elements, the associated filter is $\mathcal{F} = (1,0)$*.*

$$\begin{aligned}
\text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{F}\right) &= \text{multi2linear}\left(\mathbf{i}\left(\mathcal{F}\right), \mathbf{I}\left(\mathcal{F}\right)\right) \\
&= \text{multi2linear}\left(\left(4,2\right), \left(6,3\right)\right) \\
&= 4 + 2 \cdot (6) = 16.
\end{aligned}$$

Example 6 gives an example of using the multi2linear function to convert multi-indices of a tensor to a linear index.

Example 6. *Determine the linear index corresponding to the multiindex* $\mathbf{i} = (2,4,3,5)$ *of a tensor of size* $\mathbf{I} = (3,6,4,7)$*, considering only the modes in the order specified by* $\mathcal{F} = (1,2,0)$*.*

$$\begin{aligned}
\text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{F}\right) &= \text{multi2linear}\left(\mathbf{i}\left(\mathcal{F}\right), \mathbf{I}\left(\mathcal{F}\right)\right) \\
&= \text{multi2linear}\left(\left(4,3,2\right), \left(6,4,3\right)\right) \\
&= 4 + 3 \cdot (6) + 2 \cdot (6 \cdot 4) = 70.
\end{aligned}$$

2.2 The Tensor Contraction Operation

We now introduce the binary tensor contraction operation by relating it to matrix-matrix multiplication and the related unary tensor contraction operation.

2.2.1 Binary Tensor Contraction

The binary tensor contraction operation generalizes matrix-matrix multiplication by allowing each operand to represent an arbitrary number of modes. It is related to matrix-matrix multiplication as it is an operation that performs a sum over products. As with matrix multiplication, modes of the different operands are *paired* and, depending on how the pairing is specified, the associated computation is defined differently. For instance, consider the matrices $\mathbf{C} \in \mathbb{R}^{I_0 \times I_1}$, $\mathbf{A} \in \mathbb{R}^{I_0 \times I_2}$, $\mathbf{B} \in \mathbb{R}^{I_1 \times I_2}$, and the matrix-matrix multiplication

$$\mathbf{C} = \mathbf{A}\mathbf{B}^{\mathsf{T}} \tag{2.1}$$

defined elementwise as

$$c_{i_0,i_1} = \sum_{\ell \in \mathcal{R}(I_2)} a_{i_0,\ell} b_{i_1,\ell}. \tag{2.2}$$

The binary tensor contraction operation generalizes these ideas to support operands representing arbitrary numbers of modes.

To indicate the modes that are to be paired, we assign each mode of each object a particular label; modes labeled similarly are paired. For example, in our tensor notation, we write (2.1) as

$$\mathbf{C}^{\iota\eta} = \sum_{\kappa} \mathbf{A}^{\iota\kappa}\mathbf{B}^{\eta\kappa} \tag{2.3}$$

where the labels ι,η, and κ denote how the corresponding mode of each tensor is paired. The set of labels written below the summation indicates summation over the modes labeled κ. This notation is similar to Einstein notation [26]. In Einstein notation, a distinction between covariant vectors and contravariant vectors is made. In this document, we omit the difference and simply interpret similarly labeled modes to be paired and paired modes of input tensors to be involved in the summation. One can directly convert (2.3) to the elementwise definition as given in (2.2).

For a more complex example, consider the binary tensor contraction of an order-4 tensor and order-3 tensor to produce an order-3 tensor:

$$\mathbf{C}^{\iota\eta\kappa} = \sum_{\nu\mu} \mathbf{A}^{\iota\mu\kappa\nu}\mathbf{B}^{\eta\nu\mu}.$$

Based on the definition, we see that mode 1 of **A** is paired to mode 2 of **B**, mode 3 of **A** is paired with mode 1 of **B**, and these modes are involved in the summation as these labels pair modes of **A** and **B** and are subscripts of the summation.

Extending this example, the definition of arbitrary binary tensor contractions can be deduced based on how the modes of each operand are paired together. We do not provide a formal definition of this idea as it unnecessarily complicates the description of the operation.

2.2.2 Unary Tensor Contractions

As the *unary tensor contraction* only has one input operand, no multiplication between operands can occur; however, the interpretation of how to accumulate contributions remains the same as in the binary tensor contraction. This operation is related to the matrix operation which accumulates columns (or rows) of the matrix together. For instance, consider the unary tensor contraction defined as

$$\mathbf{C}^{\iota} = \sum_{\eta} \mathbf{A}^{\iota\eta}.$$

Based on the definition, we see that we are accumulating entries within mode 1 of $\mathbf{A}$ together. This creates a vector (one-mode) tensor $\mathbf{C}$ such that the element c_{i_0} is the sum of the corresponding row of $\mathbf{A}$.

2.3 Data Distributions

In the context of high-performance distributed-memory dense linear algebra, *Cartesian* distributions have commonly been used in high-performance libraries [18, 65, 65, 70, 81, 86]. Examples of Cartesian distributions include blocked, where each process is assigned a contiguous block of indices; block-cyclic, where each process is assigned blocks of indices in a round-robin fashion; and elemental-cyclic, where each process is assigned indices in a round-robin fashion. Illustrations of blocked, block-cyclic, and elemental-cyclic distributions are given in Figure 2.1. Notice that an elemental-cyclic distribution (Figure 2.2c) is equivalent to a block-cyclic distribution (Figure 2.2b) with block-size $b = 1$.

$\mathbf{p} = (0)$	$\mathbf{p} = (1)$	$\mathbf{p} = (2)$
$a_{0:n-1}$	$a_{n:2n-1}$	$a_{2n:I_0-1}$

(a) Blocked distribution with $n = \left\lceil \frac{I_0}{3} \right\rceil$

$\mathbf{p} = (0)$	$\mathbf{p} = (1)$	$\mathbf{p} = (2)$
$a_{0:b-1}, a_{3b:4b-1}, \ldots$	$a_{b:2b-1}, a_{4b:5b-1}, \ldots$	$a_{2b:3b-1}, a_{5b:6b-1}, \ldots$

(b) Block-cyclic distribution with block-size $(b-1) \in \mathcal{R}(I_0)$

$\mathbf{p} = (0)$	$\mathbf{p} = (1)$	$\mathbf{p} = (2)$
$a_0, a_3, \ldots$	$a_1, a_4, \ldots$	$a_2, a_5, \ldots$

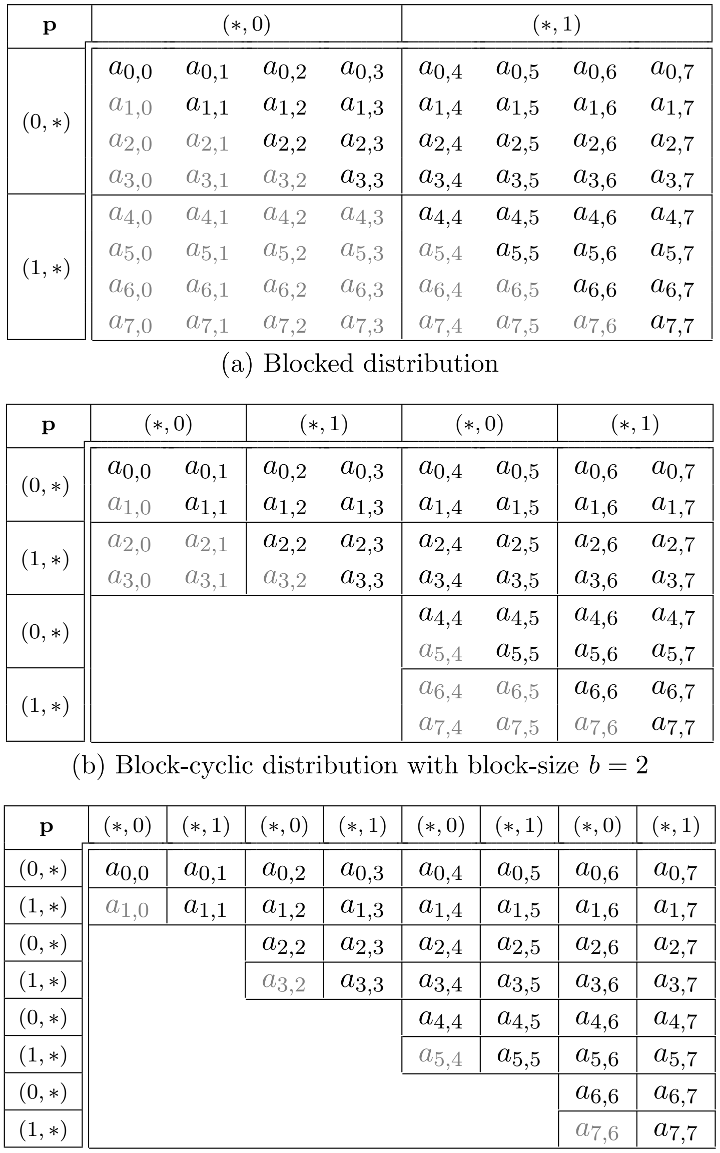
(c) Elemental-cyclic distribution

Figure 2.1: Graphical depiction of the vector $\mathbf{A}$ of dimension I_0 distributed on a linear processing mesh of size $\mathbf{P} = (3)$ processing mesh according to different Cartesian distributions.

Of these three distributions, elemental-cyclic distributions exhibit significantly less load imbalance among processes in numerous computing environments [69, 73, 80, 81], most notably when the objects being distributed exhibit forms of symmetry and we only distribute the unique entries (to reduce storage). For this reason, we focus on generalizing elemental-cyclic distributions of matrices to tensors on arbitrary-order processing meshes. In Figure 2.2, we illustrate the amount of load-imbalance introduced when distributing symmetric matrices under different cartesian distributions.

2.3.1 Elemental-cyclic Distributions for 1-D Data

An elemental-cyclic distribution is a Cartesian distribution where elements are assigned in a round-robin fashion to each process in $\mathbf{G}$. For example, consider the case where we distribute an order-1 tensor $\mathbf{A}$ of size I_0 on an order-1 processing mesh



p	$(*,0)$				$(*,1)$			
$(0,*)$	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$(1,*)$	$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
	$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
	$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
	$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

(a) Blocked distribution

p	$(*,0)$		$(*,1)$		$(*,0)$		$(*,1)$	
$(0,*)$	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$(1,*)$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$(0,*)$					$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
					$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$(1,*)$					$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
					$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

(b) Block-cyclic distribution with block-size $b = 2$

p	$(*,0)$	$(*,1)$	$(*,0)$	$(*,1)$	$(*,0)$	$(*,1)$	$(*,0)$	$(*,1)$
$(0,*)$	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$(1,*)$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$(0,*)$			$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$(1,*)$			$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$(0,*)$					$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$(1,*)$					$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$(0,*)$							$a_{6,6}$	$a_{6,7}$
$(1,*)$							$a_{7,6}$	$a_{7,7}$

(c) Elemental-cyclic distribution (block-cyclic with $b = 1$)

Figure 2.2: Illustration of a symmetric matrix **A** distributed on a size $\mathbf{P} = (2, 2)$ processing mesh (distributing only the unique entries) according to different Cartesian distributions. Extra entries required to maintain load-balance among processes are indicated in gray and empty regions of the matrix indicate entries that are not stored by any process. This figure was adapted from [81].

Assigned rank	Assigned indices
0	$\{0, P_0, 2P_0, \ldots\}$
1	$\{1, 1 + P_0, 1 + 2P_0, \ldots\}$
$\vdots$	$\vdots$

Table 2.1: Assignment of order-1 tensor indices to processes of order-1 mesh

comprised of P_0 processes based on an elemental-cyclic distribution. Assume that each process is assigned an identifier, referred to as the *rank*, in $\mathcal{R}(P_0)$. In general, the rank of a process can be any integer greater than or equal to zero; however, in this document the rank of a process is related to a linearization of the process's location in $\mathbf{G}$. The specific linearization chosen depends on the data distribution used.

In the most natural case of an elemental-cyclic distribution, processes are assigned elements at locations according to Table 2.1. For a more complex example, we can introduce an arbitrary offset σ and perform the same assignment of elements. Throughout this document, we assume an offset $\sigma = 0$ without loss of generality. Based on this description, we see that an elemental-cyclic distribution is parameterized by two values: the process *rank* and the *stride* between elements. Notice that the stride parameter is equal to the number of unique ranks we assign to processes. We may assign the same rank to multiple processes. We can also represent the distribution that replicates entries of $\mathbf{A}$ on all processes by assigning each process a rank of zero (stride parameter is one).

Now consider the case where we distribute elements of the order-1 tensor $\mathbf{A}$ among processes of an order-2 processing mesh. Illustrations of some distributions on order-2 meshes are depicted in Figure 2.3. We can assign each process a rank based on a column- or row-major ordering of processes to distribute data such that no replication occurs. This is similar to the distribution where each process of an order-1 mesh was assigned a rank according to its location in **G**. These distributions are sometimes refered to as "vector" distributions [25, 86]. Illustrations of these distributions are depicted in Figure 2.3b and Figure 2.3c. By assigning each process a rank of zero, we again describe a distribution where all elements are replicated among all processes.

In addition to these distributions, by assigning each process a rank equal to only its location in mode 0 of **G**, we can describe a data distribution where elements are distributed along mode 0 of **G** and replicated along mode 1 of **G**. We can similarly represent a distribution that replicates entries in mode 0 (instead of mode 1) by assigning each process a rank based only on its location in mode 1. These distributions are sometimes refered to as "matrix" distributions when applied to matrices [25, 86]. An illustration of the distribution based on a process's location in mode 0 is depicted in Figure 2.3a.

Connections between these vector and matrix distributions were made [11, 12, 25, 51, 86] revealing the set of matrix distributions "induced" by vector distributions. When applied to matrices, sometimes they are refered to as "Physically Based Matrix Distributions" because the distributions are directly related to where computation need to be performed. These ideas were later formalized and shown to be effective for describing a family of high-performance matrix-matrix multiplication algorithms [73].

It is due to the success of these distributions in the domain of matrix computations that we aim to generalize and formalize these distributions for tensors on arbitrary-order processing meshes. As we increase the order of the processing mesh, we increase the number of possible distributions. The notation developed in this thesis

$\mathbf{p} = (0,0)$ $r = 0$ $a_0, a_2, \ldots$	$\mathbf{p} = (0,1)$ $r = 0$ $a_0, a_2, \ldots$	$\mathbf{p} = (0,2)$ $r = 0$ $a_0, a_2, \ldots$
$\mathbf{p} = (1,0)$ $r = 1$ $a_1, a_3, \ldots$	$\mathbf{p} = (1,1)$ $r = 1$ $a_1, a_3, \ldots$	$\mathbf{p} = (1,2)$ $r = 1$ $a_1, a_3, \ldots$

(a) Mode-0 ordering (stride of two)

$\mathbf{p} = (0,0)$ $r = 0$ $a_0, a_6, \ldots$	$\mathbf{p} = (0,1)$ $r = 2$ $a_2, a_8, \ldots$	$\mathbf{p} = (0,2)$ $r = 4$ $a_4, a_{10}, \ldots$
$\mathbf{p} = (1,0)$ $r = 1$ $a_1, a_7, \ldots$	$\mathbf{p} = (1,1)$ $r = 3$ $a_3, a_9, \ldots$	$\mathbf{p} = (1,2)$ $r = 5$ $a_5, a_{11}, \ldots$

(b) Mode-(0,1) (column-major) ordering (stride of six)

$\mathbf{p} = (0,0)$ $r = 0$ $a_0, a_6, \ldots$	$\mathbf{p} = (0,1)$ $r = 1$ $a_1, a_7, \ldots$	$\mathbf{p} = (0,2)$ $r = 2$ $a_2, a_8, \ldots$
$\mathbf{p} = (1,0)$ $r = 3$ $a_3, a_9, \ldots$	$\mathbf{p} = (1,1)$ $r = 4$ $a_4, a_{10}, \ldots$	$\mathbf{p} = (1,2)$ $r = 5$ $a_5, a_{11}, \ldots$

(c) Mode-(1,0) (row-major) ordering (stride of six)

Figure 2.3: Illustration of the vector **A** distributed in an elemental-cyclic fashion on a processing mesh of size $\mathbf{P} = (2, 3)$. The rank of each process is denoted by r.

is based on this idea of representing distributions in terms of the ordered set of modes used to assign ranks to processes. We formalize these ideas in the following subsections.

2.3.2 Tensor Mode and Tensor Distributions

To define elemental-cyclic distributions for data of an order-M tensor $\mathbf{A}$, we need to specify the set of multiindices assigned to each process. We do this by assigning an elemental-cyclic distribution to each mode of the tensor; in other words, we separately distribute the indices of each tensor mode. We refer to the distribution of indices of a single mode as a *tensor mode distribution* and recognize that it is nothing more than a reinterpretation of elemental-cyclic distributions on order-1 processing meshes.

The combination of the indices in each mode specifies the multiindices of elements assigned to each process. We refer to the multiindices assigned to each process as a *tensor distribution*. We define both tensor mode and tensor distributions in Definition 6 and provide an example of a tensor distribution in Example 7.

Definition 6. *Consider a tensor* $\mathbf{A}$ *of size* $\mathbf{I}$*, and an order-*N *processing grid* $\mathbf{G}$ *of size* $\mathbf{P}$*. Given an ordered set* $\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right)$ *such that* $\mathcal{D}^{(m)} \in \mathcal{R}\left(N\right)$ *and each* $\mathcal{D}^{(m)}$ *is disjoint from all others, we say mode* m *is distributed as* $\mathcal{D}^{(m)}$ *if every process* $\mathbf{p} \in \mathcal{R}\left(\mathbf{P}\right)$ *is assigned the mode-*m *indices given by*

$$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right) = \left\{h \in \mathcal{R}\left(I_m\right) | h \equiv u \pmod{v}\right\},$$

where $u = \text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)}\right)$ *and* $v = \text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)}\right)$.
We say the $\mathbf{A}$ *has been distributed as* $\boldsymbol{\mathcal{D}}$ *if each process process* $\mathbf{p} \in \mathbf{P}$ *is assigned the set of elements at multiindices in*

$$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}\right) = \mathcal{I}_0^{(\mathbf{p})}\left(\mathcal{D}^{(0)}\right) \otimes \cdots \otimes \mathcal{I}_{M-1}^{(\mathbf{p})}\left(\mathcal{D}^{(M-1)}\right).$$

In effect, we are assigning processes the elements of $\mathbf{A}$ from a partition of all elements in $\mathbf{A}$. The idea of defining data distributions in terms of a partition on the elements has been widely used in other libraries for linear algebra [11, 12, 25, 51, 73, 86]. In these works, strong connections were made between different distributions and collective communications acting on the distributions. Due to the success of these ideas in the domain of distributed-memory parallel linear algebra, we generalize the concepts to support tensors.

Example 7. *Consider the order-*2 *tensor* $\mathbf{A}$ *of size* $\mathbf{I} = (8,3)$ *distributed as* $\boldsymbol{\mathcal{D}} = [(0,2), (1)]$ *on the order-*3 *processing grid* $\mathbf{G}$ *of size* $\mathbf{P} = (2,3,2)$*. Let us determine the set of elements assigned to each process* $\mathbf{p} \in \mathcal{R}(\mathbf{P})$*.*

First, determine what elements are distributed to the process at location $\mathbf{p} = (1,2,0)$*. The set of elements of* $\mathbf{A}$ *assigned to this process are those at locations defined by*

$$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}([(0,2),(1)]) = \mathcal{I}_0^{(\mathbf{p})}((0,2)) \otimes \mathcal{I}_1^{(\mathbf{p})}((1)) = \{(1,2),(5,2)\}$$

since

$$\begin{aligned}\mathcal{I}_0^{(\mathbf{p})}((0,2)) &= \{h \in \mathcal{R}(8) | h \equiv 1 \, (\mathrm{mod} \; \mathrm{prod}((2,3,2),(0,2)))\} \\ &= \{h \in (0,\ldots,7) | h \equiv 1 \, (\mathrm{mod} \; 4)\}\end{aligned}$$

and

$$\begin{aligned}\mathcal{I}_1^{(\mathbf{p})}((1)) &= \{h \in \mathcal{R}(3) | h \equiv 2 \, (\mathrm{mod} \; \mathrm{prod}((2,3,2),(1)))\} \\ &= \{h \in (0,\ldots,2) | h \equiv 2 \, (\mathrm{mod} \; 3)\}.\end{aligned}$$

Repeating this analysis, we see elements are assigned to each process in $\mathbf{G}$ *according to*

$\mathbf{p}$	$\mathcal{I}_0^{(\mathbf{p})}((0,2))$	$\mathcal{I}_1^{(\mathbf{p})}((1))$	$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}(\boldsymbol{\mathcal{D}})$
$(0,0,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 0 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 0 \, (\mathrm{mod} \; 3)\}$	$\{(0,0),(4,0)\}$
$(1,0,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 1 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 0 \, (\mathrm{mod} \; 3)\}$	$\{(1,0),(5,0)\}$
$(0,1,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 0 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 1 \, (\mathrm{mod} \; 3)\}$	$\{(0,1),(4,1)\}$
$(1,1,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 1 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 1 \, (\mathrm{mod} \; 3)\}$	$\{(1,1),(5,1)\}$
$(0,2,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 0 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 2 \, (\mathrm{mod} \; 3)\}$	$\{(0,2),(4,2)\}$
$(1,2,0)$	$\{h \in \mathcal{R}(8) \| h \equiv 1 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 2 \, (\mathrm{mod} \; 3)\}$	$\{(1,2),(5,2)\}$
$(0,0,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 2 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 0 \, (\mathrm{mod} \; 3)\}$	$\{(2,0),(6,0)\}$
$(1,0,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 3 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 0 \, (\mathrm{mod} \; 3)\}$	$\{(3,0),(7,0)\}$
$(0,1,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 2 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 1 \, (\mathrm{mod} \; 3)\}$	$\{(2,1),(6,1)\}$
$(1,1,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 3 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 1 \, (\mathrm{mod} \; 3)\}$	$\{(3,1),(7,1)\}$
$(0,2,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 2 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 2 \, (\mathrm{mod} \; 3)\}$	$\{(2,2),(6,2)\}$
$(1,2,1)$	$\{h \in \mathcal{R}(8) \| h \equiv 3 \, (\mathrm{mod} \; 4)\}$	$\{h \in \mathcal{R}(3) \| h \equiv 2 \, (\mathrm{mod} \; 3)\}$	$\{(3,2),(7,2)\}$

Throughout this document we denote the tensor distribution with boldface capital script $\boldsymbol{\mathcal{D}}$. As a shorthand, to represent the tensor $\mathbf{A}$ distributed as $\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right)$, we write $\mathbf{A}\left[\boldsymbol{\mathcal{D}}\right]$ or $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right]$ (square brackets used only for clarity).

Observe that if a tensor mode distribution is empty, then each process is assigned the full range of tensor mode indices (by definition of multi2linear and prod). Also notice that data is replicated over all processing mesh modes that are not used in the tensor distribution. In other words, data is replicated over mode n of $\mathbf{G}$ if

$$n \notin \bigcup_{m \in \mathcal{R}(M)} \mathcal{D}^{(m)}.$$

2.3.3 Advanced Tensor Distributions

At this point we have defined a set of tensor distributions that assign elements of $\mathbf{A}$ to each process in $\mathbf{G}$. Under the current interpretation of tensor distributions, if a processing mesh mode is excluded from the tensor distribution, the elements of $\mathbf{A}$ are replicated among processes within this excluded mode of $\mathbf{G}$. However, as we see later on, in certain situations it is useful to be able to represent cases where the data of $\mathbf{A}$ is assigned to only one process in this excluded mode of $\mathbf{G}$ (all other processes assigned no data). These distributions arise as the result of "-to-one" collectives such as gather-to-one and reduce-to-one.

For example, consider the case where $\mathbf{G}$ is of size $\mathbf{P} = (3, 4, 2)$ and we desire to distribute $\mathbf{A}$ according to $\mathbf{A}\left[(0), (1)\right]$ but only assign elements to those processes whose location in mode 2 is zero.

To support these kinds of distributions, we augment our notation with the set of modes over which we should *not* implicitly replicate[1]. The notation used for these distributions is defined in Definition 7.

Definition 7. *Consider the order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ to be distributed over processes of the order-N processing mesh $\mathbf{G}$ of size $\mathbf{P}$ according to the tensor distribution $\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right)$. Let $\mathcal{E}$ be the set of modes over which replication should not implicitly occur and let $w \in \mathcal{R}\left(\text{prod}\left(\mathbf{P}, \mathcal{E}\right)\right)$.*

We say $\mathbf{A}$ has been distributed as $\mathbf{A}\left[\boldsymbol{\mathcal{D}}; \mathcal{E}, w\right]$ if all processes in $\mathbf{p} \in \mathcal{R}\left(\mathbf{P}\right)$ are assigned elements of $\mathbf{A}$ according to

$$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}; \mathcal{E}, w\right) = \begin{cases} \boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}\right) & \textit{if } \text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{E}\right) = w \\ \emptyset & \textit{otherwise.} \end{cases}$$

We interprete the omission of $\mathcal{E}$ and w from $\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}; \mathcal{E}, w\right)$ to indicate $\mathcal{E} = \emptyset$ and $w = 0$.

For conciseness, replication over modes omitted from tensor distribution is assumed unless explicitly stated with the notation introduced in this subsection. In Figure 2.4, we show examples of different tensor distributions for matrices distributed on a rectangular processing mesh.

[1] It may seem more natural to explicitly state the modes over which replication occurs, however for historical reasons we choose to indicate the modes over which replication does *not* occur.

$\mathbf{p} = (0,0)$ $\begin{matrix} a_{0,0} & a_{0,3} & \cdots \\ a_{2,0} & a_{2,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (0,1)$ $\begin{matrix} a_{0,1} & a_{0,4} & \cdots \\ a_{2,1} & a_{2,4} & \cdots \\ \vdots & \vdots & \vdots \end{matrix}$	$\mathbf{p} = (0,2)$ $\begin{matrix} a_{0,2} & a_{0,5} & \cdots \\ a_{2,2} & a_{2,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
$\mathbf{p} = (1,0)$ $\begin{matrix} a_{1,0} & a_{1,3} & \cdots \\ a_{3,0} & a_{3,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (1,1)$ $\begin{matrix} a_{1,1} & a_{1,4} & \cdots \\ a_{3,1} & a_{3,4} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (1,2)$ $\begin{matrix} a_{1,2} & a_{1,5} & \cdots \\ a_{3,2} & a_{3,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$

(a) $\mathbf{A}\left[(0), (1)\right]$

$\mathbf{p} = (0,0)$ $\begin{matrix} a_{0,0} & a_{0,6} & \cdots \\ a_{1,0} & a_{1,6} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (0,1)$ $\begin{matrix} a_{0,1} & a_{0,7} & \cdots \\ a_{1,1} & a_{1,7} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (0,2)$ $\begin{matrix} a_{0,2} & a_{0,8} & \cdots \\ a_{1,2} & a_{1,8} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
$\mathbf{p} = (1,0)$ $\begin{matrix} a_{0,3} & a_{0,9} & \cdots \\ a_{1,3} & a_{1,9} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (1,1)$ $\begin{matrix} a_{1,4} & a_{0,10} & \cdots \\ a_{1,4} & a_{1,10} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (1,2)$ $\begin{matrix} a_{0,5} & a_{0,11} & \cdots \\ a_{1,5} & a_{1,11} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$

(b) $\mathbf{A}\left[(), (1,0)\right]$

$\mathbf{p} = (0,0)$ $\begin{matrix} a_{0,0} & a_{0,1} & \cdots \\ a_{2,0} & a_{2,1} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (0,1)$	$\mathbf{p} = (0,2)$
$\mathbf{p} = (1,0)$ $\begin{matrix} a_{1,0} & a_{1,1} & \cdots \\ a_{3,0} & a_{3,1} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p} = (1,1)$	$\mathbf{p} = (1,2)$

(c) $\mathbf{A}\left[(0), (); (1), 0\right]$

Figure 2.4: Graphical depiction of the matrix **A** distributed according to different tensor distributions represented in the defined notation. The top-left entry of every container corresponds to the process's location within the mesh.

Elemental	Indices of I_0 assigned to process (p_0, p_1)	Notation
$*$	$\{h \in \mathcal{R}(I_0) \mid h \equiv 0 \pmod{1}\}$	$()$
M_C	$\{h \in \mathcal{R}(I_0) \mid h \equiv p_0 \pmod{P_0}\}$	(0)
M_R	$\{h \in \mathcal{R}(I_0) \mid h \equiv p_1 \pmod{P_1}\}$	(1)
V_C	$\{h \in \mathcal{R}(I_0) \mid h \equiv p_0 + p_1 P_0 \pmod{P_0 P_1}\}$	$(0, 1)$
V_R	$\{h \in \mathcal{R}(I_0) \mid h \equiv p_1 + p_0 P_1 \pmod{P_0 P_1}\}$	$(1, 0)$

Figure 2.5: Distributions symbols from Elemental in terms of tensor mode distributions defined in this work. Here, we assume a processing mesh of size $\mathbf{P} = (P_0, P_1)$.

2.3.4 Tensor Distribution Constraints

For a valid tensor distribution, every element of **A** must be mapped to at least one process in **G**. The conditions of Definition 6 ensure that every element of **A** is assigned to *some* process.

For those familiar with the Elemental library (Elemental) [65], the set of distributions defined in this work express all distributions defined by Elemental (considering only matrices distributed on order-two processing meshes). Figure 2.5 relates the redistribution rules used in Elemental to those defined in this work.

2.4 Collective Communications

Collective communications refer to communications, or exchanges of data, involving groups of processes coordinating together [85]. This abstraction of data movement among processes, along with efficient algorithms developed for each collective, is the reason why collective communications are so heavily relied upon in high-performance libraries [18, 23, 61, 65, 74, 84, 86]. The communication patterns associated with the set of collective communications considered in this document are illustrated in Figure 2.6.

Operation	Before				After			
	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
Permutation	$\mathbf{X}_0$	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_1$	$\mathbf{X}_0$	$\mathbf{X}_3$	$\mathbf{X}_2$
Broadcast	$\mathbf{X}$				$\mathbf{X}$	$\mathbf{X}$	$\mathbf{X}$	$\mathbf{X}$
Reduce-to-one	$\mathbf{X}^{(0)}$	$\mathbf{X}^{(1)}$	$\mathbf{X}^{(2)}$	$\mathbf{X}^{(3)}$	$\sum_j \mathbf{X}^{(j)}$			
Scatter	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$				$\mathbf{X}_0$	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$
Gather-to-one	$\mathbf{X}_0$	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$			
Allgather	$\mathbf{X}_0$	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$	$\mathbf{X}_0$ $\mathbf{X}_1$ $\mathbf{X}_2$ $\mathbf{X}_3$
Reduce-scatter	$\mathbf{X}_0^{(0)}$ $\mathbf{X}_1^{(0)}$ $\mathbf{X}_2^{(0)}$ $\mathbf{X}_3^{(0)}$	$\mathbf{X}_0^{(1)}$ $\mathbf{X}_1^{(1)}$ $\mathbf{X}_2^{(1)}$ $\mathbf{X}_3^{(1)}$	$\mathbf{X}_0^{(2)}$ $\mathbf{X}_1^{(2)}$ $\mathbf{X}_2^{(2)}$ $\mathbf{X}_3^{(2)}$	$\mathbf{X}_0^{(3)}$ $\mathbf{X}_1^{(3)}$ $\mathbf{X}_2^{(3)}$ $\mathbf{X}_3^{(3)}$	$\sum_j \mathbf{X}_0^{(j)}$	$\sum_j \mathbf{X}_1^{(j)}$	$\sum_j \mathbf{X}_2^{(j)}$	$\sum_j \mathbf{X}_3^{(j)}$
Allreduce	$\mathbf{X}^{(0)}$	$\mathbf{X}^{(1)}$	$\mathbf{X}^{(2)}$	$\mathbf{X}^{(3)}$	$\sum_j \mathbf{X}^{(j)}$	$\sum_j \mathbf{X}^{(j)}$	$\sum_j \mathbf{X}^{(j)}$	$\sum_j \mathbf{X}^{(j)}$
All-to-all	$\mathbf{X}_0^{(0)}$ $\mathbf{X}_1^{(0)}$ $\mathbf{X}_2^{(0)}$ $\mathbf{X}_3^{(0)}$	$\mathbf{X}_0^{(1)}$ $\mathbf{X}_1^{(1)}$ $\mathbf{X}_2^{(1)}$ $\mathbf{X}_3^{(1)}$	$\mathbf{X}_0^{(2)}$ $\mathbf{X}_1^{(2)}$ $\mathbf{X}_2^{(2)}$ $\mathbf{X}_3^{(2)}$	$\mathbf{X}_0^{(3)}$ $\mathbf{X}_1^{(3)}$ $\mathbf{X}_2^{(3)}$ $\mathbf{X}_3^{(3)}$	$\mathbf{X}_0^{(0)}$ $\mathbf{X}_0^{(1)}$ $\mathbf{X}_0^{(2)}$ $\mathbf{X}_0^{(3)}$	$\mathbf{X}_1^{(0)}$ $\mathbf{X}_1^{(1)}$ $\mathbf{X}_1^{(2)}$ $\mathbf{X}_1^{(3)}$	$\mathbf{X}_2^{(0)}$ $\mathbf{X}_2^{(1)}$ $\mathbf{X}_2^{(2)}$ $\mathbf{X}_2^{(3)}$	$\mathbf{X}_3^{(0)}$ $\mathbf{X}_3^{(1)}$ $\mathbf{X}_3^{(2)}$ $\mathbf{X}_3^{(3)}$

Figure 2.6: Collective communications considered in this dissertation. This figure has been reproduced here with minor modifications from [16, 73].

Following the notation used in Figure 2.6, each illustration depicts a collective communication involving a vector $\mathbf{X}$ comprised of n data elements partitioned as

$$\mathbf{X} = \left(\begin{array}{c} \mathbf{X}_0 \\ \hline \mathbf{X}_1 \\ \hline \vdots \\ \hline \mathbf{X}_{\hat{p}-1} \end{array} \right).$$

where $\hat{p}$ is the number of processes involved in the communication, $\mathbf{X}_u$ is comprised of n_u elements, and

$$\sum_{u \in \mathcal{R}(\hat{p})} n_u = n.$$

In the case of Figure 2.6, $\hat{p} = 4$.

Various reduction operations can be defined for reduction collectives (reduce-to-one, allreduce, and reduce-scatter); however, we only consider summation as a reduction operation in this document. For collectives involving reduction, the term $\mathbf{X}^{(v)}$ represents a contribution to $\mathbf{X}$ assigned to process u such that

$$\sum_{v \in \mathcal{R}(\hat{p})} \mathbf{X}^{(v)} = \mathbf{X}.$$

In the reduce-scatter collective, the contributions $\mathbf{X}^{(v)}$ are themselves partitioned before communication is initiated. For the allreduce and reduce-to-one collectives, the subvector is not partitioned.

For the all-to-all collective, the contribution $\mathbf{X}_u^{(v)}$ represents a subvector initially assigned to process u that must be transmitted to process v.

Each collective then exchanges some combination of the objects associated with $\mathbf{X}$, the subvectors, $\mathbf{X}_u$, and the contributions, $\mathbf{X}^{(v)}$ and $\mathbf{X}_u^{(v)}$. For example, if we look at the pattern for the broadcast collective, we see that initially one process is assigned the entire vector $\mathbf{X}$. Upon completion, each process involved in the collective receives the vector $\mathbf{X}$. In the case of the reduce-to-one collective, each process is assigned a contribution to $\mathbf{X}$ and, upon completion, a single process receives data that represents the summation of all contributions. Finally, if we look at the scatter collective, initially one process is assigned $\mathbf{X}$ and, upon completion, each process its associated subvector. All other collectives can be interpreted based on a combination of these examples.

If the size of all subvectors communicated (n_u) are approximately equal[2], then we say the associated communication is "balanced"; otherwise, the communication is "unbalanced". For an example of an unbalanced communication, consider that a broadcast collective can be implemented with an all-to-all collective. In this case, one process is assigned the entire vector $\mathbf{X}$. For this process, the size of the subvector assigned is n. Consequently, all other processes must be assigned a subvector of size zero. As the size of each subvector assigned to every process involved in the collective are not approximately equal, the all-to-all communication (when used in this context) is unbalanced.

We consider the cost of each collective listed in Figure 2.6 in terms of three parameters: the time to initiate communication (latency), represented by α; the time required to transmit data (inverse of bandwidth), represented by β; and the time to perform an arithmetic computation, represented by γ. For each collective in

[2] Each process is assigned either $\left\lfloor \frac{n}{\hat{p}} \right\rfloor$ or $\left\lfloor \frac{n}{\hat{p}} \right\rfloor + 1$ data elements.

Coll.	Latency	Bandw.	Comput.	Cost used
Allgather	$\lceil\log_2(\hat{p})\rceil\alpha$	$(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$	–	$\log_2(\hat{p})\alpha+(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$
Gather-to-one	$\lceil\log_2(\hat{p})\rceil\alpha$	$(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$	–	$\log_2(\hat{p})\alpha+(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$
Broadcast	$\lceil\log_2(\hat{p})\rceil\alpha$	$\overline{n}\beta$	–	$\log_2(\hat{p})\alpha+\overline{n}\beta$
Scatter	$\lceil\log_2(\hat{p})\rceil\alpha$	$(\hat{p}-1)\overline{n}\beta$	–	$\log_2(\hat{p})\alpha+(\hat{p}-1)\overline{n}\beta$
Reduce-scatter	$\lceil\log_2(\hat{p})\rceil\alpha$	$(\hat{p}-1)\overline{n}\beta$	$(\hat{p}-1)\overline{n}\gamma$	$\log_2(\hat{p})\alpha+(\hat{p}-1)\overline{n}(\beta+\gamma)$
Reduce-to-one	$\lceil\log_2(\hat{p})\rceil\alpha$	$\overline{n}\beta$	$\frac{(\hat{p}-1)}{\hat{p}}\overline{n}\gamma$	$\log_2(\hat{p})\alpha+\overline{n}(\beta+\gamma)$
Allreduce	$\lceil\log_2(\hat{p})\rceil\alpha$	$2(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$	$\frac{(\hat{p}-1)}{\hat{p}}\overline{n}\gamma$	$\log_2(\hat{p})\alpha+(\hat{p}-1)\frac{\overline{n}}{\hat{p}}(2\beta+\gamma)$
Permutation	α	$\overline{n}\beta$	–	$\alpha+\overline{n}\beta$
All-to-all	$\lceil\log_2(\hat{p})\rceil\alpha$	$(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$	–	$\log_2(\hat{p})\alpha+(\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta$

Figure 2.7: Lower bounds associated with different aspects of each collective in Figure 2.6 when involved in balanced communications. The constants α, β, and γ are the latency, bandwidth, and computation terms respectively. The lower bounds for each aspect are determined independent of the other two. Here, $\hat{p}$ refers to the number of processes involved in the collective and $\overline{n}$ refers to the maximum number of elements assigned to a process based on the *output* distribution. Conditions for the lower bounds are given in [16] and [13]. This table is adapted from [16, 73].

Figure 2.6, the lower bounds associated with each of these parameters for balanced communications is given in Figure 2.7. The lower bounds of each parameter of the collectives are determined independent of the other two. Detailed discussions including the conditions assumed for determining the lower bounds are given in [16] and [13]. Other costs models can be used for each collective considered; we merely use these to demonstrate the ideas developed in this dissertation.

To simplify discussions involving communication cost, we assume the number of processes involved in the communication evenly divide the number of elements being communicated. This avoids needlessly complex cost analyses. For similar reasons, we assume the number of processes involved in a collective communication is a power of two.

We encode only the “balanced” forms of the collectives given in Figure 2.6. However, some redistributions defined in the notation are inherently unbalanced. In Chapter 4, we discuss transformations that can be applied to these redistributions, under certain conditions to perform the equivalent redistribution with balanced communications. The goal of this process is to reduce the cost associated with communication.

2.5 Data Redistributions

Our goal in this section is to formalize collective communications in terms of redistributions described by the notation developed in Section 2.3.

We begin with examples from parallel matrix-matrix multiplication that highlight situations where different collective communications are utilized to create algorithms with efficient implementations. These examples come from a family, or class, of algorithms for matrix-matrix multiplication that assume one operand is significantly larger (in terms of number of elements) than the other two. To reduce the volume of data communicated, algorithms in this family communicate only data of the “smaller” operands and data of the “large” operand is not communicated at all. Algorithms based on this principle are referred to as *stationary.*

A detailed discussion of each stationary algorithm for matrix-matrix multiplication, along with the derivation process used to arrive at each algorithm, is given in Morrow et al. [30] and Schatz et al. [73]. Here, we merely restate the results using our notation.

For each algorithm discussed, an illustration is provided for visual reference (Figures 2.9, 2.10, 2.11, and 2.12). In each of these figures, we show how each matrix is distributed during each step of the algorithm and provide the series of collectives required to perform the associated redistribution. To conserve space, we depict the distributions of the output object along with any introduced temporaries in the same location of the illustration.

For the example algorithms discussed, we assume computation is begin performed on a processing mesh of size $\mathbf{P} = (P_0, P_1)$ and the cost associated with each collective communication as given in Figure 2.7. Additionally, we assume that integral ratios exist between dimensions of the distributed object and those of the processing mesh. This is done only to simplify the discussion of cost analyses.

We use MATLAB-style notation to refer to a column or row of a matrix; i.e., $\mathbf{A}_{:,\ell}$ refers to the ℓ-th column of the matrix $\mathbf{A}$, and $\mathbf{A}_{\ell,:}$ refers to the ℓ-th row of the matrix $\mathbf{A}$. We interpret columns and rows of matrices as order-1 objects for consistency in the developed notation for data distribution.

2.5.1 Example: Stationary C Parallel Matrix Multiplication

Consider the operation $\mathbf{C} = \mathbf{AB} + \mathbf{C}$ where $\mathbf{C}$ is an $I_0 \times I_1$ matrix, $\mathbf{A}$ is an $I_0 \times I_2$ matrix, and I_2 is small relative to I_0 and I_1. For purposes of this example, we assume an initial distribution of $\mathbf{A}\left[(0)\,,(1)\right]$, $\mathbf{B}\left[(0)\,,(1)\right]$, and $\mathbf{C}\left[(0)\,,(1)\right]$. An illustration of this initial distribution on a processing mesh of size $\mathbf{P} = (2, 3)$ is given in Figure 2.8. Observe that column u of the matrix $\mathbf{A}\left[(0)\,,(1)\right]$ is distributed as

$$\mathbf{A}_{:,\ell}\left[(0)\,;(1)\,,w\right],$$

$\mathbf{p}=(0,0)$ $\begin{matrix} a_{0,0} & a_{0,3} & \cdots \\ a_{2,0} & a_{2,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,1)$ $\begin{matrix} a_{0,1} & a_{0,4} & \cdots \\ a_{2,1} & a_{2,4} & \cdots \\ \vdots & \vdots & \vdots \end{matrix}$	$\mathbf{p}=(0,2)$ $\begin{matrix} a_{0,2} & a_{0,5} & \cdots \\ a_{2,2} & a_{2,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
$\mathbf{p}=(1,0)$ $\begin{matrix} a_{1,0} & a_{1,3} & \cdots \\ a_{3,0} & a_{3,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,1)$ $\begin{matrix} a_{1,1} & a_{1,4} & \cdots \\ a_{3,1} & a_{3,4} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,2)$ $\begin{matrix} a_{1,2} & a_{1,5} & \cdots \\ a_{3,2} & a_{3,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$

Figure 2.8: Graphical depiction of the matrix $\mathbf{A}$ distributed according to $\mathbf{A}\left[(0),(1)\right]$. The top-left entry of every container corresponds to the process's location within the mesh.

where $w = \ell \bmod \operatorname{prod}(\mathbf{P},(1))$. Similarly, under this distribution, the row u of $\mathbf{B}\left[(0),(1)\right]$ is distributed as

$$\mathbf{B}_{\ell,:}\left[(1);(0),w\right]$$

where $w = \ell \bmod \operatorname{prod}(\mathbf{P},(0))$.

We consider an algorithm that computes the matrix product as the summation of outer-products, as follows. Partition $\mathbf{A}$ by columns and $\mathbf{B}$ by rows so that

$$\mathbf{A} = \left(\begin{array}{c|c|c|c} \mathbf{A}_{:,0} & \mathbf{A}_{:,1} & \cdots & \mathbf{A}_{:,I_2-1} \end{array} \right), \quad \mathbf{B} = \left(\begin{array}{c} \mathbf{B}_{0,:} \\ \hline \mathbf{B}_{1,:} \\ \hline \vdots \\ \hline \mathbf{B}_{I_2-1,:} \end{array} \right),$$

and

$$\mathbf{C} = ((\cdots((\mathbf{C} + \mathbf{A}_{:,0}\mathbf{B}_{0,:}) + \mathbf{A}_{:,1}\mathbf{B}_{1,:})\cdots) + \mathbf{A}_{:,I_2-1}\mathbf{B}_{I_2-1,:}).$$

One approach that can achieve high performance for this algorithm is to loop over columns of **A** and rows of **B**, broadcast the current column of **A** to processes within the same mesh row and broadcast the current row of **B** to processes within the same mesh column. This algorithm, along with an example of the algorithm being performed is given in Figure 2.9. After the communications of the appropriate portions of **A** and **B** complete, the locally stored component of **C** is updated via a local rank-1 update. This approach is closely related to the Scalable Universal Matrix Multiplication Algorithm (SUMMA) differing only in the distributions of the matrices assumed and that it operates with columns and rows rather than blocks of columns and rows [87].

Let $\hat{p} = P_0 P_1$. Under the assumed cost model, this algorithm has a cost of

$$\begin{aligned} & 2\frac{I_0 I_1 I_2}{\hat{p}}\gamma \quad + \quad I_2 \underbrace{\left(\log_2(P_1)\alpha + \frac{I_0}{P_0}\beta\right)}_{\text{Broadcast } \mathbf{A}_{:,u} \text{ in cols}} \\ & \qquad\qquad + \quad I_2 \underbrace{\left(\log_2(P_0)\alpha + \frac{I_1}{P_1}\beta\right)}_{\text{Broadcast } \mathbf{B}_{u,:} \text{ in rows}} \qquad (2.4) \\ = \quad & 2\frac{I_0 I_1 I_2}{\hat{p}}\gamma \quad + \quad I_2 \log_2(\hat{p})\alpha + \frac{P_1 I_0 + P_0 I_1}{\hat{p}} I_2 \beta. \end{aligned}$$

Assuming extra memory is available to store the encountered duplications of data, we recognize that the series of broadcast collectives performed in the previous algorithm can be implemented as a single allgather collective. Additionally, we can convert the series of local updates to a single matrix-matrix multiplication. The algorithm resulting from this series of refactorings, along with an illustration of the algorithm, is given in Figure 2.10.

```
for $\ell = 0, \ldots, I_2 - 1$
    $\mathbf{A}_{:,\ell}\left[(0)\right] \leftarrow \mathbf{A}_{:,\ell}\left[(0)\,;(1)\,,w\right]$    (Broadcast in mode 1)
    $\mathbf{B}_{\ell,:}\left[(1)\right] \leftarrow \mathbf{B}_{\ell,:}\left[(1)\,;(0)\,,w\right]$    (Broadcast in mode 0)
    $\mathbf{C}\left[(0)\,,(1)\right] := \mathbf{C}\left[(0)\,,(1)\right] + \mathbf{A}_{:,\ell}\left[(0)\right]\mathbf{B}_{\ell,:}\left[(1)\right]$    (Local rank-1 update)
endfor
```

(a) Pseudo-algorithm

A		**B**		**C**	
$a_{0,\ell}$		$b_{\ell,0}$ $b_{\ell,2}$	$b_{\ell,1}$ $b_{\ell,3}$	$c_{0,0}$ $c_{0,2}$	$c_{0,1}$ $c_{0,3}$
$a_{2,\ell}$				$c_{2,0}$ $c_{2,2}$	$c_{2,1}$ $c_{2,3}$
$a_{1,\ell}$				$c_{1,0}$ $c_{1,2}$	$c_{1,1}$ $c_{1,3}$
$a_{3,\ell}$				$c_{3,0}$ $c_{3,2}$	$c_{3,1}$ $c_{3,3}$

↓ 1. Broadcast $\mathbf{A}_{:,\ell}$ within mode 1.

A		**B**		**C**	
$a_{0,\ell}$	$a_{0,\ell}$	$b_{\ell,0}$ $b_{\ell,2}$	$b_{\ell,1}$ $b_{\ell,3}$	$c_{0,0}$ $c_{0,2}$	$c_{0,1}$ $c_{0,3}$
$a_{2,\ell}$	$a_{2,\ell}$			$c_{2,0}$ $c_{2,2}$	$c_{2,1}$ $c_{2,3}$
$a_{1,\ell}$	$a_{1,\ell}$			$c_{1,0}$ $c_{1,2}$	$c_{1,1}$ $c_{1,3}$
$a_{3,\ell}$	$a_{3,\ell}$			$c_{3,0}$ $c_{3,2}$	$c_{3,1}$ $c_{3,3}$

↓ 2. Broadcast $\mathbf{B}_{\ell,:}$ within mode 0.

A		**B**		**C**	
$a_{0,\ell}$	$a_{0,\ell}$	$b_{\ell,0}$ $b_{\ell,2}$	$b_{\ell,1}$ $b_{\ell,3}$	$c_{0,0}$ $c_{0,2}$	$c_{0,1}$ $c_{0,3}$
$a_{2,\ell}$	$a_{2,\ell}$			$c_{2,0}$ $c_{2,2}$	$c_{2,1}$ $c_{2,3}$
$a_{1,\ell}$	$a_{1,\ell}$	$b_{\ell,0}$ $b_{\ell,2}$	$b_{\ell,1}$ $b_{\ell,3}$	$c_{1,0}$ $c_{1,2}$	$c_{1,1}$ $c_{1,3}$
$a_{3,\ell}$	$a_{3,\ell}$			$c_{3,0}$ $c_{3,2}$	$c_{3,1}$ $c_{3,3}$

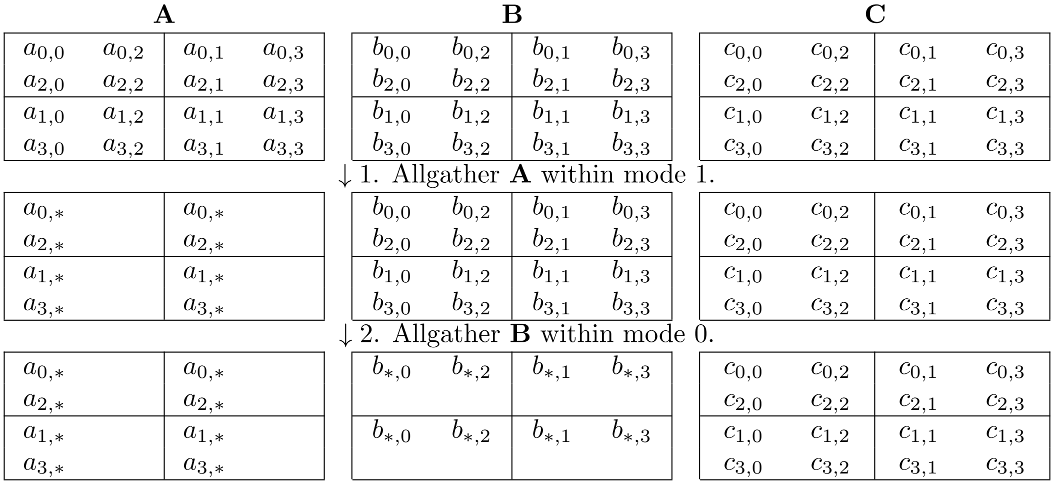
3. Update $\mathbf{C}$ via local rank-1 update.

(b) Illustration of single iteration (of I_2) where $w = 0$

Figure 2.9: Stationary C algorithm based on broadcast collectives performed on a 2×2 processing mesh.

```
$\mathbf{A}[(0),()] \leftarrow \mathbf{A}[(0),(1)]$                                      (Allgather in mode 1)
$\mathbf{B}[(),(1)] \leftarrow \mathbf{B}[(0),(1)]$                                      (Allgather in mode 0)
$\mathbf{C}[(0),(1)] := \mathbf{C}[(0),(1)] + \mathbf{A}[(0),()]\,\mathbf{B}[(),(1)]$   (Matrix-matrix multiply)
```

(a) Pseudo-algorithm



A				**B**				**C**			
$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,0}$	$b_{0,2}$	$b_{0,1}$	$b_{0,3}$	$c_{0,0}$	$c_{0,2}$	$c_{0,1}$	$c_{0,3}$
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{2,0}$	$b_{2,2}$	$b_{2,1}$	$b_{2,3}$	$c_{2,0}$	$c_{2,2}$	$c_{2,1}$	$c_{2,3}$
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{1,0}$	$b_{1,2}$	$b_{1,1}$	$b_{1,3}$	$c_{1,0}$	$c_{1,2}$	$c_{1,1}$	$c_{1,3}$
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{3,0}$	$b_{3,2}$	$b_{3,1}$	$b_{3,3}$	$c_{3,0}$	$c_{3,2}$	$c_{3,1}$	$c_{3,3}$

↓ 1. Allgather **A** within mode 1.

$a_{0,*}$		$a_{0,*}$		$b_{0,0}$	$b_{0,2}$	$b_{0,1}$	$b_{0,3}$	$c_{0,0}$	$c_{0,2}$	$c_{0,1}$	$c_{0,3}$
$a_{2,*}$		$a_{2,*}$		$b_{2,0}$	$b_{2,2}$	$b_{2,1}$	$b_{2,3}$	$c_{2,0}$	$c_{2,2}$	$c_{2,1}$	$c_{2,3}$
$a_{1,*}$		$a_{1,*}$		$b_{1,0}$	$b_{1,2}$	$b_{1,1}$	$b_{1,3}$	$c_{1,0}$	$c_{1,2}$	$c_{1,1}$	$c_{1,3}$
$a_{3,*}$		$a_{3,*}$		$b_{3,0}$	$b_{3,2}$	$b_{3,1}$	$b_{3,3}$	$c_{3,0}$	$c_{3,2}$	$c_{3,1}$	$c_{3,3}$

↓ 2. Allgather **B** within mode 0.

$a_{0,*}$		$a_{0,*}$		$b_{*,0}$	$b_{*,2}$	$b_{*,1}$	$b_{*,3}$	$c_{0,0}$	$c_{0,2}$	$c_{0,1}$	$c_{0,3}$
$a_{2,*}$		$a_{2,*}$						$c_{2,0}$	$c_{2,2}$	$c_{2,1}$	$c_{2,3}$
$a_{1,*}$		$a_{1,*}$		$b_{*,0}$	$b_{*,2}$	$b_{*,1}$	$b_{*,3}$	$c_{1,0}$	$c_{1,2}$	$c_{1,1}$	$c_{1,3}$
$a_{3,*}$		$a_{3,*}$						$c_{3,0}$	$c_{3,2}$	$c_{3,1}$	$c_{3,3}$

3. Update **C** via local matrix-matrix multiplication.

(b) Illustration

Figure 2.10: Stationary C algorithm based on allgather collective performed on a 2×2 mesh. Here, "$*$" indicates all indices of a mode.

Under the assumed cost model, the approach based on allgather collectives has a cost of

$$
\begin{aligned}
2\frac{I_0 I_1 I_2}{\hat{p}}\gamma \quad &+ \quad \underbrace{\log_2(P_1)\alpha + \frac{P_1 - 1}{P_1}\frac{I_0 I_2}{P_0}\beta}_{\text{Allgather } \mathbf{A} \text{ in rows}} \\
&+ \quad \underbrace{\log_2(P_0)\alpha + \frac{P_0 - 1}{P_0}\frac{I_2 I_1}{P_1}\beta}_{\text{Allgather } \mathbf{B} \text{ in cols}} \qquad (2.5) \\
= \quad 2\frac{I_0 I_1 I_2}{\hat{p}}\gamma \quad &+ \quad \log_2(\hat{p})\alpha + \frac{(P_1 - 1)\, I_0 + (P_0 - 1)\, I_1}{\hat{p}} I_2 \beta.
\end{aligned}
$$

Comparing (2.4) with (2.5), we see that the algorithm based on allgather collectives reduces the latency term by a factor I_2; however, this comes at a cost of additional memory required to store the duplicated matrices.

Based on this example, we see the benefits of formalizing the broadcast and allgather collectives: Broadcast collectives can be used to design an algorithm that conserves memory at the expense of additional latency and bandwidth costs, while allgather collectives can be used to design an algorithm that reduces overall cost at the expense of extra memory required. The correct choice between these two algorithms depends on the problem dimensions and the computing environment (e.g., α, β, and γ); therefore, being able to select among these two algorithms is useful. We mention here that one other benefit of the broadcast-based algorithm is that the algorithm can be restructured in such a way that the communication can be effectively pipelined, thereby reducing the overhead due to communication [17, 87].

It is important to mention here that there is one source of additional cost in the algorithm based on broadcast collectives that is not reflected in the assumed cost models. In the algorithm based on broadcast collectives, the local computations performed are expressed as rank-1 updates as opposed to matrix-matrix multiplications which are performed in the algorithm based on allgather collectives. Operations cast in terms of matrix-matrix multiplications are typically far more efficient than the equivalent operation cast in terms of matrix-vector multiplications [29, 88]. Therefore, not only does the algorithm based on allgather collectives have a lower cost in terms of communication, the local computation will likely outperform that of the algorithm based on broadcast collectives (as described).

2.5.2 Example: Stationary A Parallel Matrix Multiplication

We again consider the $\mathbf{C} = \mathbf{AB} + \mathbf{C}$ operation, only now we assume that I_1 is small relative to I_0 and I_2. We again assume an initial distribution of $\mathbf{A}\left[(0),(1)\right]$, $\mathbf{B}\left[(0),(1)\right]$, and $\mathbf{C}\left[(0),(1)\right]$.

Partition $\mathbf{C}$ by columns and $\mathbf{B}$ by columns so that

$$\mathbf{C} = \left(\begin{array}{c|c|c|c} \mathbf{C}_{:,0} & \mathbf{C}_{:,1} & \ldots & \mathbf{C}_{:,I_1-1} \end{array} \right), \mathbf{B} = \left(\begin{array}{c|c|c|c} \mathbf{B}_{:,0} & \mathbf{B}_{:,1} & \ldots & \mathbf{B}_{:,I_1-1} \end{array} \right),$$

and

$$\mathbf{C}_{:,\ell} = \mathbf{A}\mathbf{B}_{:,\ell} + \mathbf{C}_{:,\ell}.$$

One approach to implement the computation is to loop over columns of $\mathbf{B}$, redistributing columns $\mathbf{B}_{:,u}$ appropriately so that simultaneous matrix-vector multiplications can be performed, and finally performing a reduction across processes to compute $\mathbf{C}_{:,u}$ leaving it correctly distributed. This approach with details filled in, along with an illustration of the algorithm, is given in Figure 2.11.

```
for $\ell = 0, \ldots, I_1 - 1$
    $\mathbf{B}_{:,\ell}\left[(1)\right] \leftarrow \mathbf{B}_{:,\ell}\left[(0)\,;(1)\,,w\right]$    (Scatter in mode 1, Permutation,
                                                     Allgather in mode 0)
    $\mathbf{T}\left[(0)\,,(1)\right] := \mathbf{A}\left[(0)\underset{\smile}{,}(1)\right]\mathbf{B}_{:,\ell}\left[(1)\right]$    (Local Matrix-vector Multiply)
    $\mathbf{C}_{:,\ell}\left[(0)\,;(1)\,,w\right] \leftarrow \overset{\frown}{\sum}\mathbf{T}\left[(0)\,,(1)\right]$    (Reduce-to-one update in mode 1)
endfor
```

(a) Definition

A		**B**		**C/T**	
$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$ $b_{2,\ell}$			
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{1,\ell}$ $b_{3,\ell}$			

↓ 1. Scatter $\mathbf{B}_{:,\ell}$ within mode 1.

$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$	$b_{2,\ell}$		
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{1,\ell}$	$b_{3,\ell}$		

↓ 2. Permutation $\mathbf{B}_{:,\ell}$.

$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$	$b_{1,\ell}$		
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{2,\ell}$	$b_{3,\ell}$		

↓ 3. Allgather $\mathbf{B}_{:,\ell}$ within mode 0.

$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$		
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$		

↓ 4. Compute $\mathbf{T}$ via local matrix-matrix multiplies.

$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$	$t_{0,0}$ $t_{2,0}$	$t_{0,1}$ $t_{2,1}$
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$	$t_{1,0}$ $t_{3,0}$	$t_{1,1}$ $t_{3,1}$

↓ 5. Update $\mathbf{C}_{:,\ell}$ via reduce-to-one within mode 1.

$a_{0,0}$ $a_{0,2}$ $a_{2,0}$ $a_{2,2}$	$a_{0,1}$ $a_{0,3}$ $a_{2,1}$ $a_{2,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$	$c_{0,\ell}$ $c_{2,\ell}$	
$a_{1,0}$ $a_{1,2}$ $a_{3,0}$ $a_{3,2}$	$a_{1,1}$ $a_{1,3}$ $a_{3,1}$ $a_{3,3}$	$b_{0,\ell}$ $b_{2,\ell}$	$b_{1,\ell}$ $b_{3,\ell}$	$c_{1,\ell}$ $c_{3,\ell}$	

(b) Illustration of single iteration (of I_1) where $w = 0$

Figure 2.11: Stationary A algorithm based on reduce-to-one collectives performed on a 2×2 mesh.

We use the notation $\widetilde{\sum}_{\iota\eta}$ to indicate a contraction that is to be performed via a collective that communicates over modes of the processing mesh used to distribute the tensor modes labeled ι and η. Similarly, we use the location $\widehat{\sum}_{\iota\eta}$ to indicate a contraction that is to be performed locally by each process. We omit the specific labels to perform a contraction over when it is clear from the computation which modes are being contracted (for instance when computing a matrix-matrix multiplication).

Note that the reason for the final global reduction stems from the fact that, during the parallel matrix-vector multiplication, no process is assigned all indices of the mode involved in the summation. This means that no process can fully compute the locally stored portion of $\mathbf{C}_{:,u}$ and therefore a reduction of elements across processes is required to accumulate the local contributions. This is the same reason for the introduction of the (temporary) matrix $\mathbf{T}$.

Under the assumed cost model, this algorithm has a cost of

$$
\begin{aligned}
& 2\frac{I_0 I_1 I_2}{\hat{p}} && + \quad I_1 \underbrace{\left(\log_2 (P_1)\alpha + \frac{P_1 - 1}{P_1}\frac{I_2}{P_0}\beta\right)}_{\text{Scatter } \mathbf{B}_{:,u} \text{ in mode } 1} \\
& && + \quad I_1 \underbrace{\left(\alpha + \frac{I_2}{P_0 P_1}\beta\right)}_{\text{Permutation } \mathbf{B}_{:,u}} \\
& && + \quad I_1 \underbrace{\left(\log_2 (P_0)\alpha + (P_0 - 1)\frac{I_2}{P_0 P_1}\beta\right)}_{\text{Allgather } \mathbf{B}_{:,u} \text{ in mode } 0} \\
& && + \quad I_1 \underbrace{\left(\log_2 (P_1)\alpha + \frac{I_0}{P_0}\beta\right)}_{\text{Reduce-to-one } \mathbf{T} \text{ in mode } 1} \\
= \quad & 2\frac{I_0 I_1 I_2}{\hat{p}} && + \quad I_1 \left(\log_2 (\hat{p}) + P_1 + 1\right)\alpha \\
& && + \quad \frac{(P_0 + P_1 - 1)\, I_2 + P_1 I_0}{\hat{p}} I_1 \beta.
\end{aligned}
\tag{2.6}
$$

Once again, if additional memory is available to perform the computation, we recognize that the series of scatter collectives (with different root processes) can be implemented as an all-to-all, the series of permutation collectives can be implemented as a single permutation, the series of allgather collectives can be implemented as a single allgather (on more data), and the series of reduce-to-one collectives (with different root processes) can be implemented as a reduce-scatter. Further, the series of matrix-vector multiplications can be implemented as a single matrix-matrix multiplication that, instead of forming a matrix $\mathbf{T}$, forms an order-3 tensor $\mathbf{T}$. In Chapter 3 we discuss the origin of $\mathbf{T}$. The algorithm resulting from this series of refactorings, along with an illustration of the algorithm, is given in Figure 2.12.

$\mathbf{B}\left[(1),()\right] \leftarrow \mathbf{B}\left[(0),(1)\right]$ (All-to-all in mode 1, Permutation, Allgather in mode 0)

$\mathbf{T}\left[(0),(),(1)\right] \mathrel{\widetilde{:=}} \mathbf{A}\left[(0),(1)\right]\mathbf{B}\left[(1),()\right]$ (Local Matrix-matrix Multiply

$\mathbf{C}\left[(0),(1)\right] \leftarrow \sum \mathbf{T}\left[(0),(),(1)\right]$ (Reduce-scatter updates in mode 1)

(a) Definition

A				**B**				**C/T**			
$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,0}$	$b_{0,2}$	$b_{0,1}$	$b_{0,3}$				
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{2,0}$	$b_{2,2}$	$b_{2,1}$	$b_{2,3}$				
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{1,0}$	$b_{1,2}$	$b_{1,1}$	$b_{1,3}$				
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{3,0}$	$b_{3,2}$	$b_{3,1}$	$b_{3,3}$				

↓ 1. All-to-all **B** within rows.

$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,0}$	$b_{0,2}$	$b_{2,0}$	$b_{2,2}$				
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{0,1}$	$b_{0,3}$	$b_{2,1}$	$b_{2,3}$				
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{1,0}$	$b_{1,2}$	$b_{3,0}$	$b_{3,2}$				
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{1,1}$	$b_{1,3}$	$b_{3,1}$	$b_{3,3}$				

↓ 2. Permutation **B** within rows/cols.

$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,0}$	$b_{0,2}$	$b_{1,0}$	$b_{1,2}$				
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{0,1}$	$b_{0,3}$	$b_{1,1}$	$b_{1,3}$				
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{2,0}$	$b_{2,2}$	$b_{3,0}$	$b_{3,2}$				
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{2,1}$	$b_{2,3}$	$b_{3,1}$	$b_{3,3}$				

↓ 3. Allgather **B** within cols.

$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,*}$		$b_{1,*}$					
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{2,*}$		$b_{3,*}$					
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{0,*}$		$b_{1,*}$					
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{2,*}$		$b_{3,*}$					

↓ 4. Compute **T** via local matrix-matrix multiplies.

$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,*}$		$b_{1,*}$		$t_{0,*,0}$		$t_{0,*,1}$	
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{2,*}$		$b_{3,*}$		$t_{2,*,0}$		$t_{2,*,1}$	
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{0,*}$		$b_{1,*}$		$t_{1,*,0}$		$t_{1,*,1}$	
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{2,*}$		$b_{3,*}$		$t_{3,*,0}$		$t_{3,*,1}$	

↓ 5. Update **C** via Reduce-scatter within rows.

$a_{0,0}$	$a_{0,2}$	$a_{0,1}$	$a_{0,3}$	$b_{0,*}$		$b_{1,*}$		$c_{0,0}$	$c_{0,2}$	$c_{0,1}$	$c_{0,3}$
$a_{2,0}$	$a_{2,2}$	$a_{2,1}$	$a_{2,3}$	$b_{2,*}$		$b_{3,*}$		$c_{2,0}$	$c_{2,2}$	$c_{2,1}$	$c_{2,3}$
$a_{1,0}$	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$b_{0,*}$		$b_{1,*}$		$c_{1,0}$	$c_{1,2}$	$c_{1,1}$	$c_{1,3}$
$a_{3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,3}$	$b_{2,*}$		$b_{3,*}$		$c_{3,0}$	$c_{3,2}$	$c_{3,1}$	$c_{3,3}$

(b) Illustration

Figure 2.12: Stationary A algorithm based on reduce-scatter collectives performed on a 2 × 2 mesh. Here, "$*$" indicates all indices of a mode.

Under the assumed cost model, this algorithm has a cost of

$$\begin{aligned}
& 2\frac{I_0 I_1 I_2}{\hat{p}} && + \underbrace{\log_2(P_1)\alpha + \frac{P_1-1}{P_1}\frac{I_1 I_2}{P_0 P_1}\beta}_{\text{All-to-all } \mathbf{B} \text{ in mode 1}} \\
& && + \underbrace{\alpha + \frac{I_1 I_2}{P_0 P_1}\beta}_{\text{Permutation } \mathbf{B}} \\
& && + \underbrace{\log_2(P_0)\alpha + (P_0-1)\frac{I_1 I_2}{P_0 P_1}\beta}_{\text{Allgather } \mathbf{B} \text{ in mode 1}} \qquad (2.7) \\
& && + \underbrace{\log_2(P_1)\alpha + (P_1-1)\frac{I_0 I_1}{P_0 P_1}\beta}_{\text{Reduce-scatter } \mathbf{T} \text{ in mode 1}} \\
= \; & 2\frac{I_0 I_1 I_2}{\hat{p}} && + (\log_2(\hat{p}) + P_1 + 1)\alpha \\
& && + \frac{(\frac{P_1-1}{P_1} + P_0)I_2 + (P_1-1)I_0}{\hat{p}} I_1 \beta.
\end{aligned}$$

Comparing (2.6) with (2.7), we see that the algorithm resulting from the series of refactorings reduces the latency term by a factor n and also slightly reduces the bandwidth term; again, this increase in modeled performance comes at the expense of additional memory required.

Based on this example, we see the utility of formalizing the scatter, all-to-all, permutation, reduce-to-one, and reduce-scatter collectives. Once again, the optimal choice depends on the problem specification and computing environment, therefore being able to choose among these two algorithms is useful.

2.5.3 Example: Allreduce and Gather-to-one

Up to now, we have provided evidence for the utility of all collectives considered except for the allreduce, and gather-to-one collectives. The benefits of the allreduce and gather-to-one collectives do not easily manifest when only considering a single matrix-matrix operation. However, when considering a series of matrix-matrix multiplications, such as

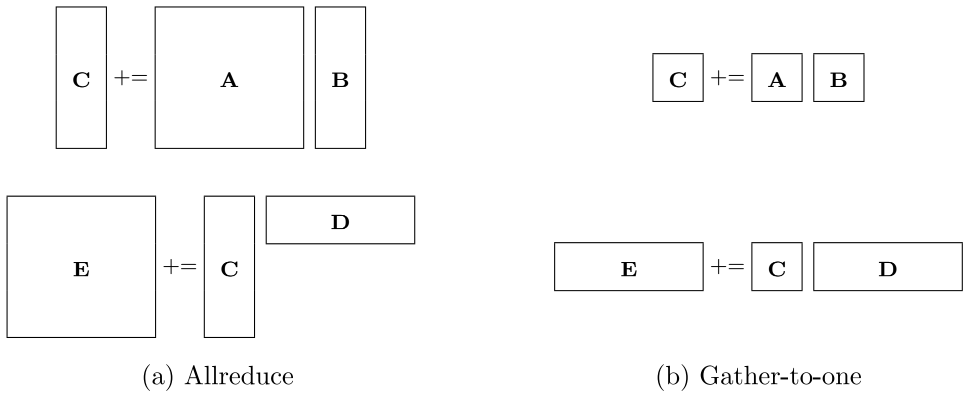
$$\mathbf{C} = \mathbf{AB} \tag{2.8}$$

followed by

$$\mathbf{E} = \underbrace{(\mathbf{AB})}_{\mathbf{C}} \mathbf{D} + \mathbf{E} \tag{2.9}$$

where $\mathbf{E}$ is an $I_0 \times I_1$ matrix, $\mathbf{C}$ is an $I_0 \times I_2$ matrix, $\mathbf{A}$ is an $I_0 \times I_3$ matrix, and both matrices $\mathbf{B}$ and $\mathbf{D}$ are of conformal size, the utility of both allreduce and gather-to-one collectives becomes more readily apparent. The relative sizes of the matrices involved to demonstrate the utility of both the allreduce and gather-to-one collectives are given in Figure 2.13.

To see the utility of the allreduce collective, consider the case where I_2 is small relative to all other dimensions (Figure 2.13a). Applying what we discussed in previous subsections concerning stationary algorithmic variants, we recognize that a stationary A algorithm is appropriate for computing (2.8), while a stationary C algorithm is appropriate for computing (2.9) as these algorithms do not communicate the "large" operand in the associated matrix-matrix multiplication. An examination of



(a) Allreduce

(b) Gather-to-one

Figure 2.13: Example problem sizes depicting utility of allreduce and gather-to-one collectives.

Figure 2.12 and Figure 2.10 reveals that the final communication of (2.8) is a reduce-scatter to update elements of $\mathbf{C}$, whereas the first communication of (2.9) involving $\mathbf{C}$ is an allgather, both occurring over the row dimension. Instead of incurring the latency cost of initiating both a reduce-scatter and an allgather collective, we recognize that a reduce-scatter followed by an allgather over the same set of processing mesh modes is equivalent to an allreduce, so the latency is reduced by replacing the associated collectives with an allreduce collective. However, having said this, the benefit of using an allreduce collective over instead of a reduce-scatter and allgather collective still only manifests when each operand is appropriately sized. The allreduce collective has also been shown to be useful in some parallel algorithms that perform the QR decomposition of a matrix and reconstruct associated Householder vectors [6].

For a small amount of data, it has been shown that an allgather implemented as a gather-to-one followed by a broadcast collective is most efficient [16]. Consider the case where I_0, I_2, I_3 are small relative to I_1 (Figure 2.13b). Let us assume that for both multiplications, the stationary C variant is best (simple arguments can be made to justify this). Then, the first collective involving $\mathbf{C}$ in (2.9) is an allgather. This means that a small amount of data is required to be duplicated among processes. Further, the amount of data required to compute $\mathbf{C}$ is relatively small. We can either perform this computation by first performing allgather collectives to replicate $\mathbf{A}$ and $\mathbf{B}$, and redundantly compute $\mathbf{C}$, or we can gather both $\mathbf{A}$ and $\mathbf{B}$ to a single process, compute $\mathbf{C}$, and replicate the result via a broadcast collective to all other processes. As the amount of computation being done is the same in both approaches (redundant computation takes the same amount of time to perform), the latter approach is the better option as it uses an allgather implementation more tailored to the relative size of the data involved.

2.5.4 Collective Redistribution Rules

Previous subsections discussed examples of uses for each collective considered in this work. In Figure 2.14 and Figure 2.15, we provide a table showing the redistributions each collective can directly implement based on the defined notation for data distributions[3] with an associated cost as given in Figure 2.7. Proofs of the rules defined in Figure 2.14 and Figure 2.15 are given in Appendix A. Again, for those familiar with the Elemental library, considering only matrices distributed on order-two processing meshes, the set of redistributions defined in this work express all redistributions defined by the Elemental library. Figure 2.16 connects the redistribution rules used in the Elemental library to those defined in this work.

Each rule in Figure 2.14 and Figure 2.15 is to be interpreted as simultaneous collective communication instances each communicating over the processing mesh modes

[3] All other redistributions can be implemented via combinations of these rules.

	Collective	Redistribution over modes $\bigcup_{m\in\mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m\in\mathcal{R}(M)} \overline{\mathcal{D}}^{(m)}$
Consistent load balance	Allgather	$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right]$
	Permutation	$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$
Inconsistent load balance	Gather-to-one	$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), (w + j \cdot \text{prod}\,(\mathbf{P}, \mathcal{E}))\right]$
	Broadcast	$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), (w + j \cdot \text{prod}\,(\mathbf{P}, \mathcal{E}))\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right]$
	Scatter	$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), (w + j \cdot \text{prod}\,(\mathbf{P}, \mathcal{E}))\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$

Figure 2.14: Redistributions associated with balanced collectives on processing mesh of size $\mathbf{P}$. Here, $(v-1) \in \mathcal{R}\,(M)$, $w \in \mathcal{R}\,(\text{prod}\,(\mathbf{P}, \mathcal{E}))$, and $j \in \mathcal{R}\left(\text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}\right)\right)$. The redistribution associated with the permutation collective applies only if the additional constraint that $\text{prod}\left(\mathbf{P}, \overline{\mathcal{D}}^{(m)}\right) = \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m)}\right)$ for $m \in \mathcal{R}\,(M)$ holds.

	Collective	Redistribution over modes
		$\bigcup_{m\in\mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m\in\mathcal{R}(M)} \overline{\mathcal{D}}^{(m)}$
Consistent load balance	Reduce-scatter	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A}\sqcup\mathcal{K}}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)},\widetilde{\mathcal{D}}^{(0)},\ldots,\widetilde{\mathcal{D}}^{(M-1-v)};\mathcal{E},w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)}\sqcup\overline{\mathcal{D}}^{(0)},\ldots,\mathcal{D}^{(v-1)}\sqcup\overline{\mathcal{D}}^{(v-1)};\mathcal{E},w\right]$
	Allreduce	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A}\sqcup\mathcal{K}}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)},\widetilde{\mathcal{D}}^{(0)},\ldots,\widetilde{\mathcal{D}}^{(M-1-v)};\mathcal{E},w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)};\mathcal{E},w\right]$
	All-to-all	$\mathbf{A}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)},\mathcal{D}^{(v)}\sqcup\widetilde{\mathcal{D}}^{(0)},\ldots,\mathcal{D}^{(M-1)}\sqcup\widetilde{\mathcal{D}}^{(M-1-v)};\mathcal{E},w\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}\sqcup\overline{\mathcal{D}}^{(0)},\ldots,\mathcal{D}^{(v-1)}\sqcup\overline{\mathcal{D}}^{(v-1)},\mathcal{D}^{(v)},\ldots,\mathcal{D}^{(M-1)};\mathcal{E},w\right]$
	Reduce-to-one	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A}\sqcup\mathcal{K}}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)},\widetilde{\mathcal{D}}^{(0)},\ldots,\widetilde{\mathcal{D}}^{(M-1-v)};\mathcal{E},w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)};\left(\mathcal{E}\sqcup\widetilde{\mathcal{D}}\right),\left(w+j\cdot\mathrm{prod}\left(\mathbf{P},\mathcal{E}\right)\right)\right]$

Figure 2.15: Redistributions associated with balanced collectives on processing mesh of size $\mathbf{P}$. Each redistribution is valid under a consistent permutation of entries in the input and output tensor distributions. Here, $(v-1)\in\mathcal{R}(M)$, $w\in\mathcal{R}(\mathrm{prod}(\mathbf{P},\mathcal{E}))$, and $j\in\mathcal{R}\left(\mathrm{prod}\left(\mathbf{P},\widetilde{\mathcal{D}}\right)\right)$.

Elemental	Collective used	Proposed
$(M_u, M_v) \leftrightarrow (M_u, *)$	$\leftarrow$ Reduce-scatter, $\rightarrow$ Allgather	$[(u), (v)] \leftrightarrow [(u), ()]$
$(M_u, M_v) \leftrightarrow (*, M_v)$	$\leftarrow$ Reduce-scatter, $\rightarrow$ Allgather	$[(u), (v)] \leftrightarrow [(), (v)]$
$(V_u, *) \leftrightarrow (M_u, *)$	$\leftarrow$ Reduce-scatter, $\rightarrow$ Allgather	$[(u, v), ()] \leftrightarrow [(u), ()]$
$(V_u, *) \leftrightarrow (M_u, M_v)$	$\leftrightarrow$ All-to-all	$[(u, v), ()] \leftrightarrow [(u), (v)]$
$(V_u, *) \leftrightarrow (V_v, *)$	$\leftrightarrow$ Permutation	$[(u, v), ()] \leftrightarrow [(v, u), ()]$
$(*, V_u) \leftrightarrow (*, V_v)$	$\leftrightarrow$ Permutation	$[(), (u, v)] \leftrightarrow [(), (v, u)]$
$(M_u, *) \leftrightarrow (*, *)$	$\leftarrow$ Reduce-scatter, $\rightarrow$ Allgather	$[(u), ()] \leftrightarrow [(), ()]$
$(*, M_u) \leftrightarrow (*, *)$	$\leftarrow$ Reduce-scatter, $\rightarrow$ Allgather	$[(), (u)] \leftrightarrow [(), ()]$
$(M_u, *) \leftrightarrow (*, M_u)$	$\leftrightarrow$ All-to-all	$[(u), ()] \leftrightarrow [(), (u)]$

Figure 2.16: Redistribution rules from Elemental in terms of defined notation. Each distribution notation is parameterized by a pair of variables u and v such that $u \neq v$ and $u, v \in \{0, 1, C, R\}$. When interpreted in the "Elemental" column, $u, v \in \{C, R\}$ and when interpreted in the "Proposed" column, $u, v \in \{0, 1\}$.

specified in the set

$$\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(m)}.$$

When we refer to a collective communicating or being performed over modes in $\widetilde{\mathcal{D}}$, we mean that simultaneous collective communications are performed and processes are involved in the same collective communication instance only if their location in $\mathbf{G}$ only differs in the modes specified by $\widetilde{\mathcal{D}}$. For instance, consider the redistribution

$$\mathbf{A}\left[(0), (1)\right] \leftarrow \mathbf{A}\left[(0, 2), (1, 3)\right]. \tag{2.10}$$

Rewriting (2.10) as

$$\mathbf{A}\left[(0), (1)\right] \leftarrow \mathbf{A}\left[(0) \sqcup (2), (1) \sqcup (3)\right]$$

and consulting Figure 2.14 and Figure 2.15 reveals that the redistribution corresponds to an allgather collective communicating over modes $\widetilde{\mathcal{D}} = (2, 3)$ (as $\widetilde{\mathcal{D}}^{(0)} = (2)$ and $\widetilde{\mathcal{D}}^{(1)} = (3)$).

Interestingly, and importantly, all redistributions affect the ends, or suffixes of tensor mode distributions. The reason for this becomes apparent when one considers the definition of how elements are distributed among processes in conjunction with the set of processing mesh modes over which communication occurs.

Other collectives have more flexible rules. In the case of the permutation collective, the final tensor mode distributions "shuffle" entries of the initial mode distributions. For example, the redistribution

$$\mathbf{A}\left[(2, 0)\,, (1)\right] \leftarrow \mathbf{A}\left[(0, 2)\,, (1)\right] \tag{2.11}$$

can be performed via a permutation collective communicating over the group of processing mesh modes in $(0, 2)$.

The absolute limit of flexibility in collectives comes with the all-to-all collective. This flexibility allows an all-to-all collective to implement all other collectives that do not perform a reduction of data being communicated (permutation, allgather, gather-to-one, scatter, and broadcast). For example, both (2.10) and (2.11) can be implemented as an all-to-all. Intuitively, this makes sense as an all-to-all collective can send some data to all processes involved in the communication; different output distributions are created depending on how the data is ordered when redistributed.

In practice it is typically better to use the more specialized collectives to perform these redistributions as the all-to-all collective required is typically “unbalanced”, and would result in a significantly higher cost in terms of bandwidth to implement the same redistribution. For instance, consider the case where we implement a permutation collective in terms of an all-to-all collective. Each process needs only to send data to one other process in the mesh and otherwise does not need to communicate any data with other processes. This pattern is highly unbalanced in terms of amount of actual data a single process needs to send to every other process. To implement this in terms of an all-to-all, one would need to send an set of “empty” data packets to every other process that would be ignored upon reception. The problem lies in the fact that each of these “empty” data packets are the same size as the actual data required to be exchanged.

2.6 Summary

Understanding how data is distributed on a processing mesh is the first step in designing efficient implementations for computations of interest. However, one must also understand how collective communications can be utilized to efficiently redistribute data among processes. When these interactions are well understood, one can design families of algorithms that achieve high-performance for computations. In our case, we see how one can design such algorithms for tensor contractions, as we will see next.

In this chapter, we introduced a notation for describing how data of an order-M tensor is distributed among processes of an order-N mesh based on an elemental-cyclic distribution. Additionally, we argued the utility of each collective depicted in Figure 2.6 and formalized each collective as redistributions in the notation.

Chapter 3

Algorithm Derivation

Now that we have an understanding of the relationship between data distributions and redistributions that can be cast in terms of collective communications, we can discuss how the ideas in the previous chapter can be leveraged to design algorithms for tensor computations with associated high-performance implementations. This work focuses on generalizing the family of algorithms for matrix-matrix multiplication, introduced earlier in Chapter 2, to tensor contractions of dense non-symmetric tensors.

We begin this chapter by first introducing the general structure of our derivation procedure followed by discussing several examples based on this approach to build an intuition. Subsequently, we formalize the procedure. We restrict the discussion in this chapter to binary contractions, although the ideas generalize to n-ary contractions.

3.1 Preliminaries

3.1.1 Approach

Consider the general form of a binary tensor contraction given by

$$\mathbf{C}^{\mathcal{C}} = \sum_{\mathcal{K}} \mathbf{A}^{\mathcal{A}} \mathbf{B}^{\mathcal{B}} + \mathbf{C}^{\mathcal{C}} \tag{3.1}$$

where $\mathcal{K} = \mathcal{A} \cap \mathcal{B}$ and the tensors $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are appropriately sized based on how modes of each tensor are paired. Let $\mathbf{I}^{(\mathbf{K})} = \left(I_0^{(\mathbf{K})}, \ldots, I_{|\mathcal{K}|-1}^{(\mathbf{K})} \right)$ represent the dimensions of the modes labeled $\mathcal{K}$. Then, each element of $\mathbf{C}$ is defined as

$$c_{\mathbf{i}^{(\mathbf{C})}} \mathrel{+}= \sum_{\mathbf{k} \in \mathcal{R}\left(\mathbf{I}^{(\mathbf{K})}\right)} a_{\mathbf{i}^{(\mathbf{A})}} \cdot b_{\mathbf{i}^{(\mathbf{B})}} \tag{3.2}$$

where $\mathbf{i}^{(\mathbf{A})}$ and $\mathbf{i}^{(\mathbf{B})}$ are consistently defined and therefore incorporate $\mathbf{k}$ in their definition.

We can refactor the computation of (3.2) into two steps, where we first form a set of temporaries each representing the result of a disjoint set of accumulations and then sum the results into the output object. The observation allows each process in our processing mesh to locally compute a portion of (3.2) and then collaborate with other processes to form results together via a global communication (if necessary). For consistency, the order of the introduced temporary, $\mathbf{T}$, must be the same as the output $\mathbf{C}$ *plus* the number of labels involved in the summation ($|\mathcal{K}|$). This stems from the fact that each process is computing independent portions of $\mathbf{T}$ that can be accumulated into the output $\mathbf{C}$. Considering this, we reformulate (3.1) as

$$\mathbf{T}^{\mathcal{C} \sqcup \mathcal{K}'} = \widehat{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A}} \mathbf{B}^{\mathcal{B}} \tag{3.3}$$

$$\mathbf{C}^{\mathcal{C}} = \widetilde{\sum_{\mathcal{K}'}} \mathbf{T}^{\mathcal{C} \sqcup \mathcal{K}'} + \mathbf{C}^{\mathcal{C}} \tag{3.4}$$

As mentioned, (3.3) does *not* represent a complete computation, only a *partial* computation. We introduce $\mathcal{K}'$ merely to indicate that the corresponding modes have only been partially contracted and therefore must be eliminated in (3.4) (the entries and corresponding order of $\mathcal{K}'$ are the same as in $\mathcal{K}$). The correct elementwise definition of (3.3) and (3.4) is given by

$$t_{\mathbf{i}^{(\mathbf{C})} \sqcup \mathbf{p}} \ = \sum_{\mathbf{k} \in \mathcal{P}_{\mathbf{p}}} a_{\mathbf{i}^{(\mathbf{A})}} \cdot b_{\mathbf{i}^{(\mathbf{B})}} \tag{3.5}$$

$$c_{\mathbf{i}^{(\mathbf{C})}} \mathrel{+}= \sum_{\mathbf{k} \in \mathcal{R}(\mathbf{P})} t_{\mathbf{i}^{(\mathbf{C})} \sqcup \mathbf{k}}, \tag{3.6}$$

where $\boldsymbol{\mathcal{P}}$ forms a partition of $\mathcal{R}\left(\mathbf{I}^{(\mathbf{K})}\right)$ into $\operatorname{prod}(\mathbf{P})$ sets (indexed by $\mathbf{p}$) and both $\mathbf{i}^{(\mathbf{A})}$ and $\mathbf{i}^{(\mathbf{B})}$ are appropriately defined. Using (3.5) and (3.6), the interpretations of (3.3) and (3.4) are clear: we form the partial contributions in (3.3) and then accumulate over the results in (3.4). Introducing $\mathbf{T}$ as a variable allows us to separate computations to be performed locally by each process from the summation that must be performed globally.

It should be noted that the refactoring chosen based on processes in a processing mesh represents only one choice of many; nothing prevents us from computing (3.1) in multiple steps instead of two as done here. Although there may be benefits to further decomposing the computation to expose additional parallelism, such as that exposed by so-called 3D algorithms for matrix-matrix multiplication [2, 41, 73, 79], in this work we focus on decomposing into two steps as we develop a general framework. Projects such as the Cyclops Tensor Framework (CTF) [81] and the RRR [70] already leverage versions of these algorithms. Extending our framework to also incorporate these insights is left as future work and is out of the scope of this document.

3.1.2 Distributed Template

To create an algorithm template for computing a binary tensor contraction based on the ideas introduced in the previous subsection, we must introduce the notion of data distributions into the discussion, transforming (3.3) and (3.4) into

$$\begin{aligned}
\mathbf{T}^{\mathcal{C}\sqcup\mathcal{K}'}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] &= \widehat{\sum_{\mathcal{K}}}\mathbf{A}^{\mathcal{A}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right]\mathbf{B}^{\mathcal{B}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] && (3.7)\\
\mathbf{C}^{\mathcal{C}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] &= \widetilde{\sum_{\mathcal{K}'}}\mathbf{T}^{\mathcal{C}\sqcup\mathcal{K}'}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{T})}\right] + \mathbf{C}^{\mathcal{C}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right]. && (3.8)
\end{aligned}$$

where $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}$, $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}$, $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$, and $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$ represent the distributions of the associated tensor at the time local computation is performed. Recall that the elementwise definition of $\mathbf{T}$ in (3.5) and (3.6) is based on a partition of the computation to be performed. The most natural partition to use is the one that assigns each process all computations that can be performed locally at this point in the algorithm. Incorporating this in (3.7) and (3.8) leads to the elementwise definition given by

$$\begin{aligned}
t_{\mathbf{i}^{(\mathbf{C})}\sqcup\mathbf{p}} &= \sum_{\mathbf{k}\in\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}^{(\mathbf{K})}\right)} a_{\mathbf{i}^{(\mathbf{A})}}\cdot b_{\mathbf{i}^{(\mathbf{B})}} && (3.9)\\
c_{\mathbf{i}^{(\mathbf{C})}} &\mathrel{+}= \sum_{\mathbf{k}\in\mathcal{R}\left(\mathbf{P}\left(\boldsymbol{\mathcal{D}}^{(\mathbf{K})}\right)\right)} t_{\mathbf{i}^{(\mathbf{C})}\sqcup\mathbf{k}}, && (3.10)
\end{aligned}$$

where $\boldsymbol{\mathcal{D}}^{(\mathbf{K})}$ represents the tensor mode distributions of modes paired by $\mathcal{K}$. The usage of $\mathbf{P}\left(\boldsymbol{\mathcal{D}}^{(\mathbf{K})}\right)$ in (3.10) ensures correctness for arbitrary distributions of data (such as when a replication of data is involved). We stress here that $\mathbf{T}$ in (3.9) is computed only via local computations. Additionally, (3.10) is performed *only* by global reductions.

To ensure the generality of the derivation procedure, we cannot assume that the incoming distributions of $\mathbf{A}$, $\mathbf{B}$ are such that the local computation can be performed without communication. Futher, we cannot assume that the distribution of $\mathbf{T}$ used for local computation is such that the global reduction in (3.8) can be performed without additional redistributions. This potentially requires redistribution of $\mathbf{C}$ both before and after the global reduction as well as redistribution of $\mathbf{A}$ and $\mathbf{B}$ to ensure the local computation proceeds correctly. With this in mind, our goal becomes to show how one can systematically derive stationary algorithms based on the template

$$
\begin{array}{lllll}
1. & \mathbf{A}^{\mathcal{A}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] & \leftarrow & \mathbf{A}^{\mathcal{A}}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{A})}\right] & \text{(redistribute } \mathbf{A}\text{)} \\
2. & \mathbf{B}^{\mathcal{B}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \leftarrow & \mathbf{B}^{\mathcal{B}}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{B})}\right] & \text{(redistribute } \mathbf{B}\text{)} \\
3. & \mathbf{C}^{\mathcal{C}}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\mathcal{C}}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{C})}\right] & \text{(redistribute } \mathbf{C}\text{)} \\
4. & \mathbf{T}^{\mathcal{C}\sqcup\mathcal{K}'}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] & = & \widehat{\sum_{\mathcal{K}}}\mathbf{A}^{\mathcal{A}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right]\mathbf{B}^{\mathcal{B}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \text{(local computation)} \\
5. & \mathbf{C}^{\mathcal{C}}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & += & \widetilde{\sum_{\mathcal{K}'}}\mathbf{T}^{\mathcal{C}\sqcup\mathcal{K}'}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] & \text{(global reduction)} \\
6. & \mathbf{C}^{\mathcal{C}}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\mathcal{C}}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right]. & \text{(redistribute } \mathbf{C}\text{)}
\end{array}
$$

Here, we have introduced initial distributions for each tensor operand of the problem specification and denoted them $\boldsymbol{\mathcal{D}}^{(\mathbf{A})}$, $\boldsymbol{\mathcal{D}}^{(\mathbf{B})}$, and $\boldsymbol{\mathcal{D}}^{(\mathbf{C})}$. As $\mathbf{T}$ was not given in the original problem specification, we are free to choose its distribution $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$. We require the introduction of $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ to ensure the global reduction succeeds and the introduction of $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ to ensure the postcondition of our to be developed algorithm is achieved.

With this template set, our goal now becomes to systematically determine the values for $\boldsymbol{\mathcal{D}}^{(\mathbf{A})}$, $\boldsymbol{\mathcal{D}}^{(\mathbf{B})}$, $\boldsymbol{\mathcal{D}}^{(\mathbf{C})}$, $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$ (initial tensor distributions), $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}$, $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}$, $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ (final tensor distributions), and $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ (intermediate distribution). We will see that depending on the algorithmic variant being derived, some of these steps become greatly simplified and some of these unknowns become predetermined. We now provide examples of how the derivation procedure proceeds for both the stationary C and stationary A algorithmic variants before we formalize the procedure.

3.2 Example: Stationary C Algorithms

Consider the tensor contraction

$$\mathbf{C}^{\alpha\beta\eta\iota} = \mathbf{A}^{\alpha\gamma\iota\kappa}\mathbf{B}^{\beta\gamma\eta\kappa} + \mathbf{C}^{\alpha\beta\eta\iota}$$

where **C**, **A** and **B** are conformally sized. For this example, our goal is to derive an algorithm that computes the above expression without communicating **C**, yielding a stationary C algorithm. Assume we are performing this computation on an order-4 processing mesh **G** of size $\mathbf{P} = (P_0, P_1, P_2, P_3)$.

3.2.1 Derivation

At this point, we know that $\mathcal{A} = (\alpha, \gamma, \iota, \kappa)$, $\mathcal{B} = (\beta, \gamma, \eta, \kappa)$, $\mathcal{C} = (\alpha, \beta, \eta, \iota)$, and $\mathcal{K} = (\gamma, \kappa)$ giving us the following partially filled template

$$\begin{array}{lllcl}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] & \leftarrow & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{A})}\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{B})}\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{C})}\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] & = & \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right].
\end{array} \tag{3.11}$$

The derivation procedure is not dependent on a choice of initial distribution, however, as stated in Section 2.3, we know that tensor distributions that involve all modes of the processing mesh do not implicitly replicate data. Let us derive an algorithm that assumes each tensor is initially distributed such that there is no replication of data among processes. A convenient form for this is to assume that the incoming tensors **A**, **B**, and **C** are initially distributed as

$$\mathbf{A}\left[(0),(1),(2),(3)\right], \mathbf{B}\left[(0),(1),(2),(3)\right], \text{ and } \mathbf{C}\left[(0),(1),(2),(3)\right],$$

in other words, via an elemental-cyclic distribution. By incorporating this into

(3.11), our template now becomes

$$\begin{array}{rlcl}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] & \leftarrow & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] & = & \widehat{\sum_{\gamma\kappa}}\mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right]\mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}}\mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right].
\end{array} \tag{3.12}$$

As we are assuming we are deriving a stationary C algorithm, we must not communicate $\mathbf{C}$ throughout the computation. Thus $\mathbf{C}$ must not be redistributed, meaning $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \boldsymbol{\mathcal{D}}^{(\mathbf{C})}$. The template becomes

$$\begin{array}{rlcl}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] & \leftarrow & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right] & = & \widehat{\sum_{\gamma\kappa}}\mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right]\mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}}\mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})}\right].
\end{array} \tag{3.13}$$

We eliminated Steps 3 and 6 from (3.12) since $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \boldsymbol{\mathcal{D}}^{(\mathbf{C})}$.

At this point, we are left with an underconstrained problem. We have no information available that uniquely specifies any remaining unknowns. Thus we can make choices that impact the overall cost of our developed algorithm. Since Step 5 in (3.13) requires a global communication, which represents overhead, we try to find a solution that eliminates this step. By distributing the paired modes of $\mathbf{T}$ and $\mathbf{C}$ similarly, we ensure that we do no need to communicate over the processing mesh modes used in these tensor mode distributions. With this information, we arrive at

$$\begin{array}{lllll}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right] & \leftarrow & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),?,?\right] & = & \widehat{\sum_{\gamma\kappa}}\mathbf{A}^{\alpha\gamma\iota\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})}\right]\mathbf{B}^{\beta\gamma\eta\kappa}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})}\right] \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}}\mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),?,?\right]
\end{array} \tag{3.14}$$

where "?" represents an unknown quantity. We know that Step 4 in (3.14) corresponds to a local tensor contraction and therefore all paired modes must be distributed similarly to ensure the local computation succeeds. Propagating this information leads to the template

$$\begin{array}{lllll}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),?,(3),?\right] & \leftarrow & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),?,(2),?\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),?,?\right] & & \\
 & \quad = \widehat{\sum_{\gamma\kappa}}\mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),?,(3),?\right]\mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),?,(2),?\right] & & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}}\mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),?,?\right].
\end{array} \tag{3.15}$$

Now, recall that no entry of a tensor mode distribution may be reused in the same tensor distribution. Considering the determined tensor mode distributions along with the fact that we have assumed an order-4 processing mesh reveals that the only valid assignment for the remaining tensor mode distributions is the empty set (indicating a duplication of the tensor-mode indices assigned to processes). Propagating this information in (3.15) leads to the template

$$\begin{array}{lll}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(),(3),()\right] & \leftarrow \quad \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),(),(2),()\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),(),()\right] & \\
 & \qquad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(),(3),()\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),(),(2),()\right] & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] & += \quad \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(1),(2),(3),(),()\right].
\end{array} \tag{3.16}$$

At this point, we have defined all unknowns and have arrived at a valid algorithm. However, notice that Step 5 in (3.16) is performing a global reduction over no processing mesh modes and is here an assignment rather than a summation. Therefore, by replacing **C** with **T** in Step 4, we can remove Step 5 from our derived algorithm. This leads us to our final template given by

$$\begin{array}{lll}
1. & \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(),(3),()\right] & \leftarrow \quad \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \\
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),(),(2),()\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
4. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] & += \quad \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(),(3),()\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[(1),(),(2),()\right].
\end{array} \tag{3.17}$$

We can summarize the actions of a process in **G** for implementing the algorithm derived in this section as follows:

1. Redistribute **A**

2. Redistribute **B**

3. Locally compute $c_{\mathbf{i}^{(\mathbf{C})}} \mathrel{+}= \sum_{\gamma\kappa} a_{\mathbf{i}^{(\mathbf{A})}} \cdot b_{\mathbf{i}^{(\mathbf{B})}}$

where $\mathbf{i}^{(\mathbf{A})}$ and $\mathbf{i}^{(\mathbf{B})}$ are appropriately defined. An examination of (3.17) reveals that none of the required redistributions are directly implemented by the rules given in Figure 2.14 and Figure 2.15. This is where expert knowledge is still required to determine the optimal path of redistributions (minimal cost or storage). However, the derivation procedure has significantly reduced the number of possibilities that need to be considered.

3.2.2 Blocking

In Step 4 of (3.17), the distributions of **A** and **B** indicate a replication of data among processes as some modes of **G** are not used in the associated tensor distribution. As we increase the number of processes involved, so does the amount of data replication, with associated memory requirements for the duplicated data. One straightforward approach to curb this effect is to block the overall tensor contraction into a series of smaller *block* tensor contractions to which we can apply the same derivation process.

Depending on how we choose the size of each block, different characteristics of the execution will be observed. If a large block size is chosen, a large amount of data is required to perform the computation. If a small block size is chosen, reducing the amount of storage required, a higher communication cost is predicted due to the increased number of communications being performed (one round for each block). It is this trade-off associated with the block size parameter that we must recognize and appropriately analyze to ensure an efficient implementation is created.

Since we are dealing with higher-order tensors, determining the correct modes to block may seem daunting at first. Observe that the source of replicated data in (3.17) are the distributed tensors $\mathbf{A}\left[(0),(),(3),()\right]$ and $\mathbf{B}\left[(1),(),(2),()\right]$. Comparing the distributions of these tensors to the initial distributions, we see that the replication of data originates from the redistribution of the tensor modes involved in the summation. If we increase the number of processes in our processing mesh, then replication will occur due to the redistribution of these tensor modes. By blocking along these modes, we can mitigate this effect.

Generalizing this observation, notice that the tensor modes we should block along in (3.17) correspond to the modes that are unpaired with our stationary tensor $\mathbf{C}$. In fact, for all stationary variants, the observation that replication occurs due to the redistribution of tensor modes not paired with the stationary operand and should be blocked holds. This reasoning provides an expert with a simple way of determining along which tensor modes to introduce blocking, thereby mitigating the increased storage effect as the size of the processing mesh increases.

3.2.3 Observations

It may seem convenient that the remaining tensor mode distributions of $\mathbf{A}$ and $\mathbf{B}$ were “forced” into a particular definition by assuming we were computing on an order-4 grid. As discussed previously, if we arrive at a state where the subsequent step is ambiguous, a choice must be made that leads to different algorithms with different performance characteristics. This concern is equivalent to that of having assumed an initial distribution. This was a heuristic, but is not necessarily the best for every problem specification, especially when more than a binary contraction is encountered.

Our goal in this work is to be able to formalize the derivation procedure to such a degree that making these choices requires less effort by the (human or mechanical) expert. By having such a systematic procedure for arriving at a valid algorithm along with a well-defined notation, an expert can instantly detect along which modes communication must occur. If the expert decides that a different set of assumptions should be used, the same rote procedure can be applied to arrive at an algorithm that is better suited under new sets of assumptions.

We now perform the same derivation for stationary A variants before generalizing the procedure.

3.3 Example: Stationary A Algorithms

We again consider the tensor contraction

$$\mathbf{C}^{\alpha\beta\eta\iota} = \mathbf{A}^{\alpha\gamma\iota\kappa}\mathbf{B}^{\beta\gamma\eta\kappa} + \mathbf{C}^{\alpha\beta\eta\iota}$$

only now we assume that we are deriving an algorithm that does not communicate **A**.

3.3.1 Derivation

Assuming the same information regarding processing mesh configuration, initial distributions, and the new information that we are deriving the stationary A variant, we arrive at the following partially filled algorithm template

$$\begin{array}{lll}
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(),(1),(),(3)\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0),(1),(2),(3)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[(0),(1),(2),(3)\right] \\
4. & \multicolumn{2}{l}{\mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(),(),(2),(1),(3)\right]} \\
 & \multicolumn{2}{l}{\qquad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0),(1),(2),(3)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[(),(1),(),(3)\right]} \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \mathrel{+}= \quad \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0),(),(),(2),(1),(3)\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right].
\end{array} \tag{3.18}$$

At this point, there are no further constraints we can apply to determine the definition of $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ or $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. For $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$, we cannot assume that paired modes must be distributed similarly as this distribution is used in a global reduction (not a local tensor contraction). Further, we have no information available to constrain $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. This is a case where additional information or heuristics are required and performance characteristics may vary depending on the choice made.

A common choice of $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ is based on the assumption that the algorithm being derived will be implemented in a loop structure; i.e., this is part of a blocked computation. In this case, it is useful to have $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \boldsymbol{\mathcal{D}}^{(\mathbf{C})}$ as this eliminates one source of redistribution between loop iterations. We adopt this heuristic for this example. Once again, this is a choice made for example purposes only. Propagating this information into (3.18) results in the following template

$$
\begin{array}{llll}
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[() , (1) , () , (3)\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0) , (1) , (2) , (3)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0) , (1) , (2) , (3)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0) , () , () , (2) , (1) , (3)\right] & & \\
 & \quad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0) , (1) , (2) , (3)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[() , (1) , () , (3)\right] & & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right] & \mathrel{+}= & \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0) , () , () , (2) , (1) , (3)\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0) , (1) , (2) , (3)\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}\right].
\end{array}
\tag{3.19}
$$

We are left with determining how to set $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. Preferably, we could find a way to set $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ so that the redistribution in Step 5 of (3.19) becomes a no-op. For this to occur, we would need the global reduction in Step 5 to be properly defined so that the redistribution could directly update $\mathbf{C}$ distributed as $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. After consulting Figure 2.15, we can conclude that there is no single rule that directly implements this redistribution. Therefore, we can only hope to mitigate the impact due to the redistributions in Step 3 and Step 6 in (3.19).

One approach we can take to achieve this is to examine the tensor mode distributions of $\mathbf{T}$ under $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$ and compare them to the tensor mode distributions of $\mathbf{C}$ under both $\boldsymbol{\mathcal{D}}^{(\mathbf{C})}$ and $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. If any paired tensor modes are distributed similarly in all three of these distributions, then by setting the corresponding tensor mode distribution $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ similarly, we avoid communication over those modes of the processing mesh. We see that we can apply this reasoning to mode 0 of $\mathbf{C}$. Using this knowledge produces the following template

$$\begin{array}{lll}
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,?\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] & \\
 & \quad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,?\right] & \mathrel{+}= \quad \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,?\right].
\end{array} \tag{3.20}$$

A similar argument can be made for any tensor mode distributions of $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$ that are not involved in the global reduction and that do not conflict with $\boldsymbol{\mathcal{D}}^{(\mathbf{C})}$ and $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$. Assigning the paired tensor modes of $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ similarly ensures we do not perform redistribution over these modes of the processing mesh. We see that we can apply this reasoning to mode 3 of $\mathbf{C}$. Applying this argument to (3.20) leads to the following template

$$\begin{array}{lll}
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,\left(2\right)\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] & \\
 & \quad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,\left(2\right)\right] & \mathrel{+}= \quad \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),?,?,\left(2\right)\right].
\end{array} \tag{3.21}$$

The last remaining piece we can exploit relies on the rule associated with the global reduction. If there exists a formulation of the redistribution in Step 5 that can reduce the communication required in Step 3 and Step 6, then we should structure the redistribution to enable this. For this, we must examing both the rule associated with the global reduction as well as the tensor mode distributions specifying what is communicated over. In doing so, we see that by setting the entry in mode 1 of $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ to (1), then we have reduced the number of unknowns in $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ while also avoiding communication in Step 3 and Step 6. Applying this to (3.21) leads to

$$
\begin{array}{lll}
2. \;\; \mathbf{B}^{\beta\gamma\eta\kappa}\left[(), (1), (), (3)\right] & \leftarrow & \mathbf{B}^{\beta\gamma\eta\kappa}\left[(0), (1), (2), (3)\right] \\
3. \;\; \mathbf{C}^{\alpha\beta\eta\iota}\left[(0), (1), ?, (2)\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0), (1), (2), (3)\right] \\
4. \;\; \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0), (), (), (2), (1), (3)\right] & & \\
\qquad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[(0), (1), (2), (3)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[(), (1), (), (3)\right] & & \\
5. \;\; \mathbf{C}^{\alpha\beta\eta\iota}\left[(0), (1), ?, (2)\right] & += & \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[(0), (), (), (2), (1), (3)\right] \\
6. \;\; \mathbf{C}^{\alpha\beta\eta\iota}\left[(0), (1), (2), (3)\right] & \leftarrow & \mathbf{C}^{\alpha\beta\eta\iota}\left[(0), (1), ?, (2)\right].
\end{array}
\tag{3.22}
$$

Once again, we are left with an underconstrained problem. Fortunately, in this case, we can determine the remaining unknowns in (3.22) as there is only one option. However, we may not be able to do this in general. For now though, let us proceed and arrive at our final derived algorithm

$$\begin{array}{lll}
2. & \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \leftarrow \quad \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
3. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(3\right),\left(2\right)\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \\
4. & \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] & \\
 & \quad = \widehat{\sum_{\gamma\kappa}} \mathbf{A}^{\alpha\gamma\iota\kappa}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] \mathbf{B}^{\beta\gamma\eta\kappa}\left[\left(\right),\left(1\right),\left(\right),\left(3\right)\right] & \\
5. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(3\right),\left(2\right)\right] & \mathrel{+}= \quad \widetilde{\sum_{\gamma\kappa}} \mathbf{T}^{\alpha\beta\eta\iota\gamma\kappa}\left[\left(0\right),\left(\right),\left(\right),\left(2\right),\left(1\right),\left(3\right)\right] \\
6. & \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(2\right),\left(3\right)\right] & \leftarrow \quad \mathbf{C}^{\alpha\beta\eta\iota}\left[\left(0\right),\left(1\right),\left(3\right),\left(2\right)\right].
\end{array}$$

3.3.2 Observations

Before we end this section, we once again acknowledge the inherent trade-off between space and computational complexity in the developed algorithm (when viewed as implementing a block operation) that must analyzed for a given problem specification.

3.4 A Systematic Procedure for Deriving Stationary Algorithms

We now formalize the derivation procedure of all stationary algorithms for an arbitrary tensor contraction performed on an arbitrary-order processing mesh. Consider the order-M tensor $\mathbf{C}$ of size $\mathbf{I}^{(\mathbf{C})}$ and the binary tensor contraction

$$\mathbf{C}^{\mathcal{C}} = \mathbf{A}^{\mathcal{A}}\mathbf{B}^{\mathcal{B}} + \mathbf{C}^{\mathcal{C}}$$

defined elementwise as

$$\begin{aligned} t_{\mathbf{i}^{(\mathbf{C})} \sqcup \mathbf{p}} &= \sum_{\mathbf{k} \in \mathcal{P}_{\mathbf{p}}} a_{\mathbf{i}^{(\mathbf{A})}} \cdot b_{\mathbf{i}^{(\mathbf{B})}} \\ c_{\mathbf{i}^{(\mathbf{C})}} &\mathrel{+}= \sum_{\mathbf{k} \in \mathcal{R}(\mathbf{P})} t_{\mathbf{i}^{(\mathbf{C})} \sqcup \mathbf{k}} \end{aligned}$$

where the multiindices $\mathbf{i}^{(\mathbf{A})}$ and $\mathbf{i}^{(\mathbf{B})}$, as well as the partition $\boldsymbol{\mathcal{P}}$ are appropriately defined. Let $\mathcal{K} = \mathcal{A} \cap \mathcal{B}$ and compute $\mathbf{C}$ based on the template

$$\begin{array}{rlcl}
1. & \mathbf{A}^{\mathcal{A}} \left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})} \right] & \leftarrow & \mathbf{A}^{\mathcal{A}} \left[\boldsymbol{\mathcal{D}}^{(\mathbf{A})} \right] \\
2. & \mathbf{B}^{\mathcal{B}} \left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})} \right] & \leftarrow & \mathbf{B}^{\mathcal{B}} \left[\boldsymbol{\mathcal{D}}^{(\mathbf{B})} \right] \\
3. & \mathbf{C}^{\mathcal{C}} \left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} \right] & \leftarrow & \mathbf{C}^{\mathcal{C}} \left[\boldsymbol{\mathcal{D}}^{(\mathbf{C})} \right] \\
4. & \mathbf{T}^{\mathcal{C} \sqcup \mathcal{K}'} \left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})} \right] & = & \widehat{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A}} \left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{A})} \right] \mathbf{B}^{\mathcal{B}} \left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{B})} \right] \\
5. & \mathbf{C}^{\mathcal{C}} \left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} \right] & \mathrel{+}= & \widetilde{\sum_{\mathcal{K}'}} \mathbf{T}^{\mathcal{C} \sqcup \mathcal{K}'} \left[\boldsymbol{\mathcal{D}}^{(\mathbf{T})} \right] \\
6. & \mathbf{C}^{\mathcal{C}} \left[\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} \right] & \leftarrow & \mathbf{C}^{\mathcal{C}} \left[\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} \right].
\end{array}$$

Choose a tensor to remain stationary during the computation and let $\mathbf{S}$ represent that tensor. Then, perform the following steps:

1. Set $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{S})} = \boldsymbol{\mathcal{D}}^{(\mathbf{S})}$.

   (a) If $\mathbf{S} = \mathbf{C}$ then set $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})} = \boldsymbol{\mathcal{D}}^{(\mathbf{C})}$.

2. Set all modes of $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$ paired to modes in $\boldsymbol{\mathcal{D}}^{(\mathbf{S})}$ similarly.

3. Enforce that all paired modes in Step 3 are distributed similarly.

4. Resolve remaining ambiguities with other knowledge. This includes:

(a) If $\mathbf{S} \neq \mathbf{C}$ then set all nonconflicting entries of $\widetilde{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$ based on $\boldsymbol{\mathcal{D}}^{(\mathbf{C})}$, $\overline{\boldsymbol{\mathcal{D}}}^{(\mathbf{C})}$, and $\boldsymbol{\mathcal{D}}^{(\mathbf{T})}$.

5. Remove any unnecessary steps.

The result of this procedure is an algorithm that performs the desired computation. A post-processing step that blocks the derived algorithm can then be applied to reduce the storage requirement of the resulting computations (at the expense of additional latency in the computation). In Section 3.2.2, we discuss how to select the correct modes to block.

3.5 Summary

In this chapter, we have shown how the notation proposed in Chapter 2 facilitates goal-oriented programming [24] for developing parallel algorithms of the tensor contraction operation. The direct benefit is that (human or mechanical) experts, tasked with implementing applications, can significantly reduce the time devoted to optimizing the developed implementations.

One additional benefit of the developed systematic derivation procedure is that optimizing a series of tensor contractions becomes a (relatively) straightforward extension of the ideas discussed in this chapter. In particular, recall that we do not require the initial or final distributions of any object to be specifically set; we only chose convenient distributions for discussion purposes. As such, optimizing a series of tensor contractions can be performed by optimizing each contraction separately and then optimizing the communications that occur between contractions. Performing this kind of optimization is facilitated by a well-defined notation.

Chapter 4

Optimizing Data Movement

In the previous chapter, we discussed how to derive parallel algorithms for the tensor contraction operation involving dense non-symmetric tensors. One result of this procedure is that the required data distributions for each step are specified. In effect, this specifies where data should be moved at each step, but does not specify *how* the data should be moved to *efficiently* redistribute the data. We would like a method that systematically decomposes an arbitrary specification of a redistribution into a series of redistributions that rely on balanced communications as balanced communications can more easily utilize the underlying communication network effectively.

In addition to this data movement that occurs globally among processes, there are data movements that occur within processes. Specifically, because we are implementing local tensor contractions in terms of matrix-matrix multiplication, there is data movement that occurs not only from the beginning or ending of a redistribution ("packing" and "unpacking"), but that also must be performed in order to correctly permute the operands to the local matrix-matrix multiplication.

In this chapter, we discuss two transformations that decompose a general redistribution of data into a series of balanced redistributions. We then discuss how local data movements can be consolidated to reduce the overhead associated with local computations.

4.1 Global Data Movement

Consider the redistribution

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right] \leftarrow \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right] \tag{4.1}$$

of an order-M tensor $\mathbf{A}$ distributed on an order-N processing mesh $\mathbf{G}$. In general, (4.1) corresponds to an unbalanced communication. Here, we discuss how to transform (4.1) into a series of balanced redistributions.

We begin with a motivating example that demonstrates the issues as well as an example of the developed transformation. We then restate the pertinent information regarding redistributions. Having done this, we then formalize the transformation used to convert (4.1) into a series of balanced redistributions. Finally, we detail an optimization to the transformation that exploits structure of the processing mesh to reduce the associated cost.

4.1.1 Motivating Example

Consider the order-2 tensor (matrix) $\mathbf{A}$ of size $\mathbf{I}$ distributed on the order-2 processing grid $\mathbf{G}$ of size $\mathbf{P} = (P_0, 2P_0)$ and the redistribution

$$\mathbf{A}\left[(1)\,,(0)\right] \leftarrow \mathbf{A}\left[(0)\,,(1)\right]$$

which simply transposes the matrix on the processing mesh. Let $\boldsymbol{\mathcal{D}} = \left[(0)\,,(1)\right]$, $\overline{\boldsymbol{\mathcal{D}}} = \left[(1)\,,(0)\right]$, and $\hat{p} = 2P_0^2$. Illustrations of these distributions are given in Figure 4.1.

Notice that in Figure 4.1, not all processes contribute data equally to every other process. For instance, the process $\mathbf{p} = (0, 0)$ does not require entries from processes at locations $(0, 1)$, $(1, 1)$, $(0, 3)$, or $(1, 3)$, but obtains half of its required data under $\overline{\boldsymbol{\mathcal{D}}}$ from both processes at locations $(0, 2)$ and $(1, 0)$. In this example, each process could obtain the required data from processes in the same row (mode 1) but may, in general, require data from processes in the same column (mode 0) of $\mathbf{G}$. This can be seen if the redistribution is performed on a processing mesh of size $(6, 8)$. As stated, this redistribution corresponds to an unbalanced communication if implemented with an all-to-all collective.

$\mathbf{p}=(0,0)$ $\begin{matrix} a_{0,0} & a_{0,2} & \cdots \\ a_{4,0} & a_{4,2} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,1)$ $\begin{matrix} a_{1,0} & a_{1,2} & \cdots \\ a_{5,0} & a_{5,2} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,2)$ $\begin{matrix} a_{2,0} & a_{2,2} & \cdots \\ a_{6,0} & a_{6,2} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,3)$ $\begin{matrix} a_{3,0} & a_{3,2} & \cdots \\ a_{7,0} & a_{7,2} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
$\mathbf{p}=(1,0)$ $\begin{matrix} a_{0,1} & a_{0,3} & \cdots \\ a_{4,1} & a_{4,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,1)$ $\begin{matrix} a_{1,1} & a_{1,3} & \cdots \\ a_{5,1} & a_{5,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,2)$ $\begin{matrix} a_{2,1} & a_{2,3} & \cdots \\ a_{6,1} & a_{6,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,3)$ $\begin{matrix} a_{3,1} & a_{3,3} & \cdots \\ a_{7,1} & a_{7,3} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$

(a) $\mathbf{A}\left[(1),(0)\right]$

$\mathbf{p}=(0,0)$ $\begin{matrix} a_{0,0} & a_{0,4} & \cdots \\ a_{2,0} & a_{2,4} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,1)$ $\begin{matrix} a_{0,1} & a_{0,5} & \cdots \\ a_{2,1} & a_{2,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,2)$ $\begin{matrix} a_{0,2} & a_{0,6} & \cdots \\ a_{2,2} & a_{2,6} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(0,3)$ $\begin{matrix} a_{0,3} & a_{0,7} & \cdots \\ a_{2,3} & a_{2,7} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
$\mathbf{p}=(1,0)$ $\begin{matrix} a_{1,0} & a_{1,4} & \cdots \\ a_{3,0} & a_{3,4} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,1)$ $\begin{matrix} a_{1,1} & a_{1,5} & \cdots \\ a_{3,1} & a_{3,5} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,2)$ $\begin{matrix} a_{1,2} & a_{1,6} & \cdots \\ a_{3,2} & a_{3,6} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$	$\mathbf{p}=(1,3)$ $\begin{matrix} a_{1,3} & a_{1,7} & \cdots \\ a_{3,3} & a_{3,7} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$

(b) $\mathbf{A}\left[(0),(1)\right]$

Figure 4.1: Graphical depiction of the matrix $\mathbf{A}$ distributed on a processing mesh of size $\mathbf{P}=(2,4)$ according to different tensor distributions. The top-left entry of every container corresponds to the process's location within the mesh.

Instead of directly implementing the redistribution based on unbalanced all-to-all collectives, let us analyze the following series of redistributions that implement the same overall redistribution:

$$\begin{array}{l} \mathbf{A}\left[(0)\,,(1)\right] \\ \qquad \downarrow\ \text{1. All-to-all over mode 1} \\ \mathbf{A}\left[(0,1)\,,()\right] \\ \qquad \downarrow\ \text{2. Permutation over modes 0 and 1} \\ \mathbf{A}\left[(1,0)\,,()\right] \\ \qquad \downarrow\ \text{3. All-to-all over mode 0} \\ \mathbf{A}\left[(1)\,,(0)\right]. \end{array}$$

Using the costs given in Figure 2.7 and recognizing that now the individual collectives are balanced, the above series of redistributions have an associated cost of

$$\begin{array}{rl} & \log_2(2P_0)\alpha + (2P_0 - 1)\dfrac{\overline{n}}{2P_0}\beta \\ + & \alpha + \overline{n}\beta \\ + & \log_2(P_0)\alpha + (P_0 - 1)\dfrac{\overline{n}}{P_0}\beta \\ = & (1 + \log_2\left(2P_0^2\right))\alpha + \left(\dfrac{(2P_0 - 1)}{2P_0} + \dfrac{(P_0 - 1)}{P_0} + 1\right)\overline{n}\beta \\ \approx & \log_2(\hat{p})\alpha + 3\overline{n}\beta, \end{array}$$

which is within a factor 3 of the cost associated with a balanced all-to-all redistribution over $\hat{p}$ processes.

In the following subsections, we generalize the technique used for the above example and formalize the transformation to apply to the more general redistribution in (4.1).

4.1.2 Preliminaries

A redistribution is a transformation on a tensor distributions that can perform the following transformations: reorder entries of tensor mode distributions and transfer entries of a tensor mode distributions to other tensor mode distributions. For instance, consider the redistribution

$$\mathbf{A}\left[(2,0),(1,3)\right] \leftarrow \mathbf{A}\left[(3,1,2),(0)\right].$$

This redistribution transfers the entries, or processing mesh modes, "1" and "3" from the mode-0 distribution to the mode-1 distribution and exchanges their order. Additionally, this redistribution transfers the entry "0" to the mode-0 distribution and changes the relative ordering of the entries "1", "2", and "3".

In the following two subsections, we discuss how to implement redistributions of the form

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right] \leftarrow \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right]$$

as a series of balanced all-to-all and permutation redistributions. We relate the effect on data distribution of each balanced redistribution using this interpretation of redistributions.

For reference, from Figure 2.14, the rule associated with a balanced permutation redistributions communicating over modes $\widetilde{\mathcal{D}} = \bigcup\limits_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup\limits_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}$ is of the form

$$\begin{gathered}\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right] \\ \downarrow \\ \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right]\end{gathered}$$

where we require that $\text{prod}\left(\widetilde{\mathcal{D}}^{(u)}, \mathbf{P}\right) = \text{prod}\left(\overline{\mathcal{D}}^{(u)}, \mathbf{P}\right)$. In other words, we require that the number of processes used to distribute along modes $\widetilde{\mathcal{D}}^{(u)}$ is the same as those used to distribute along modes $\overline{\mathcal{D}}^{(u)}$. The constraint that $\bigcup\limits_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup\limits_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}$ ensures that all entries that exist before the redistribution also exist after the redistribution. Using Figure 2.7, the associated cost with this redistribution is

$$\alpha + \overline{n}\beta,$$

where $\overline{n}$ corresponds to the total number of elements assigned to a process under the output distribution.

From Figure 2.15, the rule associated with balanced all-to-all redistributions is of the form

$$\begin{gathered}\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \mathcal{D}^{(v)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1-v)}\right] \\ \downarrow \\ \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)}, \mathcal{D}^{(v)}, \ldots, \mathcal{D}^{(M-1)}\right],\end{gathered} \tag{4.2}$$

where $(v-1) \in \mathcal{R}(M)$ and we assume the entries in both the initial and final tensor distributions have been consistently permuted.

Let us discuss the interpretation of (4.2). First, we see that we have divided the tensor modes $u \in \mathcal{R}(M)$ into two sets: the first v modes, and the last $M - v$

modes. The redistribution in (4.2) proceeds by transferring all modes in $\widetilde{\mathcal{D}}$ from their respective tensor mode distributions to the end of the first v tensor mode distributions via some ordering. As an example redistributions of the form

$$\begin{array}{lcll} \mathbf{A}\left[(0,2,3),(),()\right] & \leftarrow & \mathbf{A}\left[(0),(2),(3)\right] & (v=1) \\ \mathbf{A}\left[(0,3),(1,2),()\right] & \leftarrow & \mathbf{A}\left[(0),(1),(3,2)\right] & (v=2) \\ \mathbf{A}\left[(0,2),(1,3),(4)\right] & \leftarrow & \mathbf{A}\left[(0),(1),(4,3,2)\right] & (v=2) \end{array}$$

that each transfer entries corresponding to mode 2 and mode 3 are all valid redistributions according to (4.2).

Using Figure 2.7, the associated cost with this redistribution is

$$\log_2(\hat{p})\alpha + (\hat{p}-1)\frac{\overline{n}}{\hat{p}}\beta,$$

where $\hat{p}$ corresponds to the total number of processes involved in the communication.

4.1.3 Balancing Redistributions

We now formalize the technique utilized in Section 4.1.1 to apply to more general redistribution

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right] \leftarrow \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right] \tag{4.3}$$

where $\widetilde{\mathcal{D}} = \bigcup_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}$.

$$
\begin{array}{l}
\mathbf{A}\left[(0,4),(3,1,2)\right] \\
\qquad\quad \downarrow \;\; \text{1. Permutation} \\
\mathbf{A}\left[(0,4),(3,2,1)\right] \\
\qquad\quad \downarrow \;\; \text{2. All-to-all over mode 1} \\
\mathbf{A}\left[(0,4,1),(3,2)\right] \\
\qquad\quad \downarrow \;\; \text{3. All-to-all over mode 2} \\
\mathbf{A}\left[(0,4,1,2),(3)\right] \\
\qquad\quad \downarrow \;\; \text{4. Permutation} \\
\mathbf{A}\left[(0,1,2,4),(3)\right] \\
\qquad\quad \downarrow \;\; \text{5. All-to-all over mode 4} \\
\mathbf{A}\left[(0,1,2),(3,4)\right]
\end{array}
$$

Figure 4.2: Illustration of the redistribution $\mathbf{A}\left[(0,1,2),(3,4)\right] \leftarrow \mathbf{A}\left[(0,4),(3,1,2)\right]$ performed via balanced redistributions.

Formalization

As discussed in Section 4.1.2, we know that the amount of data replication among processes is the same in (4.3). Further, balanced all-to-all redistributions are of the form

$$
\begin{gathered}
\mathbf{A}\left[\mathcal{D}^{(0)},\ldots,\mathcal{D}^{(v-1)},\mathcal{D}^{(v)} \sqcup \widetilde{\mathcal{D}}^{(0)},\ldots,\mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1-v)}\right] \\
\downarrow \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)},\ldots,\mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)},\mathcal{D}^{(v)},\ldots,\mathcal{D}^{(M-1)}\right],
\end{gathered}
$$

where communication occurs over the processing mesh modes

$$
\widetilde{\mathcal{D}} = \bigcup_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}.
$$

We can generalize the technique utilized in Section 4.1.2 via a series of redistributions that each transfer a single entry in $\widetilde{\mathcal{D}}$ from its starting tensor mode distribution to its ending tensor mode distribution (not necessarily in the correct location though).

For example, consider the redistribution

$$\mathbf{A}\left[(0,1,2),(3,4)\right] \leftarrow \mathbf{A}\left[(0,4),(3,1,2)\right].$$

To transform this redistribution, we can first perform redistributions to move the entry "1" from the mode-1 distribution to the mode-0 distribution. This requires a permutation redistribution, followed by an all-to-all redistribution over mode 1. We can do the same process for the entry "2", and finally for the entry "4". A final permutation redistribution may be required to ensure the entries of each tensor mode distribution (processing mesh modes) are in the correct order. This results in the series of redistributions depicted in Figure 4.2. Notice that we perform a permutation redistribution followed by an all-to-all redistribution over the mode we are moving.

Thus, (4.3) can be implemented as $|\widetilde{\mathcal{D}}| + 1$ permutation redistributions (one last permutation redistributions to reorder all entries within tensor mode distributions) and $|\widetilde{\mathcal{D}}|$ all-to-all redistributions each communicating over a unique entry in $\widetilde{\mathcal{D}}$.

To avoid unnecessary complexity in our analyses, we assume that each process is assigned the same number of elements under the output distribution, denoted $\overline{n}$. If we assume that every mode in $\mathcal{R}\left(N\right)$ requires this transformation (set $\mathcal{R}\left(N\right) =$

$\bigcup_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)}$ in (4.3)), then the associated cost is given by

$$\begin{aligned}
& (N+1)\left(\alpha + \overline{n}\beta\right) + \sum_{u \in \mathcal{R}(N)} \left(\log_2\left(P_u\right)\alpha + \left(P_u - 1\right)\frac{\overline{n}}{P_u}\beta \right) \\
= \quad & (N+1)\left(\alpha + \overline{n}\beta\right) + \log_2\left(\hat{p}\right)\alpha + \sum_{u \in \mathcal{R}(N)} \left(\left(P_u - 1\right)\frac{\overline{n}}{P_u}\beta \right) \\
\approx \quad & (N+1)\left(\alpha + \overline{n}\beta\right) + \log_2\left(\hat{p}\right)\alpha + N\overline{n}\beta \\
= \quad & \left(\log_2\left(\hat{p}\right) + N + 1\right)\alpha + \left(2N+1\right)\overline{n}\beta.
\end{aligned}$$

where $\hat{p} = \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}\right)$ is the number of processes involved in an all-to-all collective instance.

Since we assume that the order of the processing mesh remains constant, the cost of our transformed redistribution is within a factor $2N+1$ of the cost associated with a balanced all-to-all collective over the same set of processing mesh modes.

Observations

It is important to mention here that the transformation analyzed represents the worst case scenario as all processing mesh modes were involved in the communication and no attempts were made to further optimize redistributions (e.g., merging pairs of permutation and all-to-all redistributions when possible). This was done for example purposes only to motivate the idea behind the transformation. Further, for simplicity we ignored the modes each permutation redistribution communicates over as this is not reflected in the current cost model used; however, this information can be tracked if better cost estimates are available.

We now discuss how to reduce the cost associated with the procedure developed in this section by utilizing permutation redistributions to perform some, but preferably all, of the required transformations on data distribution.

4.1.4 Exploiting Processing Mesh Structure

Once again, we consider the order-M tensor $\mathbf{A}$ distributed on the order-N processing mesh $\mathbf{G}$ and the redistribution

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right] \leftarrow \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right] \tag{4.4}$$

where communication occurs over modes $\widetilde{\mathcal{D}} = \bigcup\limits_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup\limits_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}$.

In the previous subsection, we developed a systematic procedure that decomposes this redistribution into a series of permutation and all-to-all redistributions that has an associated cost of

$$\left(\log_2(\hat{p}) + |\widetilde{\mathcal{D}}| + 1\right)\alpha + \left(2|\widetilde{\mathcal{D}}| + 1\right)\overline{n}\beta$$

where $\hat{p} = \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}\right)$ is the number of processes involved in an all-to-all collective instance and $\overline{n}$ is the maximum number of elements assigned to a process under the final distribution.

Notice that if we assume that $\widetilde{\mathcal{D}}^{(u)} = \overline{\mathcal{D}}^{(u)}$ for all $u \in \mathcal{R}(M)$, then this redistribution can be implemented with a single permutation redistribution which results in a significantly lower associated cost. In general, it is not possible to convert all redistributions matching the pattern in (4.4) into a single permutation redistribution, but we should still attempt to use permutation redistributions whenever possible as the cost associated with the latency term is significantly lower than that associated with an all-to-all redistribution while having approximately the same cost associated with the bandwidth term. We now detail a systematic procedure to reduce the cost associated with the redistribution in (4.4) by utilizing permutation redistributions whenever possible.

Formalization

Recall that a permutation redistribution has the pattern

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right] \leftarrow \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right]$$

where communication occurs over modes $\widetilde{\mathcal{D}} = \bigcup_{u \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(u)} = \bigcup_{u \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(u)}$ and $\text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(u)}\right) = \text{prod}\left(\mathbf{P}, \overline{\mathcal{D}}^{(u)}\right)$. In other words, the number of processes involved in a communication over modes $\widetilde{\mathcal{D}}^{(u)}$ must be the same number as would be involved if communicating over modes $\overline{\mathcal{D}}^{(u)}$.

Our goal is to determine the entries of $\widetilde{\mathcal{D}}$ whose transformation can be implemented via permutation redistributions. In doing so, we identify all entries whose transformation can be directly implemented via permutation redistributions. The transformation associated with the remaining entries then need to be decomposed to be implemented in terms of balanced redistributions; however, this procedure reduces the number of entries for which this must be done.

To do this, we consider alternative redistributions of the form

$$\begin{gathered}\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \mathcal{H}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \mathcal{H}^{(M-1)}\right] \\ \downarrow \\ \mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{H}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{H}}^{(M-1)}\right].\end{gathered} \tag{4.5}$$

where $\mathcal{H}^{(u)} \subseteq \widetilde{\mathcal{D}}^{(u)}$, $\overline{\mathcal{H}}^{(u)} \subseteq \overline{\mathcal{D}}^{(u)}$, and $\text{prod}\left(\mathbf{P}, \mathcal{H}^{(u)}\right) = \text{prod}\left(\mathbf{P}, \overline{\mathcal{H}}^{(u)}\right)$. Notice that, as defined, (4.5) corresponds to a balanced permutation redistribution. Therefore, any definition of $\mathcal{H}^{(u)}$ is comprised of entries (processing mesh modes) whose transformation can be implemented via permutation redistributions. We define $\mathcal{J}^{(u)}$ be the union of all definitions of $\mathcal{H}^{(u)}$ and $\overline{\mathcal{J}}^{(u)}$ be the union of all definitions of $\overline{\mathcal{H}}^{(u)}$. We define $\mathcal{N}^{(u)} = \widetilde{\mathcal{D}}^{(u)} \setminus \mathcal{J}^{(u)}$ and $\overline{\mathcal{N}}^{(u)} = \overline{\mathcal{D}}^{(u)} \setminus \overline{\mathcal{J}}^{(u)}$ to be the remaining entries. Finally, we define $\mathcal{J} = \bigcup_{u \in \mathcal{R}(M)} \mathcal{J}^{(u)}$ and $\mathcal{N} = \bigcup_{u \in \mathcal{R}(M)} \mathcal{N}^{(u)}$. We recognize that, in conjunction with the size of the processing mesh, an analysis of the notation for the initial and final tensor distributions can determine the definitions of $\mathcal{J}^{(u)}$ and $\overline{\mathcal{J}}^{(u)}$.

We can now decompose the original redistribution in (4.4) into the five steps depicted in Figure 4.3. Step 1 performs a permutation redistribution that prepares the entries in $\mathcal{J}^{(u)}$ for their transfer from initial tensor mode distribution to final tensor mode distribution. Step 2 performs a permutation redistribution that transfers the entries in $\mathcal{J}^{(u)}$ to their final tensor mode distributions. Step 3 performs a permutation redistribution to prepare the entries of $\mathcal{N}^{(u)}$ for transferring into their final tensor mode distribution. Step 4 performs the (potentially unbalanced) redistribution to transfer the entries of $\mathcal{N}^{(u)}$ to their final tensor mode distributions. Finally, Step 5 performs a permutation redistribution to correctly reorder the entries of each tensor

$$\begin{tikzcd}
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right] \arrow[d, "\text{1. Permutation over } \widetilde{\mathcal{D}}"] \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \mathcal{N}^{(0)} \sqcup \mathcal{J}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \mathcal{N}^{(M-1)} \sqcup \mathcal{J}^{(M-1)}\right] \arrow[d, "\text{2. Permutation over } \mathcal{J}"] \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \mathcal{N}^{(0)} \sqcup \overline{\mathcal{J}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \mathcal{N}^{(M-1)} \sqcup \overline{\mathcal{J}}^{(M-1)}\right] \arrow[d, "\text{3. Permutation over } \widetilde{\mathcal{D}}"] \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{J}}^{(0)} \sqcup \mathcal{N}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{J}}^{(M-1)} \sqcup \mathcal{N}^{(M-1)}\right] \arrow[d, "\text{4. Redistribution}"] \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{J}}^{(0)} \sqcup \overline{\mathcal{N}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{J}}^{(M-1)} \sqcup \overline{\mathcal{N}}^{(M-1)}\right] \arrow[d, "\text{5. Permutation over } \widetilde{\mathcal{D}}"] \\
\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right]
\end{tikzcd}$$

Figure 4.3: Illustration of redistributions for exploiting structure of processing mesh shape. Here, steps 1-3 may be merged into a single permutation collective; they are only shown separately for clarity.

mode distribution.

After applying this transformation, the cost of the redistribution is reduced to

$$\begin{aligned}
& \underbrace{\alpha + \overline{n}\beta}_{\text{Permutation}} + \underbrace{\alpha + \overline{n}\beta}_{\text{Permutation}} + \underbrace{\alpha + \overline{n}\beta}_{\text{Permutation}} + \\
& \underbrace{\mathrm{C}_{\mathcal{N}}^{+}}_{\text{Redistribute}} + \underbrace{\alpha + \overline{n}\beta}_{\text{Permutation}} \\
\approx \quad & 4\alpha + 4\overline{n}\beta + \mathrm{C}_{\mathcal{N}}^{+}
\end{aligned}$$

where $\overline{n}$ represents the number of data elements assigned to a process after the redistribution completes, and $\mathrm{C}_{\mathcal{N}}^{+}$ represents the cost of the redistribution to transfer entries of $\mathcal{N}$ to their associated final tensor mode distribution. If we apply the transformation discussed in the previous section to perform the redistribution for modes $\mathcal{N}$, then we can reduce the estimated cost of

$$\left(\log_2(\hat{p}) + |\widetilde{\mathcal{D}}| + 1\right)\alpha + \left(2|\widetilde{\mathcal{D}}| + 1\right)\overline{n}\beta$$

where $\hat{p} = \mathrm{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}\right)$ to

$$\begin{aligned}
& 4\alpha + 4\overline{n}\beta + \mathrm{C}_{\mathcal{N}}^{+} \\
\approx\ & 4\alpha + 4\overline{n}\beta + (\log_2(\mathrm{prod}(\mathcal{N}, \mathbf{P})) + |\mathcal{N}| + 1)\alpha + (2|\mathcal{N}| + 1)\,\overline{n}\beta \\
=\ & (\log_2(\mathrm{prod}(\mathcal{N}, \mathbf{P})) + |\mathcal{N}| + 5)\alpha + (2|\mathcal{N}| + 5)\overline{n}\beta.
\end{aligned}$$

We see that the transformation discussed in this section can significantly reduce the cost associated with the latency term of the redistribution while keeping the cost associated with the bandwidth term approximately the same.

Observations

A general optimization that can significantly reduce the associated cost of a communication by exploiting the structure of the processing mesh can be described and reasoned about with the developed notation. Being able to describe how to systematically decompose these general redistributions into a series of balanced redistributions is another example of the utility provided by the developed notation. Stated another way, the developed notation is powerful enough to capture the pertinent knowledge to perform important optimizations while exposing a mechanical procedure to perform such optimizations.

4.2 Local Data Movement

In this section, we discuss how local data movements arising from redistributions and local computations can be consolidated together, thereby reducing the associated cost. We focus on the computation of local tensor contractions.

4.2.1 Motivating Example

In Chapter 3, we showed how we derive algorithms for distributed tensor contractions. An examination of the algorithms derived reveals a common pattern. Each algorithm follows the steps

1. Globally redistribute tensor operands.
2. Perform local tensor contraction.
3. Globally reduce and/or redistribute tensor operands (if necessary).

A common method of performing the local tensor contraction is to express the tensor contraction as a matrix-matrix multiplication for which high-performance implementations exist as part of the BLAS interface [1, 28, 29, 40, 88]. To use this approach, one must, in the general case, rearrange the data of each tensor. Assuming we are accumulating into the output operand, this results in potentially performing four memory rearrangements: three to prepare each of the three tensor contraction operands for the matrix-matrix multiplication, and one to arrange the result of the matrix multiplication as specified by the tensor contraction [45]. For instance, consider the tensor contraction

$$\mathbf{C}^{\alpha\beta\eta\iota} = \mathbf{A}^{\alpha\gamma\iota\kappa}\mathbf{B}^{\beta\gamma\eta\kappa},$$

where we assume a generalized column-major storage for each tensor; that is, indices of mode 0 are stored contiguously in memory, followed by mode 1, etc. Of course, there are many possible generalizations of this storage format as we could select any order of indices to store contiguously. This is similar to how row-major storage of a matrix column-major storage with the modes interchanged. To express this computation in terms of matrix-matrix multiplication, we first rearrange the tensors as

1. $\mathbf{A}^{\alpha\iota\gamma\kappa} \leftarrow \mathbf{A}^{\alpha\gamma\iota\kappa}$.
2. $\mathbf{B}^{\gamma\kappa\beta\eta} \leftarrow \mathbf{B}^{\gamma\kappa\beta\eta}$.
3. $\mathbf{C}^{\alpha\iota\beta\eta} \leftarrow \mathbf{C}^{\alpha\beta\eta\iota}$.

Then, we recognize that since the indices of the mode pairs (α, ι), (γ, κ), and (β, η) are stored contiguously in memory, one can view these objects as matrices stored in column-major order allowing a general matrix-matrix multiplication to perform the computation. More precisely, the indices of the mode pair (α, ι) correspond to the "m" dimension, the indices of the mode pair (γ, κ) correspond to the "k" dimension, and the indices of the mode pair (β, η) correspond to the "n" dimension of a matrix-matrix multiplication. However, as the output data may not be stored in the expected order for $\mathbf{C}$, we require one last rearrangement of data to place elements of $\mathbf{C}$ in the correct order.

Expanding our pattern for distributed computations using the knowledge of how local tensor contractions are implemented results in the following series of steps

1. Globally redistribute tensor operands.
2. Rearrange local tensor data.

3. Perform tensor contraction as matrix-matrix multiplication.
4. Rearrange local tensor data (if necessary).
5. Globally reduce and/or redistribute tensor operands (if necessary).

Now, we also know that performing a redistribution via a collective communication proceeds according to the following series of steps

(a) Copy data to a send buffer.

(b) Perform communication of data over network storing receiving data in a receive buffer.

(c) Unpack receive buffer into local storage.

Expanding our distribute algorithm pattern with this knowledge, we arrive at the following flow for our overall computation

1. (a) Copy data to a send buffer.

   (b) Communicate data over the network.

   (c) Unpack receive buffer.
2. Rearrange tensor data.
3. Perform matrix-matrix multiplication.
4. Rearrange tensor data (if necessary).
5. (a) Copy data to a send buffer (if necessary).

   (b) Communicate data over the network (if necessary).

   (c) Unpack receive buffer (if necessary).

Notice that the packing/unpacking of communication buffers and the rearrangement of tensor data both correspond to memory rearrangements. By merging these into a single rearrangement of data, the overhead associated with these steps is reduced. As we discussed previously, the only information needed is the order of tensor mode indices. In doing so, we arrive at our final algorithm template

1. (a) Copy data to a send buffer.
   (b) Communicate data over the network.
2. Unpack receive buffer and rearrange tensor data in one step.
3. Perform matrix-matrix multiplication.
4. Rearrange and pack tensor data in one step (if necessary).
5. (a) Communicate data over the network (if necessary).
   (b) Unpack receive buffer (if necessary).

Notice that we have roughly halved the number of memory copies required to prepare a single tensor for local computation. Because this happens at least once for each tensor, this change removes a large number of unnecessary memory copies.

4.2.2 Generalization

By extending the notation slightly, we can describe how data of each process is locally stored. With this additional knowledge, an expert can optimize unnecessary local data movements, thereby increasing the local performance. We do not detail the extension of the defined notation to incorporate this information as it is not interesting from a theoretical standpoint.

4.3 Summary

In this chapter, we discussed ways to consolidate data movements, both global movements among processes and local movements within processes, associated with parallel tensor contraction algorithms and implementations to improve performance. We discussed two transformations that implement a general redistribution as a series of balanced redistributions, the second of which reduces the cost of the first transformation by exploiting the structure of the processing mesh. Then, we discussed a transformation that consolidated memory movements associated with redistribution and local computation by noticing that the results of both memory movements are permutations on the data.

We stress here the utility obtained from having developed a notation to describe data distribution and redistribution. Not only does the developed notation enable existing transformations to be concisely described, but the notation also facilitates the generalization of such transformations. For instance, a special case of the optimization discussed in Section 4.1.4 is utilized in the distributed-memory tensor library CTF and the distributed-memory linear algebra libraries Elemental and ScaLAPACK. When restricted to matrices, a special case of the same optimization is used for computing certain linear algebra operations, such as the tridiagonalization of a symmetric matrix via Householder transformations [33].

In particular, both optimizations discussed in Section 4.1.4 convert the encountered redistribution into a single permutation redistribution. Section 4.1.4 detailed a procedure that generalizes this idea to arbitrary redistributions, detecting and applying the optimization through intermediate redistributions. In doing so, an optimization whose application choice was binary, now becomes a spectrum of possibilities.

Chapter 5

Implementation and Experimental Results

Previous chapters of this document focused on formalizing different aspects of the development of high-performance distributed-memory implementations for tensor contractions. As we saw in Chapter 3 and Chapter 4, there still may exist places where choices must be made; i.e., the derivation procedure is not entirely systematic. The space of implementations created by these choices can rapidly become too large to be reasonably explored even by an expert. In this chapter, we demonstrate how the ideas presented previously can be translated into a domain-specific language that can be used by prior work in software engineering for efficiently generating the optimal implementation among a space of implementations[1] for applications of considerable importance to the domain of computational chemistry.

We begin this chapter by first briefly discussing the application used to test the

[1] When we refer to optimal, we mean optimal according to the underlying model of computation

ideas developed in this document: the spin-adapted CCSD method from computational chemistry. We then discuss the ROTE library that implements the ideas associated with data distribution and redistribution as an application programming interface (API). Next, we discuss some of the ideas behind the software engineering approach, referred to as Design-by-Transformation (DxT), used to create the space of implementations for CCSD. As we describe DxT, we relate it to aspects of the CCSD application and ROTE library for example purposes. Once this is done, we briefly discuss how the prototype system that implements the ideas of DxT, DxTer, is used in conjunction with ROTE to generate efficient implementations for the CCSD application.

We finally move on to discuss experiments used to test the thesis of this dissertation and provide an analysis of the results. In doing so, we show how the ideas presented here benefit the state of the art in areas of computer science with potential to benefit areas of computational chemistry, and provide an argument for approaches to software development that are based on this clear separation of concerns between implementation and theoretical aspects of the application domain.

5.1 Coupled Cluster Singles and Doubles Method (CCSD)

We provide a brief overview of the application used to motivate the ideas in this dissertation. This section is by no means comprehensive; for a thorough discussion of CCSD as well as other methods from coupled cluster theory, we recommend [20, 21].

5.1.1 Computational Chemistry Background

Quantum mechanical models have proven to be highly accurate for physical systems. These systems are described by an associated function, called the system's *wave function*. Unfortunately, the exact wave function of a system is typically too complex to define explicitly. Instead, an approximation to the wave function is used that maintains useful mathematical properties. The approximated wave function is typically defined by incorporating models for different physical phenomena into a reference wave function.

Different methods, referred to as theories, of approximating the true wave function exist, each having a different associated cost and achieving different levels of accuracy [20, 32, 77, 78]. One such theory, known as coupled cluster theory, has been shown to accurately model chemical systems while retaining desired properties of the model. In coupled cluster theory, a set of so-called *cluster operators* are defined that, when applied to a reference wave function (via the exponential of each cluster operator), provide an exact representation of the wave function associated with the system of interest. Each of these operators describe a different instance within a class of phenomena referred to as *excitation*. Unfortunately, applying all cluster operators to construct the true wave function is prohibitively expensive; instead approximations are made that only apply a subset of the operators defined. A different computational method is then defined by the maximum number of excitations considered when applying cluster operators.

For instance, the method that accounts for both single and double excitations is referred to as the "coupled cluster singles and doubles (CCSD)" method [66], while the method that incorporates single, double, and triple excitations is referred to as the "coupled cluster singles, doubles, and triples (CCSDT)" method [62]. A set of equations is then derived that defines how to compute associated coefficients for representing the approximate wave function for the system of interest. Depending on how the equations are derived, the resulting set of equations can have different computational characteristics. Further, the derived set of equations may require exploiting structure such as symmetry within the tensor data. Incorporating symmetry is a topic of future research and is not discussed in this thesis.

Beyond limiting the set of cluster operators, further approximations can be made or techniques can be used to reduce the complexity of the resulting equations or factorization process. For instance, methods based on perturbation theory can be applied to approximate certain cluster operators (such as the triples operator) [68]. Spin-adaptation refers to a technique that produces sets of equations that can be efficiently computed while enforcing certain constraints to accurately model the system of interest [50, 59, 75].

5.1.2 The Specific Formulation Studied

For the purposes of this dissertation, it suffices to understand that these methods boil down to a computation involving many tensor contractions where the output of one tensor contraction may be the input for another and/or the results must be summed. The specific formulation of the spin-adapted CCSD method studied in this document is given in Figure 5.1. There, we use the notation familiar to computational chemists. Focusing on the computation of G_i^m in Figure 5.1,

$$\sum_e H_e^m t_i^e$$

$$
\begin{aligned}
W^{bm}_{je} &= (2w^{bm}_{je} - x^{bm}_{ej}) + \sum_f (2r^{bm}_{fe} - r^{bm}_{ef}) t^f_j - \sum_n (2u^{nm}_{je} - u^{mn}_{je}) t^b_n \\
&\quad + \sum_{fn} (2v^{fe}_{nm} - v^{fe}_{mn})(T^{bf}_{jn} + \frac{1}{2} T^{bf}_{nj} - \tau^{bf}_{nj}) \\
X^{bm}_{ej} &= x^{bm}_{ej} + \sum_f r^{bm}_{ef} t^f_j - \sum_n u^{mn}_{je} t^b_n - \sum_{fn} v^{fe}_{mn} (\tau^{bf}_{nj} - \frac{1}{2} T^{bf}_{nj}) \\
U^{mn}_{ie} &= u^{mn}_{ie} + \sum_f v^{fe}_{mn} t^f_i \\
Q^{mn}_{ij} &= q^{mn}_{ij} + (1 + \mathcal{P}^{mi}_{nj}) \sum_e u^{mn}_{ie} t^e_j + \sum_{ef} v^{ef}_{mn} \tau^{ef}_{ij} \\
P^{ji}_{mb} &= u^{ji}_{mb} + \sum_{ef} r^{bm}_{ef} \tau^{ef}_{ij} + \sum_e w^{bm}_{ie} t^e_j + \sum_e x^{bm}_{ej} t^e_i \\
H^m_e &= \sum_{fn} (2v^{ef}_{mn} - v^{ef}_{nm}) t^f_n \\
F^a_e &= -\sum_m H^m_e t^a_m + \sum_{fm} (2r^{am}_{ef} - r^{am}_{fe}) t^f_m - \sum_{fmn} (2v^{ef}_{mn} - v^{ef}_{nm}) T^{af}_{mn} \\
G^m_i &= \sum_e H^m_e t^e_i + \sum_{en} (2u^{mn}_{ie} - u^{nm}_{ie}) t^e_n + \sum_{efn} (2v^{ef}_{mn} - v^{ef}_{nm}) T^{ef}_{in} \\
z^a_i &= -\sum_m G^m_i t^a_m - \sum_{emn} (2U^{mn}_{ie} - U^{nm}_{ie}) T^{ae}_{mn} + \sum_{em} (2w^{am}_{ie} - x^{am}_{ei}) t^e_m \\
&\quad + \sum_{em} (2T^{ae}_{im} - T^{ae}_{mi}) H^m_e + \sum_{efm} (2r^{am}_{ef} - r^{am}_{fe}) \tau^{ef}_{im} \\
Z^{ab}_{ij} &= v^{ab}_{ij} + \sum_{mn} Q^{mn}_{ij} \tau^{ab}_{mn} + \sum_{ef} y^{ab}_{ef} \tau^{ef}_{ij} + (1 + \mathcal{P}^{ai}_{bj}) \Bigg\{ \sum_e r^{ej}_{ab} t^e_i \\
&\quad - \sum_m P^{ij}_{mb} t^a_m + \sum_e F^a_e T^{eb}_{ij} - \sum_m G^m_i T^{ab}_{mj} + \frac{1}{2} \sum_{em} W^{bm}_{je} (2T^{ae}_{im} - T^{ae}_{mi}) \\
&\quad - (\frac{1}{2} + \mathcal{P}^i_j) \sum_{em} X^{bm}_{ej} T^{ae}_{mi} \Bigg\}
\end{aligned}
$$

Figure 5.1: Equations for a single iteration of the spin-adapted CCSD method based on the formulation from Scuseria, Scheiner, Lee, Rice, and Schaefer [76]. Following the notation in [76], both superscripts and subscripts of each tensor are used to represent labels assigned to modes (under some order). This notation differs from what is used in this dissertation. Each summation indicates a contraction.

is an example of a tensor contraction involving the tensors H and t. The modes of each tensor involved are labeled with the letters e, m, and i. In our notation, this contraction would have been written as

$$\sum_{\epsilon} \mathbf{H}^{\mu\epsilon}\mathbf{T}^{\epsilon\iota},$$

where the labels ϵ, μ, and ι correspond to the labels e, m, and i. Additionally, focusing on the computation of Q^{mn}_{ij}, the symbol $\mathcal{P}$ indicates a permutation of data; e.g., $\mathcal{P}^{mi}_{nj}\mathbf{X}^{mj}_{ni} = \mathbf{X}^{ni}_{mj}$.

5.2 The Redistribution Operations and Tensor Expressions (ROTE) API

The ROTE API was developed to implement the ideas introduced in this dissertation and to define a set of primitives that experts (possibly automated) could utilize for optimizing applications. The library was designed so that the ideas of this dissertation correspond one-to-one with the code that implements them. For instance, in Figure 5.2 we depict code used to initialize a distributed tensor in the ROTE API and in Figure 5.3, we depict code that computes the tensor contraction

$$\mathbf{G}^{\mu\iota} = \sum_{\epsilon} \mathbf{H}^{\mu\epsilon}\mathbf{T}^{\epsilon\iota}.$$

ROTE was developed using an object-oriented paradigm so properties related to distributed objects are considered members of the objects. This includes relevant size data and, most importantly, the associated distribution and allows the API to

```
#include "rote.hpp"
using namespace rote;
using namespace std;

int main(int argc, char* argv[]) {
  Initialize(argc, argv);
  mpi::Comm comm = mpi::COMM_WORLD;

  try{
    //Declare size of processing mesh
    //(std::vector<unsigned>)
    ObjShape gridSize;
    //<Initialize gridSize>

    //Initialize processing mesh g
    const Grid g(comm, gridSize);

    //Declare size of data tensor
    //(std::vector<unsigned>)
    ObjShape tenSize:
    //<Initialize tenSize>

    //Declare distributed tensor A
    DistTensor<double> A(tenSize, "[(0),(1,2)]|(3)", g);

    //Initialize A with uniformly random elements
    MakeUniform(A);

  } catch (std::exception& e) {
        ReportException(e);
  }

  Finalize();
  return 0;
}
```

Figure 5.2: Initializing distributed tensors in ROTE. This sample initializes an order-2 distributed tensor “A” with uniformly random double-precision floating-point elements distributed over the processing mesh “g” according to the tensor distribution “[(0),(1,2)]|(3)”, which encodes the tensor distribution $\boldsymbol{\mathcal{D}} = [(0),(1,2);(3),0]$.

```
// H[(),(2,3,0,1)] <- H[(0,1),(2,3)]
H__tmp1.AllToAllRedistFrom(H__D_0_1__D_2_3,
                           modes_0_1);

// H[(),(0,1,2,3)] <- H[(),(2,3,0,1)]
H__tmp2.PermutationRedistFrom(H__tmp1,
                              modes_2_3_0_1);
H__tmp1.EmptyData();

// H[(),(0,1)] <- H[(),(0,1,2,3)]
H__tmp3.AllGatherRedistFrom(H__tmp2,
                            modes_2_3);
H__tmp2.EmptyData();

tmpShape = G__D_0_1__D_2_3.Shape();
tmpShape.push_back( g.Shape()[0] * g.Shape()[1] );
G__tmp1.ResizeTo( tmpShape );

// G[(),(2,3),(0,1)]_mie = H[(),(0,1)]_me
//                       * t[(0,1),(2,3)]_ei
LocalContract(1.0,
              H__tmp3.LockedTensor(), indices_me,
              t__D_0_1__D_2_3.LockedTensor(), indices_ei,
              0.0, G__tmp1.Tensor(), indices_mie);
H__tmp3.EmptyData();

// G[(0,1),(2,3)] <- G[(),(2,3),(0,1)]
// (with SumScatter on (0,1))
G__D_0_1__D_2_3.ReduceScatterUpdateRedistFrom(G__tmp1,
                                              1.0,
                                              modes_2);
G__tmp1.EmptyData();
```

Figure 5.3: ROTE API code that computes $\mathbf{G}^{\mu\iota} = \sum_{\epsilon} \mathbf{H}^{\mu\epsilon}\mathbf{T}^{\epsilon\iota}$ as generated by DxTer. Declaration and initialization details are not shown. The distributions of each tensor are attributes of the associated objects. Variables with the prefix “modes” indicate an ordered set of modes and variables with the prefix “indices” indicate an ordered set of labels assigned to the corresponding mode. Each redistribution rule is associated with a “RedistFrom” method.

capture the notation in this dissertation while hiding low level details. As a result, ROTE becomes a domain-specific language for tensor contractions on distributed memory architectures.

Redistributions, in terms of the general rules defined in Figure 2.14 and Figure 2.15, are implemented in the API as sophisticated wrapper routines to underlying MPI collective communications and local tensor contractions are presently implemented as a wrapper to a high-performance matrix-matrix multiplication kernel.

5.3 Design-by-Transformation (DxT) and DxTer

Here, we provide a general overview of the DxT approach to software development [53, 57] and the prototype system DxTer. We then discuss how DxT and DxTer is used in the context of this dissertation work. For a detailed discussion of DxT and DxTer, we encourage the reader to read [58].

5.3.1 Background

The DxT approach to software development is to create application implementations by converting abstract representations of the application to efficient, concrete implementations via a series of correct transformations [57]. This process facilitates the automatic generation of efficient implementations that is ordinarily performed by a human expert.

We now discuss how this approach is applied to the design of implementations for CCSD using the functionality defined in the ROTE API as building blocks, or *primitives* as defined in the DxT terminology. Let us focus on how an implementation for the expression

$$G_i^m = \sum_e H_e^m t_i^e + \sum_{en} (2u_{ie}^{mn} - u_{ie}^{nm}) t_n^e + \sum_{efn} (2v_{mn}^{ef} - v_{nm}^{ef}) T_{in}^{ef}$$

in Figure 5.1 would be created with the DxT approach. We see that **G** is formed by the accumulation of three contractions involving five inputs in total (H_e^m, t_i^e, u_{ie}^{mn}, T_{in}^{ef}, and v_{mn}^{ef}).

First, the computation is encoded as a dataflow graph. Each node in this graph represents some computation and edges between nodes represent data dependencies between computations. For example, each term (contraction) would be represented as a node in the graph, and the data dependencies of each input to the computation of G_i^m (H_e^m, T_{in}^{ef}, v_{mn}^{ef}, u_{ie}^{mn}) would be represented as edges between different nodes in the graph. We mention that no details of how each computation should proceed are defined at this point, only the abstract representations of the computations along with data dependencies are encoded.

Our end goal is to transform our abstract representation of the computation into a representation that only relies on concretely defined operations whose implementations are provided, referred to as *primitives*. We do this by continually applying a set of transformation rules, called *refinements*, to our dataflow graph until all details of the computation are concrete. Once this is done, we have an implementation for the original problem statement. For instance, a refinement corresponds to a transformation that converts a node in our dataflow graph representing a tensor contraction into one representing a stationary variant of the algorithm; we have not yet arrived at a implementation as not all details have been filled in (such as how the necessary redistributions are performed)[2]. Examples of primitives are the implementations provided by the ROTE API of each redistribution rule given in Figures 2.14 and 2.15.

Having defined computations (as dataflow graphs), refinements, and an associated set of primitives, we are able to transform our abstract representation into a concrete implementation. Depending on the order that we apply our refinements, we can arrive at different implementations. Therefore, our set of refinements and primitives define a space of implementations for our algorithm. Of course, we would like to ensure that some of the produced implementations are efficient. For this, we associate a cost with each primitive considered and define transformations on the dataflow graph that, upon application, result in a dataflow graph with better overall cost. These transformations are referred to as *optimizations*. For example, the transformations described in Chapter 4 to convert general redistributions into a series that rely on balanced communications are optimizations as they reduce the cost associated with the bandwidth term of the communications.

At this point, we have a way to create a space of implementations for an application that is based on a systematic process and a way to rank order each implementation. The task of creating an efficient implementation for a specific computation has been reduced to one of searching within a space of implementations. The prototype system DxTer, created by Bryan Marker as part of his dissertation work, was used to implement the ideas of DxT discussed in this section and efficiently search the space of implementations for the optimal [55, 58].

[2] Strictly speaking, refinements are not required to provide more concrete details, but this detail does not add to the discussion.

5.3.2 DxT and This Dissertation

In the context of this dissertation, DxTer used the methods defined in ROTE as primitives to generate efficient implementations for the CCSD application. Due to the generality of the ideas developed in this document, the encoded rules allow DxTer to consider implementations for a particular arrangement of processes, a contribution that is of great importance since, for certain architectures such as the tested Blue Gene/Q architecture, the exact arrangement of processes (called a partition on that architecture) is made available to the user and can be exploited in the optimization.

At this point we cannot guarantee that the generated implementations are optimal for the experimental setup. We simply rely on DxTer to generate implementations that are reasonable in terms of the choices made for algorithmic variants and composition of redistribution rules. In encoding the rules for implementation, the expert should refine the rules so that DxTer can find an efficient solution using the encoded knowledge (this corresponds either to cost adjustments or additions to the encoded knowledge). Note that this process requires significant work by an expert to encode and refine the rules; however, once done, DxTer can be used to generate implementations for all applications relying on the encoded knowledge. Additionally, the implementations developed using DxTer and ROTE were post-processed to add blocking so that the problems being computed could mitigate the effect of extra storage required for data replication. This enabled larger problems to be solved and potentially achieve better overall performance, as discussed in Section 5.4.4.

5.4 Experimental Results

We now turn to results of computing the spin-adapted CCSD method on massively-parallel distributed-memory architectures. The ideas of this dissertation directly apply to this application as the computations involved are a series of dense, non-symmetric, tensor contractions. Implementations were generated assuming a specified order-4 processing mesh and sizes for tensors that represent real-world applications of CCSD. For those familiar with the method, this corresponds to problem specifications where the number of virtual orbitals is an order of magnitude greater than the number of occupied orbitals.

5.4.1 Target Architectures

Experiments were conducted on the IBM Blue Gene/Q [83] and Cray XC30 [22] computing architectures, comparing against two other widely used packages for performing distributed-memory parallel tensor computations: the Cyclops Tensor Framework (CTF)[3] and the CCSD module in the NorthWest computational Chemistry (NWChem) software package [14]. Details of CTF and the related module in NWChem are given in Chapter 6. Implementations were created in CTF that represent the same set of equations computed by implementations generated by DxTer with ROTE. As NWChem is not properly tuned for the IBM Blue Gene/Q architecture, we only compare against NWChem on the Cray XC30 architecture.

For the IBM Blue Gene/Q architecture, compute nodes consist of sixteen 1600MHz PowerPC A2 cores for a combined theoretical peak performance of 204.8 GFlops per node using double-precision arithmetic. Nodes are connected via a five-dimensional

[3] The version of CTF used was dated Feb. 5, 2015.

torus topology that supports a chip-to-chip bandwidth of 2GB/s.

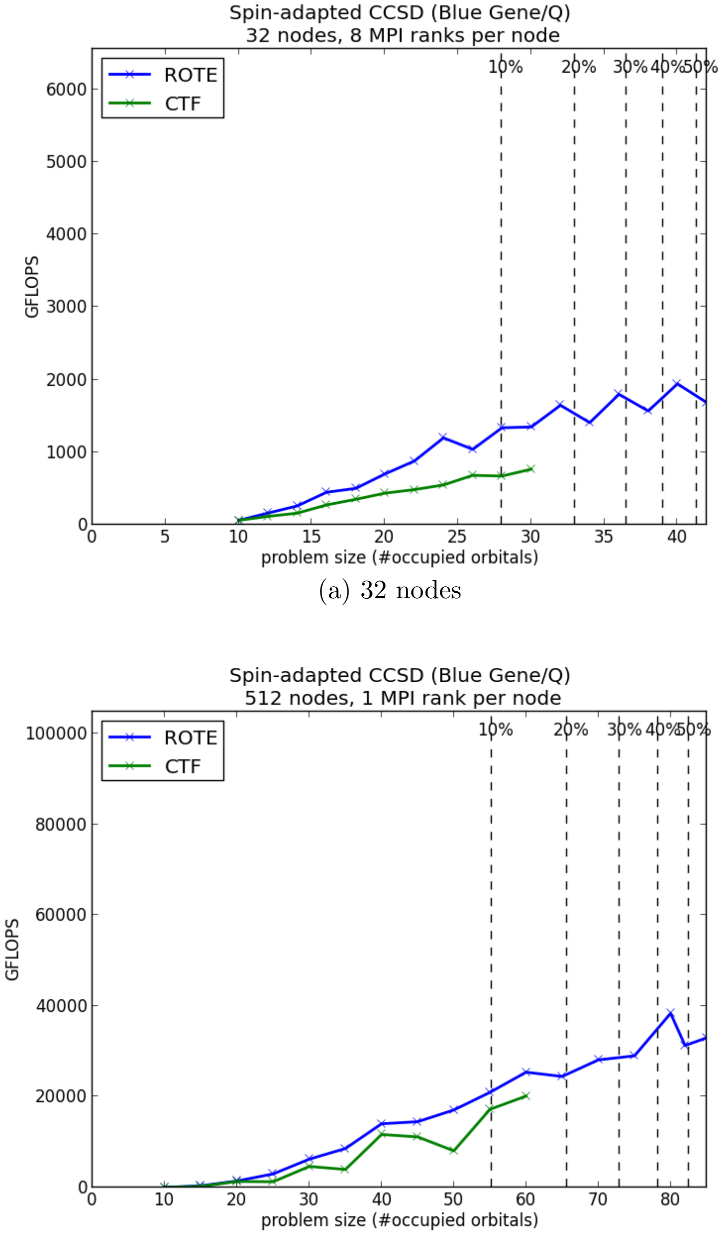
For the Cray XC30 architecture, compute nodes consist of twelve 2400MHz Intel "Sandybridge" cores for a combined theoretical peak of 460.8 GFlops per node using double-precision arithmetic. In this case, nodes are connected via a Cray Aries interconnect with DragonFly topology [43] supporting a global bandwidth of 23.7 TB/s.

When possible, experiments were performed using the same parameters for compiler optimization, underlying BLAS library, and MPI implementation (multithreaded ESSL on Blue Gene/Q and multithreaded MKL on Cray XC30). For a fair comparison, tuning parameters for each set of implementations were adjusted and values associated with the best performance results were reported. For CTF, ROTE, and NWChem, these include a form of on-node configuration; however, in the case of ROTE, the additional parameter of the processing mesh used to generate implementations was tested.

We mention here one important aspect of the results for the ROTE-based implementations. For computational chemists, a sought-after property of any distributed-memory tensor library is that high performance is achieved when fifty percent of the available memory is reserved for inputs. As we see in the following sections, our implementations achieve this.

5.4.2 IBM Blue Gene/Q Experiments

Comparison with CTF. In Figure 5.4, we show experimental results comparing ROTE-based implementations against implementations created in CTF using 32 nodes (512 cores) and 512 nodes (8192 cores). In each of these experiments, we



(a) 32 nodes

(b) 512 nodes

Figure 5.4: Performance results on IBM Blue Gene/Q architecture comparing to CTF with different numbers of compute nodes. The top of each graph represents the theoretical peak for the configuration. Dashed vertical lines indicate the percentage of total memory consumed by inputs.

plot raw performance achieved by each implementation as the associated problem size is increased (in terms of the number of occupied orbitals represented in the computation). The top of each graph indicates the maximum performance, or *peak*, achievable by the configuration. Each set of implementations were tested until memory exhaustion; therefore, the final data point shown represents the largest problem able to be computed with the available resources. Dashed vertical lines are overlaid on the figures to depict the percentage of total memory required to store the inputs for the computation with associated problem size.

It can be seen that the ROTE-based implementations outperform CTF. Additionally, the ROTE-based implementations can perform computations that require approximately three times the amount of memory for input using the same resources.

In the case of the 32-node experiment, a configuration of eight MPI ranks per node and eight OpenMP threads per MPI rank achieved the best results for each library; for the 512-node experiment, a configuration of one MPI rank and sixty four OpenMP threads per rank achieved the best performance for both libraries. These configurations correspond to using a processing mesh of size $(4, 4, 4, 4)$ for the 32-node experiment and a processing mesh of size $(2, 4, 8, 8)$ for the 512-node experiment for the ROTE library. As mentioned in the previous subsection, we cannot comment on the processing mesh configuration chosen by CTF as this is handled internally by the library. However, notice that in the 32-node experiment, all grid modes have the same dimension meaning that many all-to-all redistributions can be implemented in terms of permutation redistributions as discussed in Chapter 4.

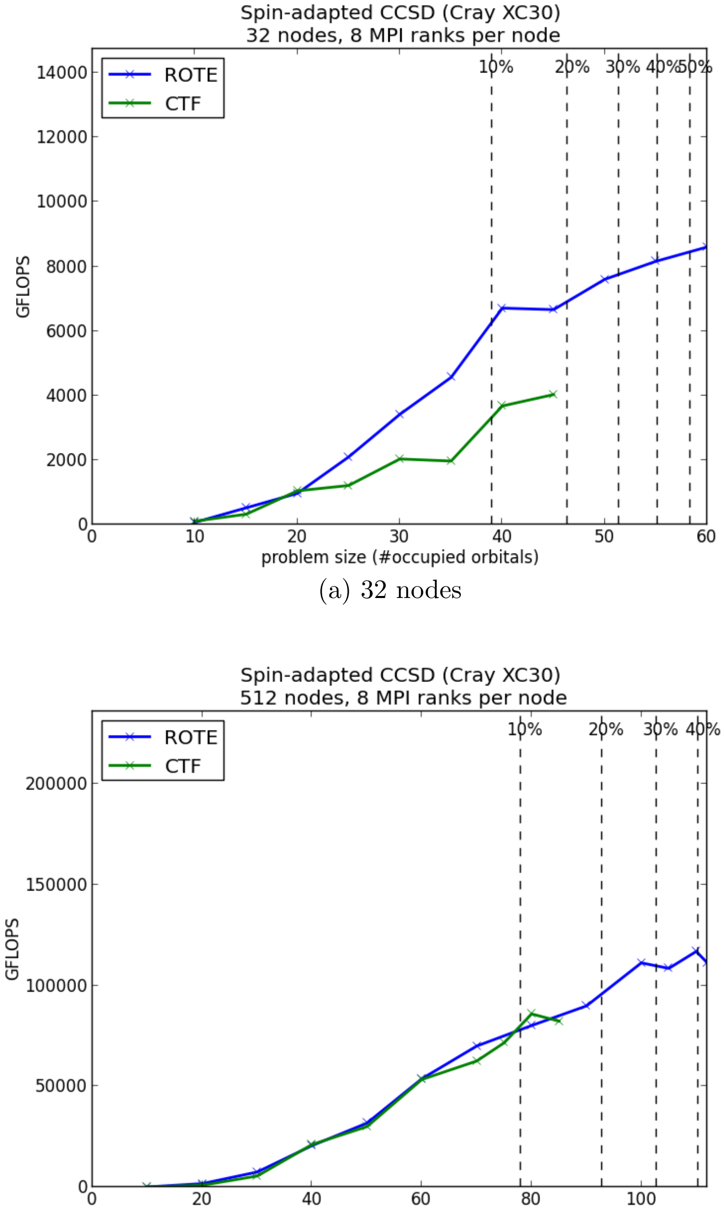
Interestingly, for the 512-node experiment, a processing mesh of size $(2, 4, 8, 8)$ achieved the best performance even though a processing mesh configuration of

$(8, 8, 8, 8)$ can be created by assigning eight MPI ranks per node and eight OpenMP threads per rank instead of one MPI rank and sixty four OpenMP threads. The reason for this is likely due to a trade-off between allgather redistributions and all-to-all redistributions that occurs in each configuration. In the $(8, 8, 8, 8)$ configuration, many all-to-all redistributions can be implemented with permutation redistributions, but the allgather redistributions may involve a greater number of processes than in the $(2, 4, 8, 8)$ configuration.

Notice that, based on the problem specification, many allgather redistributions occur on mode 0 and mode 1 of the processing mesh. In the case of the $(8, 8, 8, 8)$ configuration, this means that up to sixty four processes are involved in the redistributions, whereas in the $(2, 4, 8, 8)$ configuration, only at most eight are. This is an example of where the trade-off between performance of different transformations and collective communications must be considered to achieve a higher performing implementation; in the $(2, 4, 8, 8)$ case, we limit the opportunities to implement an all-to-all collective as a permutation collective, but reduce the cost of allgather collectives, whereas the $(8, 8, 8, 8)$ case has the opposite property.

5.4.3 Cray XC30 Experiments

Comparison with CTF. In Figure 5.5, we show performance results comparing the ROTE-based implementation to the CTF implementation using 32 nodes (768 cores) and 512 nodes (12288 cores) of the Cray XC30 architecture. As with the experiments performed on the IBM Blue Gene/Q, in each of these experiments, we plot raw performance achieved by each implementation as the associated problem size is increased (in terms of the number of occupied orbitals represented in the computation). The top of each graph indicates the maximum performance, or *peak*,



(a) 32 nodes

(b) 512 nodes

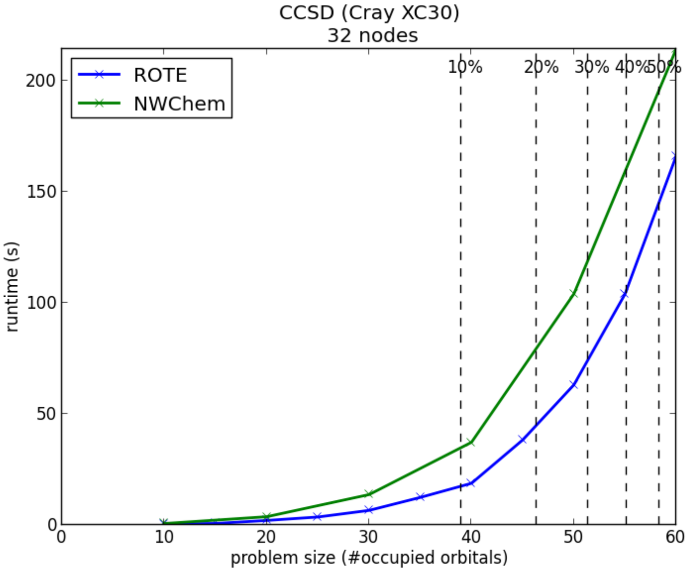
Figure 5.5: Performance results on Cray XC30 architecture comparing to CTF with different numbers of compute nodes. The top of each graph represents the theoretical peak of the configuration. Dashed vertical lines indicate the percentage of total memory consumed by inputs.

achievable by the configuration. Each set of implementations were tested until memory exhaustion; therefore, the final data point shown represents the largest problem able to be computed with the available resources. Dashed vertical lines are overlaid on the figures to depict the percentage of total memory required to store the inputs for the computation with associated problem size.

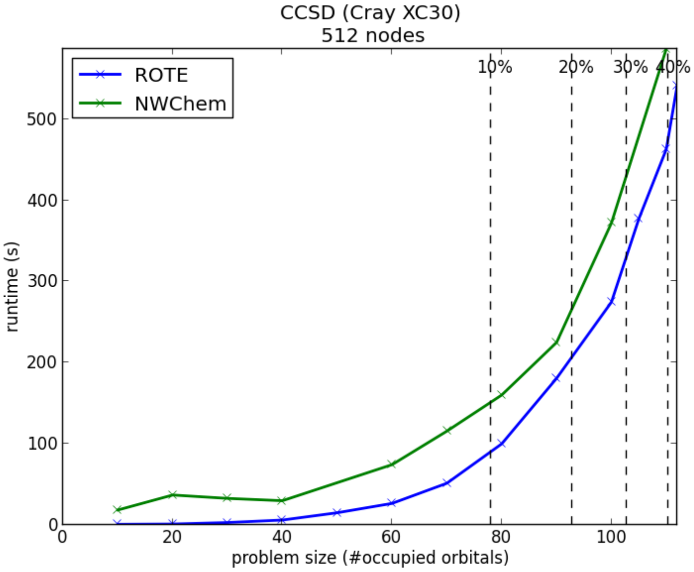
A node configuration of eight MPI ranks per node and three OpenMP threads per MPI rank achieved the best results for both the ROTE-based implementation and CTF in both the 32-node and 512-node experiments. These configurations correspond to using a processing mesh of size $(4, 4, 4, 4)$ for the 32-node case and a processing mesh of size $(8, 8, 8, 8)$ for the 512-node case. We see that the ROTE-based implementation is able to achieve performance that is at least comparable to the CTF implementation, but in some cases outperforms CTF.

It is interesting that in this case, the performance of the ROTE-based implementation does not outperform CTF code as much as on the BG/Q architecture. We suspect that this is due in part to the relatively faster communication network of the Cray XC30 architecture that allows differences between communication performance to be hidden [60, 64].

Comparison with NWChem. In Figure 5.6, we show performance results comparing the ROTE-based implementation to the hand-coded implementation provided in the NWChem package [84] (version 6.3) using 32 nodes (768 cores) and 512 nodes (12288 cores) of the Cray XC30 architecture. When performing these experiments, an implementation a bug was discovered in more recent NWChem implementations that potentially hindered the achievable performance. Instead, a recent stable version of NWChem was used for performance experiments.



(a) 32 nodes



(b) 512 nodes

Figure 5.6: Timing results on Cray XC30 architecture comparing to NWChem with different numbers of compute nodes (lower is better). Dashed vertical lines indicate the percentage of total memory consumed by inputs.

Although the set of equations implemented in NWChem are chemically equivalent to those in the ROTE-based implementations, they have different computational and communication characteristics. Of particular note is the fact that the NWChem implementation relies on an approach to computing the dominant term of CCSD that significantly reduces the overall storage requirement. This enables the NWChem implementation to perform computations that require significantly more storage for inputs as the ROTE-based implementations cannot exploit the structure of the computation to obtain this benefit. Therefore, we plot the relative runtimes of each implementation instead of the raw performance achieved and only show results up to the largest computation able to be performed by ROTE-based implementations with the given resources (to conserve computing resources). Here, we also use dashed vertical lines to indicate the percentage of memory required to store the inputs associated with different computations.

In the 32-node case, implementations based on a processing mesh of size $(4, 4, 4, 4)$ achieved the best performance for the ROTE-based implementation, and implementations based on a processing mesh of size $(8, 8, 8, 8)$ achieved the best in the 512-node case. Considering the assumptions about these experiments, we see that although the ROTE-based implementations can only solve smaller problems with the same resources, they are able to outperform the hand-coded implementations provided by NWChem.

5.4.4 The Importance of Blocking

As mentioned in Section 2.5, blocking the overall computation into a series of subproblems can mitigate the effect on storage at the expense of additional communication overhead; a larger block size requires a larger amount of storage to perform the overall computation but also reduces the amount of overhead due to latency. As mentioned in Section 2.5.1, picking a small block size can negatively impact the performance of local computation. Given the statement that computational chemists would like to limit required workspace so that at least fifty percent of available storage was reserved for inputs, an analysis was performed for each term in the CCSD application (assuming a representative processing mesh configuration) to determine the largest blocking parameter that respected this requirement.

In Figure 5.7, we show results performed on the IBM Blue Gene/Q architecture using 512 compute nodes comparing a ROTE-based implementation where computations are blocked into smaller subproblems to an implementation where computations are not blocked into smaller subproblems. In these experiments only the block size was changed; the implementation in both tests is the same.

Based on the results, we see that, for these experiments, blocking the computation results in slightly lower performance for a given problem size but enables the computation of problem sizes that require up to approximately three times as much storage. These results vary depending on the specific processing mesh configuration chosen as different, potentially cheaper, collectives can be used for various redistributions.

To our knowledge, CTF does not feature a way for the user to block computations as is done in ROTE. Therefore, this difference in design choice is the primary reason why CTF is only able to perform a significantly smaller problem as compared to the ROTE-based implementation.

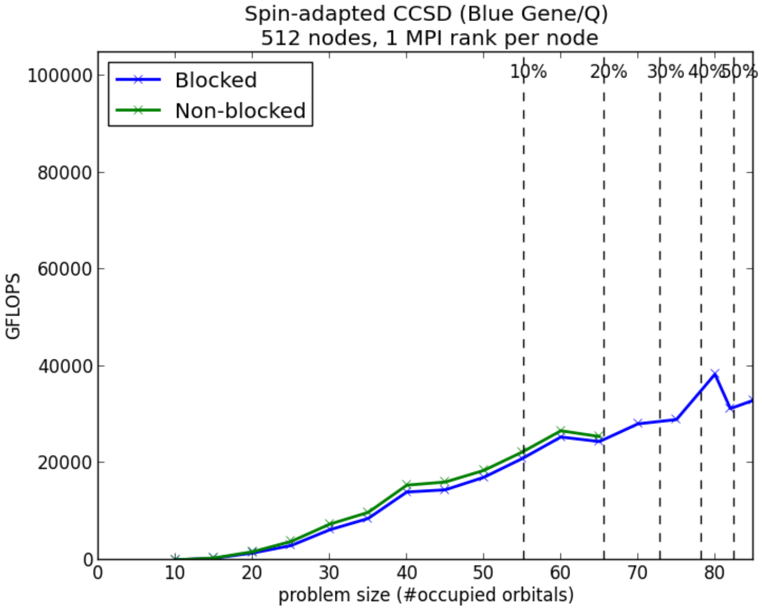


Figure 5.7: Comparison of ROTE-based implementations on 512 nodes of the IBM Blue Gene/Q architecture when enabling/disabling the blocking of computations. The top of the graph represents the theoretical peak for this architecture. Dashed vertical lines indicate the percentage of total memory consumed by inputs.

5.4.5 Weak Scalability Experiments

Metrics referred to as *scalability* are commonly used measure the quality of a parallel implementation. The scalability of a parallel implementation refers to how the implementation performs when additional processing elements are added to the system.

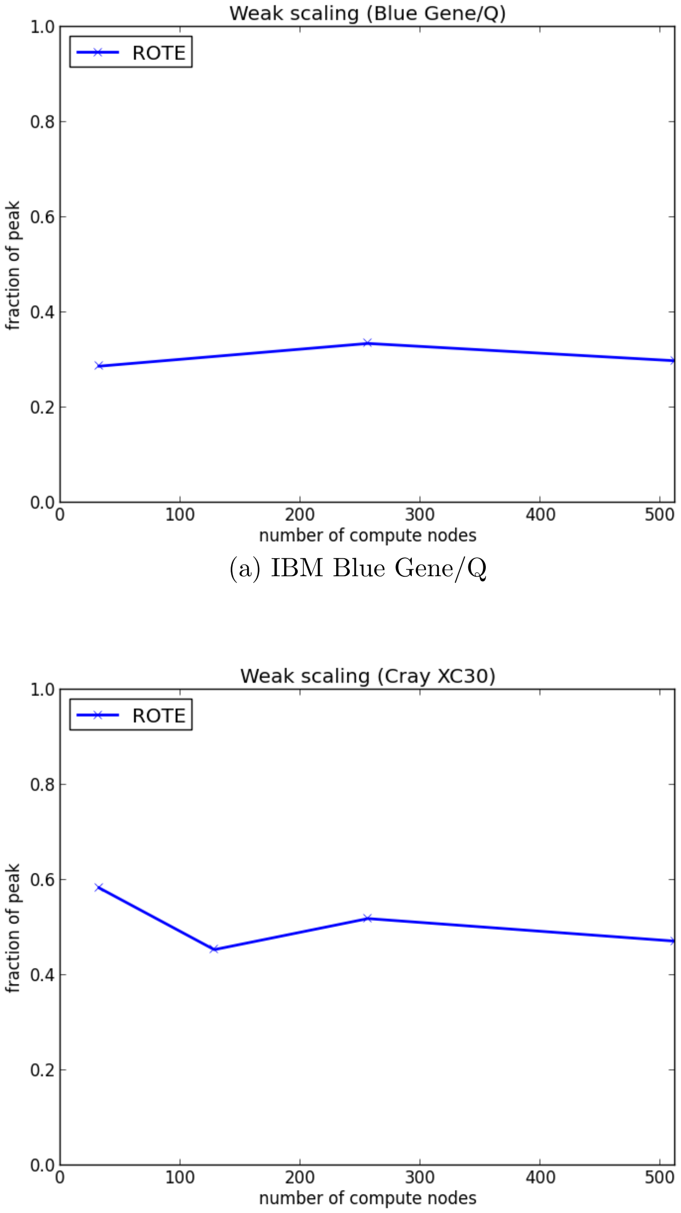
Strong scalability refers to how the implementation performs when additional processing elements are added to the system with the goal of reducing the time required to perform a computation of fixed size. For this metric, a perfectly scalable implementation would halve the time required to perform a computation when the number of processing elements is doubled. Perfect strong scalability is not achievable for problems in this domain. Because the total amount of computation remains constant while the overhead associated with communication inherently scales with the number of processes in the system, the efficiency of each processing element cannot be maintained.

Weak scalability refers to how the implementation performs when additional processing elements are added to the system with the goal of maintaining the computing efficiency. For this metric, a perfectly scalable implementation would maintain the computing efficiency when the problem size (in terms of memory usage) is scaled in proportion to the number of processes involved. For example, a perfectly scalable implementation would maintain the computing efficiency when twice as many processes are used to solve a problem requiring twice as much storage for inputs. For applications in this domain, we are typically concerned with weak scalability.

In Figure 5.8, we show the results of weak scalability experiments of the ROTE-based implementations up to 512 nodes on both the IBM Blue Gene/Q and Cray

XC30 architectures (8192 and 12288 cores respectively). For each configuration tested, approximately fifty percent of the available storage was reserved for inputs of the CCSD application thus making memory use per process roughly constant. An examination of these results reveals that the computing efficiency of each process is held nearly constant as the number of nodes is increased in the Blue Gene/Q case, and drops slightly as the number of nodes increases in the Cray XC30 case. This provides some evidence of relatively good weak scalability in the implementation used. To prove weak scalability, a cost analysis would have to be performed for the generated implementation, which is beyond the scope of this dissertation.

We mention here that the implementations chosen for each configuration in this case are not necessarily the same. We expect that if the same implementation was chosen for the 32 node case as for the 512 node case, the weak scalability would worsen as the relationship between processing mesh configuration and collectives utilized may be less advantageous. Different implementations were chosen for each configuration to mitigate this effect. This is still a valid measure of scalability, as our implementation for this domain is to incorporate the configuration of the processing mesh when developing an implementation for the application (therefore the implementation being measured is not dependent on any specific implementation chosen).



(a) IBM Blue Gene/Q

(b) Cray XC30

Figure 5.8: Weak scalability of the ROTE-based implementations on different architectures. Weak scalability measurements were performed with problems whose inputs consume approximately fifty percent of available memory. The top of the graph corresponds to the theoretical peak of each configuration tested.

5.5 Summary

Commonly, application users in this domain would like to reserve at least fifty percent of the available storage for inputs and still achieve high performance from libraries. Our results show that, by considering the processing mesh configuration and incorporating the blocking of problems into subproblems, one can achieve significant performance while restricted to only half the available storage for workspace. By incorporating these features into our approach, we have significantly advanced the state of the art in the domain of high-performance distributed-memory parallel tensor computations. Further, our results provide additional evidence in support of the idea to enable the automation of high-performance application implementation by formalizing the key aspects of a domain.

Chapter 6

Related Work

In this chapter, we compare approaches and ideas in other projects related to those in this thesis.

6.1 Tensor Contraction Engine (TCE)

The Tensor Contraction Engine project (circa 2005) [8, 35, 36, 37], a part of the NWChem package for computational chemistry [14, 84], is perhaps the first project of its kind with the goal of automating the task of creating high-performance distributed-memory implementations for applications based on tensor contractions. In fact, many important aspects of designing high-performance implementations in this domain originated in TCE. As such, it can be considered the precursor to many of the other projects discussed in this chapter.

The overarching goal of TCE is to create a distributed-memory implementation for a problem specification that performs the fewest number of operations in the available space. To do this, TCE searches a space of algorithms created by iteratively applying a series of transformations and analyses until it either finds a valid algorithm or halts having found no viable algorithm. The applied transformations include algebraic transformations that manipulate the number of required operations to implement the problem specification. One side effect of manipulating the number of required operations is that the storage requirements of the devised method can also be affected (either increase or decrease). For this, transformations and analyses related to compiler techniques consider how to manipulate the storage requirements; these include transformations related to loop fusion and those that consider the addition/removal of temporary variables.

If an algorithm is selected for implementation, TCE synthesizes implementations that rely on dynamic scheduling to maintain load-balance among processes [35, 46, 63]. This is done because the arithmetic complexity associated with different tensor contractions in a typical application can vary greatly, making approaches that utilize all processes for a single task wasteful. When assigned a task of work, each process asynchronously retrieves the required data from other processes, performs the computation, and updates the relevant portions of the output.

Other works relying on dynamic task scheduling for parallelizing tensor applications include the Dynamically Load-balanced Tensor Contraction (DLTC) library [48]. In this work, a hierarchical view of the tasks is created, allowing for groups of computations to be assigned to a single process. In doing so, a better balance of load among processes can be achieved.

Most relevant to this dissertation is the approach TCE takes to perform and optimize the distributed computations. Contrary to the approach developed here, tensors are interpreted as matrices when distributed (they are "flattened"). Because the distributed computation proceeds based on Cannon's algorithm, redistributions are implemented via permutation collectives. Optimizations to reduce communication are then applied when it is determined that specific tensor operands need not be redistributed for the computation to proceed. As such, only the redistributions required to prepare each operand for subsequent computations are optimized; TCE does not examine the redistributions required by the distributed tensor contraction algorithm in conjunction with those required for subsequent contractions.

In this work, we develop a notation and language that describes how distributed computations must proceed and how redistributions can efficiently be implemented (even optimized). With this knowledge encoded, we can optimize a series of computations together. Most relevant to the comparison with TCE is that an automated system, having encoded the ideas developed in this document, can perform communication optimizations similar to what TCE does as well as optimizations that TCE cannot currently perform.

We mention here that transformations utilized by TCE, such as the algebraic transformations, are not considered in this work. As such, one key difference between these two works is that here we assume the given problem specification is optimized already, whereas TCE can perform that optimization itself.

6.2 Advanced Concepts in Electronic Structure III (ACES III)

The Advanced Concepts in Electronic Structure (circa 2011) [23] is a package that provides parallel implementations for various methods from computational chemistry that are written in the Super Instruction Assembly Language (SIAL) and are computed using the Super Instruction Processor (SIP) [71]. The SIP has both static and runtime system components that are used in conjunction to implement tensor contractions written in SIAL. We mention here that the level of abstraction exposed by SIAL is vastly different from that exposed by the current work. In this work, we consider entire distributed objects and derive algorithms for entire computations (applying blocking when necessary); whereas, in ACES and SIAL, the user must reason about each block of data.

Perhaps the most distinct aspect of ACES compared to this work is that ACES relies on a master-slave paradigm for data redistribution. The master initially determines how data should be distributed among processes and then handles all redistribution requests. Because of this design decision, redistributions cannot effectively utilize the underlying communication network as a single process is responsible for orchestrating the required redistributions. The burden on the master process increases as processes are added to the system, thereby decreasing the ability to effectively utilize the communication network. It is for this reason that we do not compare against ACES III.

6.3 Cyclops Tensor Framework (CTF)

The Cyclops Tensor Framework (circa 2014) [81] is a high-performance distributed-memory library developed for both dense and sparse tensor contraction computations that supports forms of tensor symmetry. As the ideas developed in this dissertation do not presently support sparse data structures or forms of symmetry, we focus on the aspects of CTF that pertain to dense, non-symmetric tensors.

Given a problem specified as a series of tensor contractions, CTF computes a single contraction at a time, handling data distribution, redistribution, and algorithm choice internally. As with this work, CTF relies on elemental-cyclic distributions of data. In contrast to this work that encodes several collective communications, redistributions of data between contraction operations in CTF are implemented via all-to-all collectives and redistributions performed within a distributed tensor contraction only rely on those specified by the chosen algorithm (typically broadcast, permutation, and reduction collectives).

The specific algorithm chosen by CTF for computing the current tensor contraction is based on a cost model analysis of different combinations of generalized SUMMA algorithms and 3D-like parallel algorithms for matrix-matrix multiplication. More specifically, CTF considers a nested form of SUMMA algorithms (one algorithm for each stationary variant) in conjunction with replication-based 3D-like algorithms that replicate the smallest tensor operand in order to reduce the amount of communication overhead.

As CTF separates the communications between distributed tensor contractions from communications as part of the tensor contraction algorithm, redistributions cannot be optimized across a series of tensor contractions at once. This is another difference between CTF and the work of this dissertation.

6.4 RRR and The Contraction Algorithm for Symmetric Tensors (CAST)

The RRR framework[1] (circa 2014) [70] is a high-performance distributed-memory library designed for dense tensor contractions. Computations involving tensors with forms of symmetry are supported through the Contraction Algorithm for Symmetric Tensors (CAST) [69], which is built on the RRR framework. Here, we discuss ideas underlying the RRR as they apply to CAST as well.

The underlying idea behind RRR is similar in spirit to those discussed in this document. RRR relies on elemental-cyclic data distributions and for redistribution relies on three communication primitives that are implemented via pipelined versions (when applicable) of reduction, broadcast, and all-to-all collectives; the three primitives are: reduction, recursive broadcast, and rotation. Additionally, RRR derives algorithms for individual tensor contraction operations that follow the pattern of redistributing data, performing local computations, and finally performing any final redistributions. RRR considers a space of algorithms created by applying the derivation process to all possible distributions of operands and selects the best according to a cost model.

The derivation process utilized by RRR is more like recursion, in comparison to the derivation method developed in this document that is akin to constraint propagation [44]. As part of the underlying theory, RRR defines a set of valid tensor mode distributions of paired tensor modes that can be involved in a distributed computation and defines the necessary redistributions that enable valid local computations.

[1] The RRR framework is named after the communication primitives utilized: reduction, recursive broadcast, and rotation.

The combination of redistributions and local computations to be performed depends on the relationship between the pair of tensor modes being considered. The distributed contraction algorithm then becomes a recursive application of this process to each set of paired tensor modes. As a result of this approach, RRR can consider generalizations of SUMMA algorithms, 3D-like algorithms, and Cannon's algorithm for computation. Contrast this approach to that taken in this work where we consider the entire tensor distribution of each object together and propagate information until we arrive at a valid algorithm.

As with CTF, RRR only considers how to optimize communications in individual tensor contractions and between pairs of contractions; optimizing a series tensor contractions at once is not supported yet.

6.5 Elemental

The Elemental project (circa 2012) [65] is a high-performance distributed-memory library for dense linear algebra. Elemental relies on elemental-cyclic and block-cyclic distributions of data and formalizes both data distributions and redistributions in terms of collective communications.

As we have mentioned throughout this document, the ideas in this work are heavily influenced by those underlying Elemental, except that the ideas for distribution and redistribution are generalized to support arbitrary-order tensors on arbitrary-order processing meshes. Additionally, the ideas underlying how algorithms for matrix-matrix multiplication in Elemental have been made systematic.

We mention here that it is due to the rigor behind the ideas in Elemental that made key insights in this dissertation possible.

6.6 Summary

In this chapter we summarized works related to the ideas developed in this dissertation. As we have discussed in the relevant sections, a significant difference between our work and these projects is that our work facilitates the optimization of a series of tensor contractions at once, instead only optimizing tensor contractions individually. This is facilitated by the fact that the process of algorithm derivation is described in the same notation as that used for distribution and redistribution.

Chapter 7

Conclusion

In this chapter, we summarize our results and mention opportunities for future research.

7.1 Contributions

In this dissertation, we presented a notation that formalizes three major aspects of distributed-memory parallel algorithm design for tensor contractions of dense, non-symmetric tensors: data distribution, redistribution, and algorithm derivation. This facilitated the automatic generation of high-performance implementations for a series of tensor contractions. The defined notation was implemented in an API (ROTE). The knowledge from this dissertation was encoded into the prototype system DxTer to create high-performance implementations for the spin-adapted CCSD method from computational chemistry. A post-processing step was applied to block the computations to reduce the required workspace.

We now, in more detail, present the major achievements and contributions of this dissertation.

7.1.1 A Notation for Data Distributions of Tensors

Data distribution is required when computing on distributed-memory systems. As discussed in Section 2.3, this dissertation focused on elemental-cyclic distributions of elements as they have been shown to be useful in many areas of high-performance computing. As done in Elemental for matrices, this work related the idea of a data distribution to that of a set partition [73]. Using this underlying idea, a general notation was developed to express arbitrary-order tensors distributed on arbitrary-order processing meshes that concisely expresses the pertinent information.

7.1.2 A Notation for Data Redistributions of Tensors

Effective use of collective communications for data redistribution is at the heart of bulk synchronous parallel algorithms like those encountered in this domain. This work links a broad set of collective communications to the redistributions expressed with the defined notation. In doing so, we provided a useful abstraction for experts when considering how data moves among processes. Care was taken to only choose collective communications that are balanced. As such, an expert can more easily determine how to efficiently implement an arbitrary redistribution.

7.1.3 A Generalization of Transformations for Improving Performance

When tasked with optimizing an application for performance, experts wish to utilize every technique known to them to improve performance. In Chapter 4, we demonstrated the utility and expressiveness of the defined notation by generalizing and formalizing a series of transformations that, in certain situations, can significantly improve the predicted cost of the application. This provides a general set of techniques to use, enabling the expert to quickly transform implementations into what is predicted to be more efficient implementations.

7.1.4 A Systematic Method for Algorithm Derivation

Recognizing that different algorithmic variants are more appropriate than others in certain situations, we developed a systematic approach to deriving members of a family of algorithms. With this, the number of considerations required by a (human or mechanical) expert when designing algorithms with efficient implementations can be significantly reduced.

7.1.5 An API for Distributed Tensor Library Development

The ideas presented in this document were implemented in the ROTE API for distributed-memory parallel tensor computations. The ROTE API is designed to be a one-to-one mapping from the ideas presented here, allowing a simple transition from developed algorithms in the notation to corresponding implementation.

7.1.6 An Advancement in State-of-the-Art Tensor Computations

The ideas presented in this document were tested by applying them to the spin-adapted CCSD method from computational chemistry using the DxTer prototype system for code generation. Results from these experiments show that the ideas in this document can significantly improve upon the state of the art in terms of both performance and memory usage (in some cases a fifty percent improvement in performance in conjunction with a factor three improvement in memory). In addition, this demonstrates that the ideas and ROTE API can be incorporated into existing tools for automatic code generation.

7.1.7 A New Case Study for DxTer

The utility and flexibility of automated tools and software-engineering methodologies must be demonstrated for them to gain acceptance from a broader audience. As leveraged in this work, both the utility and flexibility of DxTer were demonstrated for the domain of distributed-memory applications based on tensor contractions, thereby providing a new case study for DxTer. The ideas developed in this document provide the required knowledge for DxTer to be applied.

7.2 Future Work

Here we discuss topics for future research related to this dissertation.

7.2.1 Symmetry

The most important feature lacking in this work is the consideration of symmetric objects for distribution. Certain applications in computational chemistry that require tensor contractions, such as CCSDT(Q) [47, 62], exploit this feature. We will investigate how to incorporate symmetry into the notation and theory in future research.

7.2.2 Sparsity

In addition to symmetry, some methods rely on sparsity in the tensors to reduce the overall computational cost and storage requirements [31, 42, 52]. A key benefits of only considering dense tensors is that each process can be assigned a predictable and structured set of elements while maintaining balance (in computation and storage) among processes. We plan to investigate what forms of sparsity can be incorporated in the developed notation, thereby supporting a broader set of applications.

7.2.3 Additional Families of Algorithms

As argued in this dissertation, incorporating families of algorithms is crucial to tailoring implementations. In this work, we focused on the stationary family of algorithms; however, so-called 3D families of algorithms [2, 73, 79] for matrix-matrix multiplication have benefits in certain settings and generalizations have been incorporated into other related projects. We will investigate how to incorporate that family, and other potential families, of algorithms in the developed notation as part of future research.

7.2.4 Additional Data Distributions

We chose to focus on formalizing elemental-cyclic distributions. Other projects rely on different forms of Cartesian distributions [8, 23] and other applications as well. For instance, some chemistry methods rely on distributions that are more like blocked distributions than elemental-cyclic [9]. We plan to investigate how different Cartesian distributions can be incorporated in the defined notation as well as the potential benefits of doing so in future research.

7.2.5 Generalizations of the Derivation Process

In Chapter 3, we presented a procedure for algorithm derivation that expresses the elementwise definition of a tensor contraction as a two-step approach: first, specifying the local computations to be performed; second, specifying the global accumulations needed. It is unclear if there are benefits to converting the original elementwise definition to a different two-step, or a multi-step, approach. We will investigate this in future research.

7.2.6 Additional Optimizing Transformations

We discussed how the notation can describe two optimizing transformations to improve performance of redistributions in Chapter 4. We plan to investigate if other transformations can be expressed. Additionally, we plan to investigate if there are other communication patterns that should be described by the current notation. Doing so would create additional knowledge that automated systems can utilize for reasoning or provide knowledge that can reduce the space of choices to consider when optimizing an application.

7.2.7 Additional Tensor Operations

The application focused on in this dissertation arises from computational chemistry and the operation investigated was the tensor contraction. Tensor operations, such as factorizations, have also been shown to arise in the area of data analysis [4, 5, 45]. As part of future research, we plan to investigate how to incorporate these operations into the defined notation and formalism.

7.2.8 Heuristics for Reducing the Space of Implementations

There is opportunity to generalize the insights discussed in Chapter 4 so that arbitrary redistributions (including reduction, duplication, and intentional load-imbalance) can be efficiently implemented. Systematically determining an efficient composition of redistribution rules to implement an arbitrary redistribution can significantly limit the search space needed to be considered by the (human or mechanical) expert. We will investigate the feasibility of this idea as well as the potential impact on implementation generation.

7.2.9 Aiding Automated Tools

As demonstrated in Section 5.4.4, blocking computations into subproblems can significantly reduce the amount of workspace required for computation. In Section 3.2.2, we gave guidance for selecting the correct tensor modes to block along. Presently, the process of blocking computations was performed as a post-processing step after an efficient implementation was generated. Incorporating this knowledge of blocking into automated systems provides additional expert knowledge with which to make decisions.

Appendices

Appendix A

Proofs of Redistribution Rules

In this appendix, we prove that each of the redistribution rules given in Figure 2.14 and Figure 2.15 can be implemented via balanced forms of simultaneous invocations of the associated collective communication communicating over the specified set of processing mesh modes. We assume an offset parameter of $\sigma = 0$ as minor modifications can be made to the proofs to enable arbitrary assignments of σ.

A.1 Proofs of Correctness Strategy

Recall that when we say that a redistribution is performed or communicated "over modes $\widetilde{\mathcal{D}}$ of $\mathbf{G}$", we mean that multiple instances of the associated collective are invoked simultaneously with each instance involving all processes whose locations in $\mathbf{G}$ only differ in the modes specified by $\widetilde{\mathcal{D}}$.

For example, performing a redistribution associated with the allgather collective over modes $\widetilde{\mathcal{D}} = (2, 0)$ of the order-4 processing mesh $\mathbf{G}$ means that simultaneous allgather instances are invoked, each involving all processes whose locations in $\mathbf{G}$ differ only in mode 0 and mode 2. As a result, a redistribution performed over modes $\widetilde{\mathcal{D}}$ of a N-order processing mesh of size $\mathbf{P}$ involves $\text{prod}\left(\mathbf{P}, \mathcal{R}\left(N\right) \setminus \widetilde{\mathcal{D}}\right)$ simultaneous invocations, each involving $\hat{p} = \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}\right)$ processes.

We can capture the interaction of processes involved in the same collective invocation with four sets of processes

1. the set of processes involved in the same communication[1], "peers", denoted $\mathcal{Q}$,
2. the set of processes that send portions of the required data to be communicated, "sources", denoted $\mathcal{S}$,
3. the set of processes that receive portions of the required data that is communicated, "targets", denoted $\mathcal{T}$, and
4. the subset of $\mathcal{S}$ from which each target $\mathbf{q} \in \mathcal{T}$ obtains all required data, "contributors", denoted $\mathcal{S}^{(\mathbf{q})}$.

.

By definition, any set of contributors must be a subset of the senders as only senders can send data. Similarly, the set of senders and targets must be subsets of the peers. For instance, in the case of the all-to-all collective, every process involved is a sender, target, and the set of contributors for a given target is the entire set of senders. In contrast, in the case of the permutation collective, every process is both a sender and target, but each target must be assigned a single unique contributor. Observe that Table A.1 summarizes the relationship between $\mathcal{Q}$, $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{S}^{(\mathbf{q})}$ for each

[1] Group in MPI.

	General constraints on defined sets		
Collective	$\mathcal{S}$	$\mathcal{T}$	$\mathcal{S}^{(\mathbf{q})}$
All-to-all	$\mathcal{S} = \mathcal{Q}$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Allgather	$\mathcal{S} = \mathcal{Q}$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Reduce-scatter	$\mathcal{S} = \mathcal{Q}$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Allreduce	$\mathcal{S} = \mathcal{Q}$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Scatter	$\lvert\mathcal{S}\rvert = 1$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Broadcast	$\lvert\mathcal{S}\rvert = 1$	$\mathcal{T} = \mathcal{Q}$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Gather-to-one	$\mathcal{S} = \mathcal{Q}$	$\lvert\mathcal{T}\rvert = 1$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Reduce-to-one	$\mathcal{S} = \mathcal{Q}$	$\lvert\mathcal{T}\rvert = 1$	$\mathcal{S}^{(\mathbf{q})} = \mathcal{S}$
Permutation	$\mathcal{S} = \mathcal{Q}$	$\mathcal{T} = \mathcal{Q}$	$\left(\lvert\mathcal{S}^{(\mathbf{q})}\rvert = 1\right) \wedge$ $\forall_{\mathbf{p},\mathbf{q}\in\mathcal{T}} \left(\mathbf{p} \neq \mathbf{q}\right) \implies \left(\mathcal{S}^{(\mathbf{p})} \neq \mathcal{S}^{(\mathbf{q})}\right)$

Table A.1: Communication patterns of different collective communications.

collective considered.

In this work, we are concerned with showing that each redistribution defined in Figure 2.14 and Figure 2.15 can be implemented with a simultaneous balanced instantiations of the associated collective communicating over the correct set of processing mesh modes. To do this, for each redistribution rule, we need to show that each target can obtain the required data from the correct set of contributors and that the associated communication is balanced. We assume that if this has been done, each process can correctly invoke the associated collective to ensure the redistribution succeeds.

We use proofs for the redistributions associated with the all-to-all, scatter, gather-to-one, and permutation collectives as templates for all other redistributions as they provide representative examples of the strategies employed.

A.2 Lemmas

Consider a multiindex $\mathbf{i}$, a size $\mathbf{I}$, and the ordered sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ such that $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$. To help us more easily reason about where data resides under different distributions, we must understand the relationship between the result $\text{prod}\left(\mathbf{I}, \mathcal{C}\right)$ and both the results $\text{prod}\left(\mathbf{I}, \mathcal{A}\right)$ and $\text{prod}\left(\mathbf{I}, \mathcal{B}\right)$ as well as the analogous relationship for the multi2linear function involving $\mathbf{i}$. Understanding these relationships is crucial to the proofs of correctness for our defined redistributions. We define this relationship for the prod function in Lemma 1.

Lemma 1. *Given an order-M size array $\mathbf{I}$, and an ordered set $\mathcal{D} \sqcup \widetilde{\mathcal{D}}$, then*

$$\text{prod}\left(\mathbf{I}, \mathcal{D} \sqcup \widetilde{\mathcal{D}}\right) \quad = \quad \text{prod}\left(\mathbf{I}, \mathcal{D}\right) \cdot \text{prod}\left(\mathbf{I}, \widetilde{\mathcal{D}}\right).$$

Proof: If $M = 0$, the claim trivially holds. Otherwise, let $\mathbf{u}^{(0)} = \mathbf{I}\left(\mathcal{D}\right)$, $\mathbf{u}^{(1)} = \mathbf{I}\left(\widetilde{\mathcal{D}}\right)$, and $\mathbf{u} = \mathbf{u}^{(0)} \sqcup \mathbf{u}^{(1)} = \mathbf{I}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right)$. By the definition of prod,

$$\begin{aligned}
\text{prod}\left(\mathbf{I}, \mathcal{D} \sqcup \widetilde{\mathcal{D}}\right) &= \text{prod}\left(\mathbf{I}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right)\right) \\
&= \text{prod}\left(\mathbf{u}\right) \\
&= \prod_{\ell \in \mathcal{R}(M)} u_\ell \\
&= \prod_{\ell \in \mathcal{R}\left(|\mathbf{u}^{(0)}|\right)} u_\ell^{(0)} \cdot \prod_{\ell \in \mathcal{R}\left(|\mathbf{u}^{(1)}|\right)} u_\ell^{(1)} \\
&= \text{prod}\left(\mathbf{u}^{(0)}\right) \cdot \text{prod}\left(\mathbf{u}^{(1)}\right) \\
&= \text{prod}\left(\mathbf{I}\left(\mathcal{D}\right)\right) \cdot \text{prod}\left(\mathbf{I}\left(\widetilde{\mathcal{D}}\right)\right) \\
&= \text{prod}\left(\mathbf{I}, \mathcal{D}\right) \cdot \text{prod}\left(\mathbf{I}, \widetilde{\mathcal{D}}\right).
\end{aligned}$$

endofproof

We define the analogous relationship for the multi2linear function in Lemma 2.

Lemma 2. *Given an order-M multiindex* $\mathbf{i}$*, corresponding size array* $\mathbf{I}$*, and an ordered set* $\mathcal{D} \sqcup \widetilde{\mathcal{D}}$*, then*

$$\begin{aligned}\text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{D} \sqcup \widetilde{\mathcal{D}}\right) &= \text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{D}\right) + \\ &\quad \text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \widetilde{\mathcal{D}}\right) \text{prod}\left(\mathbf{I}, \mathcal{D}\right).\end{aligned}$$

Proof: If $M = 0$, the claim trivially holds. Otherwise, let $\mathbf{u}^{(0)} = \mathbf{i}\left(\mathcal{D}\right)$, $\mathbf{u}^{(1)} = \mathbf{i}\left(\widetilde{\mathcal{D}}\right)$, and $\mathbf{u} = \mathbf{u}^{(0)} \sqcup \mathbf{u}^{(1)} = \mathbf{i}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right)$. Similarly define $\mathbf{v}^{(0)}$, $\mathbf{v}^{(1)}$, and $\mathbf{v}$ so that $\mathbf{v} = \mathbf{I}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right)$. By definition of multi2linear and Lemma 1,

$$\begin{aligned}
& \text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{D} \sqcup \widetilde{\mathcal{D}}\right) \\
= \; & \text{multi2linear}\left(\mathbf{i}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right), \mathbf{I}\left(\mathcal{D} \sqcup \widetilde{\mathcal{D}}\right)\right) \\
= \; & \text{multi2linear}\left(\mathbf{u}, \mathbf{v}\right) \\
= \; & \sum_{\ell \in \mathcal{R}(|\mathbf{u}|)} u_\ell \cdot \text{prod}\left(\mathbf{v}, \mathcal{R}\left(\ell\right)\right) \\
= \; & \sum_{\ell \in \mathcal{R}\left(|\mathbf{u}^{(0)}|\right)} u_\ell^{(0)} \cdot \text{prod}\left(\mathbf{v}^{(0)}, \mathcal{R}\left(\ell\right)\right) \\
& + \sum_{\ell \in \mathcal{R}\left(|\mathbf{u}^{(1)}|\right)} u_\ell^{(1)} \cdot \text{prod}\left(\mathbf{v}^{(1)}, \mathcal{R}\left(\ell\right)\right) \cdot \text{prod}\left(\mathbf{v}^{(0)}\right) \\
= \; & \text{multi2linear}\left(\mathbf{u}^{(0)}, \mathbf{v}^{(0)}\right) + \text{multi2linear}\left(\mathbf{u}^{(1)}, \mathbf{v}^{(1)}\right) \text{prod}\left(\mathbf{v}^{(0)}\right) \\
= \; & \text{multi2linear}\left(\mathbf{i}\left(\mathcal{D}\right), \mathbf{I}\left(\mathcal{D}\right)\right) \\
& + \text{multi2linear}\left(\mathbf{i}\left(\widetilde{\mathcal{D}}\right), \mathbf{I}\left(\widetilde{\mathcal{D}}\right)\right) \text{prod}\left(\mathbf{I}\left(\mathcal{D}\right)\right) \\
= \; & \text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \mathcal{D}\right) \\
& + \text{multi2linear}\left(\mathbf{i}, \mathbf{I}, \widetilde{\mathcal{D}}\right) \text{prod}\left(\mathbf{I}, \mathcal{D}\right).
\end{aligned}$$

endofproof

A.3 Proofs of Correctness

One now establish the correspondance between each collective communication in Figure 2.6 and its associated redistribution in Figure 2.14 and Figure 2.15.

A.3.1 All-to-all

Theorem 1. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The redistribution*

$$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \mathcal{D}^{(v)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1-v)}; \mathcal{E}, w\right]$$
$$\downarrow$$
$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)}, \mathcal{D}^{(v)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right]$$

where $(v-1) \in \mathcal{R}(M)$ and π represents a permutation of entries, can be performed via simultaneous all-to-all collectives communicating over modes $\bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(m)}$ in $\mathbf{G}$.

Proof: We prove this for the simple case entries have not been permuted and that $\mathcal{E} = \emptyset$ and $w = 0$. Identical arguments can be made for other consistent permutations, and assignment of $\mathcal{E}$ and w.

Let

$$\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \mathcal{D}^{(v)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1-v)}\right),$$

$$\overline{\boldsymbol{\mathcal{D}}} = \left(\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)}, \mathcal{D}^{(v)}, \ldots, \mathcal{D}^{(M-1)}\right),$$

$$\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M-v)} \widetilde{\mathcal{D}}^{(m)},$$

and

$$\mathcal{D} = \bigcup_{m \in \mathcal{R}(M)} \mathcal{D}^{(m)}.$$

Consider how an arbitrary all-to-all collective would proceed. We know how to define the sets $\mathcal{Q}$, $\mathcal{S}$, and $\mathcal{T}$.

Our first goal is to show that the required set of elements for each target $\mathbf{q} \in \mathcal{T}$ can be obtained from the associated set of contributors $\mathcal{S}^{(\mathbf{q})} = \mathcal{Q}$. Our strategy is to consider an arbitrary element required by $\mathbf{q}$ under $\overline{\boldsymbol{\mathcal{D}}}$ and construct a process $\mathbf{p}$ such that $\mathbf{p} \in \mathcal{S}$ and is assigned the same element under $\boldsymbol{\mathcal{D}}$.

Consider the target $\mathbf{q} \in \mathcal{T}$ and the element at location $\mathbf{i}$ assigned to $\mathbf{q}$ under $\overline{\boldsymbol{\mathcal{D}}}$. Assign the entries of $\mathbf{p}$ in the modes specified by $\mathcal{R}\left(N\right) \setminus \widetilde{\mathcal{D}}$ to be the same as $\mathbf{q}$. This ensures that if $\mathbf{q}$ is assigned elements under $\overline{\boldsymbol{\mathcal{D}}}$, then $\mathbf{p}$ is as well

Now, consider the mode $m \in \mathcal{R}\left(M\right)$ such that $m < v$. Notice that in this case

$$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right) \supseteq \mathcal{I}_m^{(\mathbf{q})}\left(\mathcal{D}^{(m)} \sqcup \overline{\mathcal{D}}^{(m)}\right)$$

meaning that $\mathbf{p}$ is assigned the index i_m of mode m under $\boldsymbol{\mathcal{D}}$.

This argument can be applied for all other modes $m < v$. We ensure the same location in $\mathbf{p}$ is not assigned more than once as, by definition, no two tensor mode distributions can share the same entries meaning we only ever assign each entry of $\mathbf{p}$ once.

Now consider the case where $m \geq v$. Based on $\overline{\boldsymbol{\mathcal{D}}}$, i_m is defined as

$$i_m = \text{multi2linear}\left(\mathbf{q}, \mathbf{P}, \mathcal{D}^{(m)}\right) + j^{(0)}\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)}\right)$$

for some non-negative integer $j^{(0)}$.

Recall that each mode-m index assigned to $\mathbf{p}$ under $\boldsymbol{\mathcal{D}}$ can be defined as

$$\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m-v)}\right) + j^{(1)}\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m-v)}\right)$$

for some non-negative integer $j^{(1)}$.

By Lemma 2

$$\begin{aligned}&\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m-v)}\right) = \\ &\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)}\right) + \text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right)\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)}\right),\end{aligned}$$

so each mode-m index assigned to $\mathbf{p}$ under $\boldsymbol{\mathcal{D}}$ can also be defined as

$$\begin{aligned}\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)}\right) &+ \text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right)\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)}\right) \\ &+ j^{(1)}\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m-v)}\right)\end{aligned}$$

By Lemma 1 and Lemma 2, each mode-m index assigned to $\mathbf{p}$ under $\boldsymbol{\mathcal{D}}$ can also be defined as

$$\begin{aligned}&\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \mathcal{D}^{(m)}\right) + \\ &\left(\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right) + j^{(1)}\text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right)\right)\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)}\right)\end{aligned}$$

Assign the entries of $\mathbf{p}$ in the modes specified by $\widetilde{\mathcal{D}}^{(m-v)}$ such that

$$\text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right) \equiv j^{(0)} \ \left(\text{mod prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right)\right).$$

Notice that $i_m \in \mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m-v)}\right)$ as, based on our construction of $\mathbf{p}$, we can always choose a $j^{(1)}$ such that

$$j^{(0)} = \text{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right) + j^{(1)}\text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m-v)}\right)$$

and all other relevant entries of $\mathbf{p}$ are defined similarly to $\mathbf{q}$. Therefore, $\mathbf{p}$ is assigned the mode-m index i_m under $\boldsymbol{\mathcal{D}}$.

Applying this argument to all modes $m \in \mathcal{R}(M)$ defines the remaining entries for $\mathbf{p}$ as $\widetilde{\mathcal{D}} = \bigcup\limits_{m \in \mathcal{R}(M-1-v)} \widetilde{\mathcal{D}}^{(m)}$. Therefore, we have constructed a process $\mathbf{p}$ that is assigned the element at location $\mathbf{i}$ under $\boldsymbol{\mathcal{D}}$. As $\mathbf{p} \in \mathcal{S}$, by Table A.1, we know that $\mathbf{p} \in \mathcal{S}^{(\mathbf{q})}$.

endofproof

A.3.2 Scatter

Theorem 2. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The redistribution*

$$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), \left(w + j \cdot \operatorname{prod}\left(\mathbf{P}, \mathcal{E}\right)\right)\right]$$
$$\downarrow$$
$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$$

can be performed via simultaneous scatter collective instances communicating over the modes $\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}$ of $\mathbf{G}$.

Proof: Let

$$\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right),$$

$$\overline{\boldsymbol{\mathcal{D}}} = \left(\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right),$$

and

$$\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}.$$

Consider how an arbitrary scatter collective would proceed. We know how to define the sets $\mathcal{Q}$, $\mathcal{S}$, and $\mathcal{T}$.

Here, we must show that each target $\mathbf{q} \in \mathcal{T}$ can obtain all elements in $\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}\right)$ from the same sender. Our goal is to construct a unique $\mathbf{p} \in \mathcal{S}$ such that

$$\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}; \mathcal{E}, w\right) \subseteq \boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), w + j \cdot \mathrm{prod}\left(\mathbf{P}, \mathcal{E}\right)\right).$$

for each $\mathbf{q} \in \mathcal{T}$.

By assigning the entries of $\mathbf{p}$ in the modes specified by $\mathcal{R}\left(N\right) \setminus \widetilde{\mathcal{D}}$ to the same as those in $\mathbf{q}$, we ensure that $\mathbf{p} \in \mathcal{Q}$.

By assigning the entries of $\mathbf{p}$ in the modes specified by $\widetilde{\mathcal{D}}$ such that

$$j = \mathrm{multi2linear}\left(\mathbf{p}, \mathbf{P}, \widetilde{\mathcal{D}}\right)$$

then we know that $\mathbf{p}$ is assigned data under $\boldsymbol{\mathcal{D}}$ if $\mathbf{q}$ is under $\overline{\boldsymbol{\mathcal{D}}}$.

Notice that

$$\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}; \mathcal{E}, w\right) \subseteq \boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), w + j \cdot \mathrm{prod}\left(\mathbf{P}, \mathcal{E}\right)\right)$$

meaning that under $\boldsymbol{\mathcal{D}}$, $\mathbf{p}$ is assigned all elements required by $\mathbf{q}$ under $\overline{\boldsymbol{\mathcal{D}}}$. Therefore, $\mathbf{p} \in \mathcal{S}^{(\mathbf{q})}$. Finally, notice that this is the case for any $\mathbf{q} \in \mathcal{T}$. Thus, $\mathbf{p} \in \mathcal{S}$ and $|\mathcal{S}| = 1$. **endofproof**

A.3.3 Gather-to-one

Theorem 3. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The redistribution*

$$\begin{gathered}\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right] \\ \downarrow \\ \mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), \left(w + j \cdot \operatorname{prod}\left(\mathbf{P}, \mathcal{E}\right)\right)\right]\end{gathered}$$

can be performed via simultaneous gather-to-one collective instances communicating over the modes $\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}$ of $\mathbf{G}$.

Proof: Let

$$\boldsymbol{\mathcal{D}} = \left(\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right)$$

and

$$\overline{\boldsymbol{\mathcal{D}}} = \left(\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}\right).$$

Consider how an arbitrary gather-to-one collective would proceed. We know how to define the sets $\mathcal{Q}$, $\mathcal{S}$, and $\mathcal{T}$.

Here, we must show that the single target $\mathbf{q} \in \mathcal{T}$ can obtain all elements in $\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}\right)$ from processes in $\mathcal{S}$. Our goal is to construct a unique $\mathbf{p} \in \mathcal{S}$ such that

$$\mathbf{i} \in \boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}; \mathcal{E}, w\right),$$

for an arbitrary element at location $\mathbf{i} \in \boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), w + j \cdot \operatorname{prod}\left(\mathbf{P}, \mathcal{E}\right)\right)$.

By assigning the entries of $\mathbf{p}$ in the modes specified by $\mathcal{R}(N) \setminus \widetilde{\mathcal{D}}$ to the same as those in $\mathbf{q}$, we ensure that if $\mathbf{q}$ is assigned data under $\overline{\boldsymbol{\mathcal{D}}}$ then $\mathbf{p}$ is assigned data under $\boldsymbol{\mathcal{D}}$. Further, we ensure that $\mathbf{p} \in \mathcal{Q}$ and therefore $\mathbf{p} \in \mathcal{S}$.

By using the same argument as that used in Theorem 1, we can assign the entries of $\mathbf{p}$ in the modes specified by $\widetilde{\mathcal{D}}$ to ensure that $\mathbf{p}$ is assigned the element at location $\mathbf{i}$ under $\boldsymbol{\mathcal{D}}$. The difference here is that we can apply the argument to all modes in $\mathcal{R}(M)$, instead of only the last $M-1-v$ as was done in Theorem 1. Therefore, $\mathbf{p} \in \mathcal{S}^{(\mathbf{q})}$.

endofproof

A.3.4 Permutation

Theorem 4. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The redistribution*

$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$$
$$\downarrow$$
$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$$

can be performed via simultaneous permutation collective instances communicating over the modes $\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}$ of $\mathbf{G}$ under the assumption

$$\forall_{m \in \mathcal{R}(M)} \text{prod}\left(\mathbf{P}, \overline{\mathcal{D}}^{(m)}\right) = \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m)}\right).$$

Proof: For this proof, we assume $\mathcal{E} = \emptyset$ and $w = 0$. Identical arguments can be made for other assignments of $\mathcal{E}$ and w. Define the variables $\boldsymbol{\mathcal{D}}$, $\overline{\boldsymbol{\mathcal{D}}}$, and $\mathcal{D}$ as was done in Theorem 1.

Let

$$\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}.$$

Consider how an arbitrary permutation collective would proceed. We know how to define the sets $\mathcal{Q}$, $\mathcal{S}$, and $\mathcal{T}$.

Here, we must show that each target $\mathbf{q} \in \mathcal{T}$ can obtain all elements in $\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}\right)$ from a single unique process $\mathbf{p} \in \mathcal{S}$. This then enforces the constraints that $|\mathcal{S}^{(\mathbf{q})}| = 1$ and each $\mathcal{S}^{(\mathbf{q})}$ is unique. Our goal is to construct a unique $\mathbf{p} \in \mathcal{S}$ such that

$$\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}\right) = \boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}\right).$$

By assigning the entries of $\mathbf{p}$ in the modes specified by $\mathcal{R}\left(N\right) \setminus \widetilde{\mathcal{D}}$ to the same as those in $\mathbf{q}$, we ensure that $\mathbf{p} \in \mathcal{S}$ and that if $\mathbf{q}$ is assigned data, so is $\mathbf{p}$.

We know that

$$\boldsymbol{\mathcal{I}}^{(\mathbf{q})}\left(\overline{\boldsymbol{\mathcal{D}}}\right) = \mathcal{I}_0^{(\mathbf{q})}\left(\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}\right) \otimes \cdots \otimes \mathcal{I}_{M-1}^{(\mathbf{q})}\left(\mathcal{D}^{(M-1)} \sqcup \overline{\mathcal{D}}^{(M-1)}\right)$$

and

$$\boldsymbol{\mathcal{I}}^{(\mathbf{p})}\left(\boldsymbol{\mathcal{D}}\right) = \mathcal{I}_0^{(\mathbf{p})}\left(\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}\right) \otimes \cdots \otimes \mathcal{I}_{M-1}^{(\mathbf{p})}\left(\mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}\right).$$

Notice that if we set the entries of $\mathbf{p}$ such that

$$\mathbf{p}\left(\widetilde{\mathcal{D}}^{(m)}\right) = \mathbf{q}\left(\overline{\mathcal{D}}^{(m)}\right)$$

for each $m \in \mathcal{R}(M)$, then

$$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)} \sqcup \widetilde{\mathcal{D}}^{(m)}\right) = \mathcal{I}_m^{(\mathbf{q})}\left(\mathcal{D}^{(m)} \sqcup \overline{\mathcal{D}}^{(m)}\right).$$

In other words, we set the entries of $\mathbf{p}$ in modes specified by $\widetilde{\mathcal{D}}^{(m)}$ to the entries of $\mathbf{q}$ in the modes specified by $\overline{\mathcal{D}}^{(m)}$.

Similar to the reasoning in Theorem 1, we are assured that we do not set the same entry of $\mathbf{p}$ more than once. Notice that at this point we have set all entries of $\mathbf{p}$ as

$$\bigcup_{m \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(m)} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}.$$

This process must create a unique $\mathbf{p}$ for each target in $\mathcal{T}$ as each target is unique and each entry in $\mathbf{p}$ was created based on only one entry in $\mathbf{q}$. Therefore, there is only one such $\mathbf{p}$ defined and $\mathbf{p} \in \mathcal{S}^{(\mathbf{q})}$.

endofproof

A.3.5 Others

Theorem 5. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The following redistributions can be implemented via simultaneous instances of the associated collective communicating over modes in $\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m \in \mathcal{R}(M)} \overline{\mathcal{D}}^{(m)}$.*

Collective	*Redistribution*
Allgather	$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right]$
Reduce-scatter	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A} \sqcup \mathcal{K}}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \widetilde{\mathcal{D}}^{(0)}, \ldots, \widetilde{\mathcal{D}}^{(M-1-v)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)}; \mathcal{E}, w\right]$
Allreduce	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A} \sqcup \mathcal{K}}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \widetilde{\mathcal{D}}^{(0)}, \ldots, \widetilde{\mathcal{D}}^{(M-1-v)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}; \mathcal{E}, w\right]$

Proof: For the allgather collective, we can apply the same strategy as that used in Theorem 1.

For the reduce-scatter collective, notice that the redistribution considered is equivalent (in terms of data moved) to first collecting all necessary elements from other processes, and then locally performing the reduction of elements. The pattern of data movement is then equivalent to that of the all-to-all collective, thus we can apply the same strategy as that used in Theorem 1.

For the allreduce collective, we can apply the same strategy as that used for the reduce-scatter collective.

endofproof

Theorem 6. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The following redistributions can be implemented via simultaneous instances of the associated collective communicating over modes in* $\widetilde{\mathcal{D}} = \bigcup\limits_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}$.

Collective	*Redistribution*
Broadcast	$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), (w + j \cdot \operatorname{prod}(\mathbf{P}, \mathcal{E}))\right]$ $\downarrow$ $\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right]$
Reduce-to-one	$\widetilde{\sum_{\mathcal{K}}} \mathbf{A}^{\mathcal{A} \sqcup \mathcal{K}}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \widetilde{\mathcal{D}}^{(0)}, \ldots, \widetilde{\mathcal{D}}^{(M-1-v)}; \mathcal{E}, w\right]$ $\downarrow$ $\mathbf{B}^{\mathcal{A}}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}; \left(\mathcal{E} \sqcup \widetilde{\mathcal{D}}\right), (w + j \cdot \operatorname{prod}(\mathbf{P}, \mathcal{E}))\right]$

Proof: For the broadcast collective, we can apply the same strategy as that used in Theorem 2.

For the reduce-to-one collective, notice that the redistribution considered is equivalent (in terms of data moved) to first collecting all necessary elements from other processes, and then locally performing the reduction of elements. The pattern of data movement is then equivalent to that of the gather-to-one collective, thus we can apply the same strategy as that used in Theorem 3.

endofproof

A.4 Proofs of Balance

Here we prove that each of the rules defined in Figure 2.14 and Figure 2.15 correspond to balanced communications. We focus only on the all-to-all redistribution as all others rely on very similar arguments.

For these proofs, we must be assured that the distributions defined in the notation balance the data among processes; that is, each process is assigned approximately the same amount of data and all data entries are assigned to some process. An argument based on the fact that the function multi2linear is bijective can be made to show this.

A.4.1 All-to-all

Theorem 7. *Consider an order-M tensor $\mathbf{A}$ of size $\mathbf{I}$ distributed on an order-N processing grid $\mathbf{G}$ of size $\mathbf{P}$. The redistribution*

$$\mathbf{A}\left[\mathcal{D}^{(0)}, \ldots, \mathcal{D}^{(v-1)}, \mathcal{D}^{(v)} \sqcup \widetilde{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(M-1)} \sqcup \widetilde{\mathcal{D}}^{(M-1-v)}; \mathcal{E}, w\right]$$
$$\downarrow$$
$$\mathbf{A}\left[\mathcal{D}^{(0)} \sqcup \overline{\mathcal{D}}^{(0)}, \ldots, \mathcal{D}^{(v-1)} \sqcup \overline{\mathcal{D}}^{(v-1)}, \mathcal{D}^{(v)}, \ldots, \mathcal{D}^{(M-1)}; \mathcal{E}, w\right],$$

where $(v-1) \in \mathcal{R}(M)$ and communication occurs over processing mesh modes $\widetilde{\mathcal{D}} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)} = \bigcup_{m \in \mathcal{R}(M)} \widetilde{\mathcal{D}}^{(m)}$, is balanced.

Proof: Identical arguments can be made for all consistent permutations of this distribution and assignments of $\mathcal{E}$. As such, we focus on the simple case where $\mathcal{E} = \emptyset$ and we are performing the redistribution as defined above.

To prove this, we determine the maximum number of elements any contributor $\mathbf{p}$ can send to any target $\mathbf{q}$, thereby also specifying the maximum number of elements a target can receive from any contributor. We do this by considering the set of elements shared by $\mathbf{p}$ to $\mathbf{q}$ under the respective distributions by reasoning about the sets of tensor mode indices assigned to each process. The total number of transmitted elements from $\mathbf{p}$ to $\mathbf{q}$ can then be determined.

For this argument, we need only to consider the sets of indices assigned to the process $\mathbf{p}$ under both distributions. A simple argument can be made to show that this represents the maximum number of elements any contributor can send to any target.

Consider a mode $m < v$. In this case, the number of mode-m indices shared by $\mathbf{p}$

under both distributions is

$$\left|\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right)\cap\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\overline{\mathcal{D}}^{(m)}\right)\right|=\left|\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\overline{\mathcal{D}}^{(m)}\right)\right|$$

as

$$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\overline{\mathcal{D}}^{(m)}\right)\subseteq\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right).$$

Consider a mode $m\geq v$. In this case, the number of mode-m indices shared by $\mathbf{p}$ under both distributions is

$$\left|\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right)\cap\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\widetilde{\mathcal{D}}^{(m-v)}\right)\right|=\left|\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\widetilde{\mathcal{D}}^{(m-v)}\right)\right|$$

as

$$\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\sqcup\widetilde{\mathcal{D}}^{(m-v)}\right)\subseteq\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right).$$

Therefore, the greatest number of elements any process $\mathbf{p}$ may send to $\mathbf{q}$ is

$$\prod_{m\in\mathcal{R}(v)}\left|\mathcal{I}_m^{(\mathbf{p})}\left(\mathcal{D}^{(m)}\right)\right|\cdot\prod_{m\in\mathcal{R}(M-1-v)}\left|\mathcal{I}_{m+v}^{(\mathbf{p})}\left(\mathcal{D}^{(m+v)}\sqcup\widetilde{\mathcal{D}}^{(m)}\right)\right|$$

or

$$\prod_{m\in\mathcal{R}(v)}\left(\frac{I_m}{\text{prod}\left(\mathbf{P},\mathcal{D}^{(m)}\sqcup\overline{\mathcal{D}}^{(m)}\right)}\right)\cdot\prod_{m\in\mathcal{R}(M-1-v)}\left(\frac{I_{m+v}}{\text{prod}\left(\mathbf{P},\mathcal{D}^{(m+v)}\sqcup\widetilde{\mathcal{D}}^{(m)}\right)}\right) \tag{A.1}$$

by definition.

Let $\hat{p}=\text{prod}\left(\mathbf{P},\widetilde{\mathcal{D}}\right)$ be the number of processes involved in the communication.

The maximum number of elements assigned to any one process under $\overline{\boldsymbol{\mathcal{D}}}$ is

$$\overline{n} = \prod_{m \in \mathcal{R}(v)} \left(\frac{I_m}{\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m)} \sqcup \overline{\mathcal{D}}^{(m)}\right)} \right) \cdot \prod_{m \in \mathcal{R}(M-1-v)} \left(\frac{I_{m+v}}{\text{prod}\left(\mathbf{P}, \mathcal{D}^{(m+v)}\right)} \right). \quad \text{(A.2)}$$

Notice that the ratio between (A.1) and (A.2) is a factor

$$\hat{p} = \prod_{m \in \mathcal{R}(M-1-v)} \text{prod}\left(\mathbf{P}, \widetilde{\mathcal{D}}^{(m)}\right).$$

Using the interpretation of Figure 2.7, if all $\hat{p}$ processes can contribute approximately the same amount of data to all other processes, then the associated subvectors assigned to each process are approximately the same size. Therefore, the communication is balanced. **endofproof**

Bibliography

[1] OpenBLAS. `http://xianyi.github.com/OpenBLAS/`, 2012.

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[3] R. C. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[4] C. M. Andersen and R. Bro. Practical aspects of PARAFAC modeling of fluorescence excitation-emission data. *Journal of Chemometrics*, 17(4):200–215, 2003.

[5] B. W. Bader, M. W. Berry, and M. Browne. Discussion tracking in Enron email using PARAFAC. In Michael W. Berry and Malu Castellanos, editors, *Survey of Text Mining II*, pages 147–163. Springer London, 2008.

[6] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik. Reconstructing Householder vectors from Tall-Skinny QR. In

Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, pages 1159–1170. IEEE Computer Society, 2014.

[7] R. J. Bartlett. Many-body perturbation theory and coupled cluster theory for electron correlation in molecules. *Annual Review of Physical Chemistry*, 32(1):359–401, 1981.

[8] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005.

[9] N. H. F. Beebe and J. Linderberg. Simplifications in the generation and transformation of two-electron integrals in molecular calculations. *International Journal of Quantum Chemistry*, 12(4):683–705, 1977.

[10] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12. ACM, 2009.

[11] R. H. Bisseling. Parallel iterative solution of sparse linear systems on a transputer network. In A. E. Fincham and B. Ford, editors, *Parallel Computation*, volume 46 of *The Institute of Mathematics and its Applications Conference*, pages 253–271. Oxford University Press, Oxford, UK, 1993.

[12] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Proceedings of the*

28th Annual Hawaii International Conference on System Sciences, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[13] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 298–309, 1997.

[14] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, and M. Valiev. NWChem, a computational chemistry package for parallel computers, version 6.3.

[15] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozeman, MT, USA, 1969.

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[17] J. Choi. A fast scalable universal matrix multiplication algorithm on distributed-memory concurrent computers. In *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, pages 310–314. IEEE Computer Society, 1997.

[18] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[19] J. Choi, D. W. Walker, and J. Dongarra. PUMMA: Parallel Universal Ma-

trix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.

[20] J. Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in Ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[21] T. D. Crawford and H. F. Schaefer. *An Introduction to Coupled Cluster Theory for Computational Chemists*. John Wiley & Sons, Inc., 2007.

[22] Cray. CrayXC30. `http://www.cray.com/Assets/PDF/products/xc/CrayXC30PDCProductBrief.pdf`, 2013.

[23] E. Deumens, V. F. Lotrich, A. Perera, M. J. Ponton, B. A. Sanders, and R. J. Bartlett. Software design of ACES III with the super instruction architecture. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(6):895–901, 2011.

[24] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[25] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: Have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[26] A. Einstein. Die Grundlage der allgemeinen Relativitätstheorie. *Annalen der Physik*, 354:769–822, 1916.

[27] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, 1988.

[28] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, 2008.

[29] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, 2008.

[30] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[31] T. Hazan, S. Polak, and A. Shashua. Sparse image coding using a 3D non-negative tensor factorization. In *Proceedings of the 10th IEEE International Conference on Computer Vision*, volume 1, pages 50–57, 2005.

[32] T. Helgaker, P. Jørgensen, and J. Olsen. *Molecular Electronic Structure Theory*. John Wiley & Sons, Ltd., Chichester, 2000.

[33] B. Hendrickson, E. Jessup, and C. Smith. Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM Journal on Scientific Computing*, 20(3):1132–1154, 1998.

[34] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *International Journal of High Speed Computing*, 7:73–88, 1995.

[35] S. Hirata. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body per-

turbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[36] S. Hirata. Higher-order equation-of-motion coupled-cluster methods. *The Journal of Chemical Physics*, 121(1):51–59, 2004.

[37] S. Hirata. Symbolic algebra in quantum chemistry. *Theoretical Chemistry Accounts*, 116(1-3):2–17, 2006.

[38] S. Huss-Lederman, E. Jacobson, and A. Tsao. Comparison of scalable parallel matrix multiplication libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 142–149, 1993.

[39] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[40] Intel. Reference manual for Intel Math Kernel Library; update 2, 2015.

[41] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[42] H. A. L. Kiers, J. M. F. ten Berge, and R. Rocci. Uniqueness of three-mode factor models with sparse cores: The 3x3x3 case. *Psychometrika*, 62(3):349–374, 1997.

[43] J. Kim, W.J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 77–88. IEEE Computer Society, 2008.

[44] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[45] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[46] K. Kowalski, S. Krishnamoorthy, R. M. Olson, V. Tipparaju, and E. Apra. Scalable implementations of accurate excited-state coupled cluster theories: Application of high-level methods to porphyrin-based systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[47] S.A. Kucharski and R. J. Bartlett. Recursive intermediate factorization and complete computational linearization of the coupled-cluster single, double, triple, and quadruple excitation equations. *Theoretica Chimica Acta*, 80(4-5):387–405, 1991.

[48] P.-W. Lai. *A Framework for Performance Optimization of Tensor Contraction Expressions*. PhD thesis, The Ohio State University, 2014.

[49] H.-J. Lee, J. P. Robertson, and J. A. B. Fortes. Generalized Cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 44–51. ACM, 1997.

[50] T. J. Lee and J. E. Rice. An efficient closed-shell singles and doubles coupled-cluster method. *Chemical Physics Letters*, 150(6):406 – 415, 1988.

[51] J. G. Lewis and R. A. van de Geijn. Implementing matrix-vector multiplication and conjugate gradient algorithms on distributed memory multicomputers. In *Proceedings of Supercomputing*, 1993.

[52] C.-Y. Lin, Y.-C. Chung, and J.-S. Liu. Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme. *IEEE Transactions on Computers*, 52(12):1640–1646, 2003.

[53] B. Marker. *From Domain Knowledge to Optimized Program Generation.* PhD thesis, The University of Texas at Austin, 2014.

[54] B. Marker, D. Batory, and R. van de Geijn. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, 27(4):439–452, 2013.

[55] B. Marker, D. Batory, and R. van de Geijn. Code generation and optimization of distributed-memory dense linear algebra kernels. *Procedia Computer Science*, 18(0):1282 – 1291, 2013.

[56] B. Marker, D. Batory, and R. van de Geijn. Understanding performance stairs: Elucidating heuristics. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 301–312. ACM, 2014.

[57] B. Marker, J. Poulson, D. S. Batory, and R. A. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In M. Daydé, O. Marques, and K. Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851, pages 362–378. Springer Berlin Heidelberg, 2013.

[58] B. Marker, M. D. Schatz, D. A. Matthews, I. Dillig, R. van de Geijn, and D. Batory. DxTer: An extensible tool for optimal dataflow program generation. Technical Report TR-15-03, The University of Texas at Austin, 2015.

[59] D. A. Matthews. *Non-orthogonal Spin-adaptation and Applications to Coupled Cluster up to Quadruple Excitations*. PhD thesis, The University of Texas at Austin, 2014.

[60] NERSC. NERSC Edison configuration. `https://www.nersc.gov/users/computational-systems/edison/configuration/`.

[61] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[62] J Noga and R. J. Bartlett. The full CCSDT model for molecular electronic structure. *The Journal of Chemical Physics*, 86(12):7041–7050, 1987.

[63] D. Ozog, J. R. Hammond, J. Dinan, P. Balaji, S. Shende, and A. Malony. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *Proceedings of the 42nd International Conference on Parallel Processing*, pages 30–39. ACM, 2013.

[64] S. Parker. BG/Q architecture. `https://www.alcf.anl.gov/files/bgq-arch_0.pdf`, 2013.

[65] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):13:1–13:24, 2013.

[66] G. D. Purvis and R. J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76(4):1910–1918, 1982.

[67] M. Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232– 275, 2005.

[68] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479 – 483, 1989.

[69] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan. CAST: Contraction Algorithm for Symmetric Tensors. In *Proceedings of the 43rd International Conference on Parallel Processing*, pages 261–272. IEEE, 2014.

[70] S. Rajbhandari, A. Nikam, P.-W. Lai, K. Stock, S. Krishnamoorthy, and P. Sadayappan. A communication-optimal framework for contracting distributed tensors. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 375–386. IEEE Press, 2014.

[71] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton. A block-oriented language and runtime system for tensor algebra with very large arrays. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[72] M. D. Schatz, T. M. Low, R. A. van de Geijn, and T. G. Kolda. Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors. *SIAM Journal on Scientific Computing*, 36(5):C453–C479, 2014.

[73] M. D. Schatz, J. Poulson, and R. van de Geijn. Parallel matrix multiplciation: 2D and 3D. Technical Report TR-12-13, The University of Texas at Austin, Department of Computer Sciences, 2012.

[74] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.

[75] G. E. Scuseria, C. L. Janssen, and H. F. Schaefer. An efficient reformulation of the closed-shell coupled cluster single and double excitation (CCSD) equations. *The Journal of Chemical Physics*, 89(12):7382–7387, 1988.

[76] G. E. Scuseria, A. C. Scheiner, T. J. Lee, J. E. Rice, and H. F Schaefer. The closed-shell coupled cluster single and double excitation (CCSD) model for the description of electron correlation. a comparison with configuration interaction (CISD) results. *The Journal of Chemical Physics*, 86(5):2881–2890, 1987.

[77] I. Shavitt. The method of configuration interaction. In H. F. III Schaefer, editor, *Methods of Electronic Structure Theory*, volume 3 of *Modern Theoretical Chemistry*, pages 189–275. Springer US, 1977.

[78] I. Shavitt and R. J. Bartlett. *Many-Body Methods in Chemistry and Physics*. Cambridge University Press, 2009.

[79] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*,

volume 6853 of *Lecture Notes in Computer Science*, pages 90–109. Springer Berlin Heidelberg, 2011.

[80] E. Solomonik, D. Matthews, J. R. Hammond, and J. Demmel. Cyclops Tensor Framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing*, pages 813–824. IEEE Computer Society, 2013.

[81] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[82] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, pages 23–32. ACM, 2014.

[83] IBM Blue Gene team. Design of the IBM Blue Gene/Q compute chip. *IBM Journal of Research and Development*, 57(1/2):1:1–1:13, 2013.

[84] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[85] R. van de Geijn and J. L. Traff. *Encyclopedia of Parallel Computing*, pages 318–327. Springer, 2011.

[86] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The

MIT Press, 1997.

[87] R. A. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[88] F. G. van Zee, T. Smith, B. Marker, T. M. Low, R. A. van de Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. A. Gunnels, and L. Killough. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, (2015) (accepted).

[89] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 25:1–25:12. ACM, 2013.

[90] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27. IEEE Computer Society, 1998.