# Communication Avoidance for Algorithms with Sparse All-to-all Interactions

*Penporn Koanantakool*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-221
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-221.html

December 15, 2017

Copyright © 2017, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I acknowledge the Fulbright Scholarship, the Department of Defense, and the Department of Energy, Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory contract DE-AC02-05CH11231 and DOE contract DE-SC0008700 at UC Berkeley. This research used resources of the National Energy Research Scientific Computing Center, the Argonne Leadership Computing Facility, and the Oak Ridge Leadership Computing Facility, all supported by the Office of Science of the U.S. DOE contracts DE-AC02-05CH11231, DE-AC02-06CH11357, and DE-AC05-00OR22725, respectively. The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

**Communication Avoidance for Algorithms with Sparse All-to-all Interactions**

by

Penporn Koanantakool

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Chair
Professor James Demmel
Assistant Professor Lin Lin

Fall 2017

The dissertation of Penporn Koanantakool, titled Communication Avoidance for Algorithms with Sparse All-to-all Interactions, is approved:

Chair ______________________________ Date ______________

______________________________ Date ______________

______________________________ Date ______________

University of California, Berkeley

**Communication Avoidance for Algorithms with Sparse All-to-all Interactions**

Copyright 2017
by
Penporn Koanantakool

## Abstract

Communication Avoidance for Algorithms with Sparse All-to-all Interactions

by

Penporn Koanantakool

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine Yelick, Chair

In parallel computing environments from multicore systems to cloud computers and supercomputers, data movement is the dominant cost in both running time and energy usage. Even worse, hardware trends suggest that the gap between computing and data movement, both in memory systems and interconnect networks, will continue to grow. Minimizing communication is therefore necessary in devising scalable parallel algorithms. This work discusses parallelizing kernels in applications ranging from chemistry and cosmology to machine learning.

We have developed new communication-avoiding algorithms for problems with all-to-all interactions such as many-body and matrix computations, taking into account their sparsity patterns, either from cutoff distance, symmetry, or data sparsity. Our algorithms are communication-efficient (some are provably optimal) and scalable to tens of thousands of processors, exhibiting orders of magnitude speedup over more commonly used algorithms.

These all-to-all computational patterns arise in scientific simulations and machine learning. The last part of the thesis will present a case study of communication-avoiding sparse-dense matrix multiplication as used in graphical model structure learning. The resulting high-performance sparse inverse covariance matrix estimation algorithm enables processing high-dimensional data with arbitrary underlying structures at a scale that was previously intractable, e.g., 1.28 million dimensions (over 800 billion parameters) in under 21 minutes on 24,576 cores of a Cray XC30. Our method is used to automatically estimate the underlying functional connectivity of the human brain from resting-state fMRI data. The results show good agreement with a state-of-the-art clustering, which used manual intervention, from the neuroscience literature.

To my family.

# Contents

**Contents** ii

**List of Figures** iv

**List of Tables** vi

**1 Introduction** 1
- 1.1 Communication Avoidance . . . 2
- 1.2 Communication Optimality . . . 3
- 1.3 All-to-all Algorithms . . . 5
- 1.4 Contributions . . . 7
- 1.5 Outline . . . 8

**2 Preliminaries** 9
- 2.1 Performance Model . . . 9
- 2.2 Communication Lower Bounds . . . 9
- 2.3 Processor Topology and Replication . . . 10
- 2.4 Communication Operations . . . 12
- 2.5 Matrix Notation . . . 12

**3 2-way N-Body** 13
- 3.1 Communication Lower Bounds . . . 14
- 3.2 Previous Work . . . 18
- 3.3 Interactions with No Cutoff Radius . . . 21
- 3.4 Finite Cutoff Distance . . . 25
- 3.5 Taking Advantage of Symmetry . . . 33
- 3.6 Conclusions . . . 34

**4 $k$-way N-Body** 36
- 4.1 Background and Previous Work . . . 37
- 4.2 Communication Lower Bounds . . . 40
- 4.3 Algorithms . . . 41
- 4.4 Performance Results . . . 57

4.5 Extension for Cutoff Distance . . . . . . . . . . 61
4.6 Conclusions . . . . . . . . . . 66

**5 Sparse-Dense Matrix-Matrix Multiplication 68**
5.1 Background and Previous Work . . . . . . . . . . 68
5.2 Communication Lower Bounds . . . . . . . . . . 70
5.3 Algorithms . . . . . . . . . . 71
5.4 Performance Results . . . . . . . . . . 81
5.5 Consecutive Multiplications . . . . . . . . . . 88
5.6 Conclusions . . . . . . . . . . 90

**6 Generalizing 1.5D Matrix Multiplication 93**
6.1 Mixing Replication Factors in 1.5D Matrix Multiplication . . . . . . . . . . 93
6.2 Matrix Transpose . . . . . . . . . . 98
6.3 Symmetric Matrix Multiplication . . . . . . . . . . 102

**7 Sparse Inverse Covariance Matrix Estimation 105**
7.1 Background and Previous Work . . . . . . . . . . 105
7.2 HP-CONCORD . . . . . . . . . . 108
7.3 Experimental Results . . . . . . . . . . 113
7.4 Massive-scale Structure Learning from fMRI Data . . . . . . . . . . 121
7.5 Expanded Set of Results . . . . . . . . . . 125
7.6 Conclusions . . . . . . . . . . 145

**8 Conclusions 146**
8.1 Future Work . . . . . . . . . . 148

**Bibliography 150**

# List of Figures

1.1 How 2D, 2.5D, and 3D algorithms partition the matrix multiplication iteration space between $p$ processors . . . . 4

2.1 Mappings of processor meshes and data, without and with replication . . . . 11

3.1 The force matrix for the all-pairs N-body problem . . . . 15
3.2 Communication-avoiding all-pairs algorithm . . . . 22
3.3 Communication-avoiding all-pairs algorithm: cost breakdown . . . . 24
3.4 Communication-avoiding all-pairs algorithm: strong scaling . . . . 26
3.5 Communication-avoiding 2-body algorithm with 1D cutoff . . . . 27
3.6 Communication-avoiding 2-body algorithm with 2D cutoff . . . . 28
3.7 Communication-avoiding 2-body algorithm with cutoff: cost breakdown . . . . 30
3.8 Load imbalance introduced with the reflective boundary conditions . . . . 31
3.9 Communication-avoiding 2-body algorithm with cutoff: strong scaling . . . . 32
3.10 Communication-avoiding all-pairs algorithm with force symmetry . . . . 34

4.1 The force cube for the all-triplets 3-body problem . . . . 38
4.2 Uniformly distributed all-unique particle triplets . . . . 39
4.3 The all-unique-triplets 3-body algorithm . . . . 45
4.4 The number of ways to generate offset patterns . . . . 46
4.5 The triplets each processor in the all-unique-triplets algorithm computes. . . . . 51
4.6 All-unique-triplets algorithm for $n = 24$ and $p = 6$: rounds 1-4 . . . . 52
4.7 All-unique-triplets algorithm for $n = 24$ and $p = 6$: rounds 5-7 . . . . 53
4.8 All-unique-triplets algorithm for $n = 24$ and $p = 6$: rounds 8-10 . . . . 54
4.9 Communication-avoiding all-unique-triplets algorithm: cost breakdown . . . . 59
4.10 Communication-avoiding all-unique-triplets algorithm: strong scaling . . . . 60
4.11 $k$-body cutoff distance in a 1D-space simulation . . . . 62
4.12 Redundancy in the $k$-body computation with cutoff distance . . . . 62

5.1 The matrix multiplication iteration space . . . . 70
5.2 Processor mesh layouts for 1.5D and 2.5D algorithms . . . . 73
5.3 Example computations of the 1.5D block column A matrix multiplication . . . . 76

5.4 Configurations that each communication-avoiding matrix multiplication algorithm has the best theoretical bandwidth cost . . . . . 80
5.5 Communication-avoiding matrix multiplication cost breakdown comparison for a square Erdős-Rényi matrix of size $n$ = 65,536 with 41 nonzeroes per row on $p$ =3,072 cores . . . . . 82
5.6 Single-node MKL matrix multiplication efficiency for Figure 5.5 . . . . . 82
5.7 Communication-avoiding matrix multiplication algorithm strong scaling for an Erdős-Rényi matrix of size $n$ = 65,536 with 1% nonzeroes . . . . . 83
5.8 Single-node MKL matrix multiplication efficiency for Figure 5.7 . . . . . 83
5.9 Weak Scaling of square Erdős-Rényi and Graph500 matrices with fixed $d = 164$ nonzeroes per row . . . . . 85
5.10 Nonzero structures of the real-world matrices. . . . . . 86
5.11 Strong scaling and cost-breakdown results from multiplying the Mouse gene network matrix (*mouse_gene.mtx*) (45,101 $\times$ 45,101, 1.42% nonzeroes) with a dense 45,101 $\times$ 45,101 matrix . . . . . 87
5.12 Strong scaling and cost-breakdown results from multiplying the Simplicial complexes matrix (*shar_te2-b2.mtx*) (200,200 $\times$ 17,160, 0.0175% nonzeroes) with a dense 17,160 $\times$ 200,200 matrix . . . . . 87
5.13 Strong scaling and cost-breakdown results from multiplying the Stochastic linear programming problem matrix (*stormg2-215.mtx*) (66,185 $\times$ 172,431, 0.0038% nonzeroes) with a dense 172,431 $\times$ 66,185 matrix . . . . . 88
5.14 Configurations that each communication-avoiding matrix multiplication algorithm has the best theoretical bandwidth cost when there are 10 and 20 consecutive multiplications . . . . . 90

6.1 The 1.5D Inner-A matrix multiplication algorithm . . . . . 95
6.2 The 1.5D Inner-B matrix multiplication algorithm . . . . . 95
6.3 Transposing the resulting matrix from the 1.5D Inner-ABC algorithm . . . . . 100
6.4 Transposing the resulting matrix from the 1.5D Inner-A algorithm . . . . . 100
6.5 The 1.5D Inner-A algorithm for multiplying symmetric matrices. . . . . . 103

7.1 Distributed operations of two HP-CONCORD variants . . . . . 111
7.2 Cov vs. Obs on synthetic chain and random graphs . . . . . 113
7.3 The time Cov and Obs took in each line search iteration . . . . . 116
7.4 Obs' cost breakdown for chain graph on 256 nodes . . . . . 117
7.5 More Obs' cost breakdowns for Figure 7.4 . . . . . 118
7.6 BigQUIC vs. HP-CONCORD on chain and random graphs . . . . . 120
7.7 Left and right brain hemisphere clusterings from Glasser et al. . . . . . 124
7.8 The sparsity pattern of an HP-CONCORD estimate recovers locality . . . . . 126

# List of Tables

1.1 Communication optimality and empirical speedups of previous *communication-avoiding* algorithms . . . 5

2.1 Common notation . . . 11
2.2 Common communication operations and their costs . . . 12

5.1 Communication-avoiding matrix multiplication algorithm cost comparison . . . 79
5.2 The best replication factors ($c$) for each strong or weak scaling graph in this chapter 84

6.1 List of all possible 1D algorithms . . . 94
6.2 Communication costs of the 1.5D matrix multiplication algorithms . . . 97

7.1 Experimental result details for Figure 7.2 . . . 114
7.2 Numbers of iterations BigQUIC and HP-CONCORD took to converge in the chain and random graphs experiments . . . 120
7.3 Our best human brain clusterings relative to Glasser et al.'s . . . 124
7.4 List of Tables containing various clustering experimental results . . . 126
7.5 Sparsity patterns of the whole human brain . . . 127
7.6 Sparsity patterns of the left hemisphere . . . 128
7.7 Sparsity patterns of the right hemisphere . . . 129
7.8 Degree matrices of the left hemisphere . . . 130
7.9 Degree matrices of the right hemisphere . . . 131
7.10 Coarse left hemisphere clusterings generated by HP-CONCORD followed by the persistent homology method. . . . 132
7.11 Coarse right hemisphere clusterings generated by HP-CONCORD followed by the persistent homology method. . . . 133
7.12 Fine left hemisphere clusterings generated by HP-CONCORD followed by the persistent homology method. . . . 134
7.13 Fine right hemisphere clusterings generated by HP-CONCORD followed by the persistent homology method. . . . 135
7.14 Coarse left hemisphere clusterings generated by HP-CONCORD followed by the Louvain method. . . . 136

7.15 Coarse right hemisphere clusterings generated by HP-CONCORD followed by the Louvain method. . . . . 137
7.16 Fine left hemisphere clusterings generated by HP-CONCORD followed by the Louvain method. . . . . 138
7.17 Fine right hemisphere clusterings generated by HP-CONCORD followed by the Louvain method. . . . . 139
7.18 Coarse left and right hemisphere clusterings generated by thresholding the sample covariance matrix followed by the Louvain method . . . . . 140
7.19 Fine left and right hemisphere clusterings generated by thresholding the sample covariance matrix followed by the Louvain method . . . . . 140
7.20 The Jaccard scores for Table 7.10 . . . . . 141
7.21 The Jaccard scores for Table 7.11 . . . . . 141
7.22 The Jaccard scores for Table 7.12 . . . . . 142
7.23 The Jaccard scores for Table 7.13 . . . . . 142
7.24 The Jaccard scores for Table 7.14 . . . . . 143
7.25 The Jaccard scores for Table 7.15 . . . . . 143
7.26 The Jaccard scores for Table 7.16 . . . . . 144
7.27 The Jaccard scores for Table 7.17 . . . . . 144

8.1 Communication optimality and empirical speedups of our *communication-avoiding* algorithms . . . . . 147

## Acknowledgments

I would like to thank my advisor, Kathy Yelick, for all her guidance, support, and kindness. She taught me to always look for the big picture and gave the best advice on anything from research to editing. Being at Berkeley opened my world; I will be forever grateful for the opportunity to work with her and all amazing people here, and for all the invaluable experiences I have gained. I would also like to thank the rest of my dissertation and qualifying examination committee members, Jim Demmel, Lin Lin, and Joey Gonzalez for their insightful feedback and suggestions.

I would like to acknowledge my collaborators with whom I have worked most closely, Evangelos Georganas, Michael Driscoll, Sang-Yun Oh, and Aydın Buluç. I have learned a lot from them. Additionally, I would like to thank the rest of my collaborators, many of whom contributed to the work in this thesis, Alnur Ali, Ariful Azad, Erin Carson, Omar Demerdash, Laura Grigori, Nicholas Knight, Dmitriy Morozov, Leonid Oliker, Oded Schwartz, Harsha Simhardri, Edgar Solomonik, Samuel Williams, and Yili Zheng. Many thanks to Aditya Devarakonda, Marquita Ellis, past and present members of the BeBOP group, UPC/DEGAS members, and ParLab/ASPIRE members for wonderful discussions. I also appreciate the superb help from the ParLab/ASPIRE/LBNL/EECS staff, Roxana Infante, Tamille Chouteau, Kostadin Ilov, Tara White, Lisa Theobald, Shirley Salanio, and Audrey Sillers. A big thank you to my family and friends for positivity and support.

This work was supported in part by the Fulbright Scholarship, the Department of Defense, and the Department of Energy, Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory contract DE-AC02-05CH11231 and DOE contract DE-SC0008700 at UC Berkeley. This research used resources of the National Energy Research Scientific Computing Center, the Argonne Leadership Computing Facility, and the Oak Ridge Leadership Computing Facility, all supported by the Office of Science of the U.S. Department of Energy under contracts No. DE-AC02-05CH11231, DE-AC02-06CH11357, and DE-AC05-00OR22725, respectively. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

# Chapter 1

# Introduction

Data movement, whether between levels of memory hierarchy or between processors over the network, is the most expensive operation in both time and energy, and hardware trends suggest the gap between its cost and the cheaper arithmetic cost will only grow [64, 77, 11]. Data movement is especially costly in a distributed memory supercomputer, cluster, or cloud setting where the cost of interprocessor communication is at least 4 orders of magnitude higher than the cost of a floating-point operation (flop). One approach to bridging the growing performance gap, along with addressing the increasing energy concerns [61], is the design of new algorithms that provably minimize communication.

This thesis presents a set of algorithms that are designed to avoid network communication in a parallel environment. The algorithms address a class of problems that require some form of all-to-all style computation that includes N-body methods and matrix multiplication, and also scenarios that are in some sense sparse due to features such as distance cutoff or matrix sparsity. The algorithms use a common strategy of data replication to avoid communication, and they are parameterized over the number of replicas of a given input or output data structure. The work includes formal communication complexity analysis, communication lower bounds, and experimental analysis for new parallel implementations on high performance computing systems. Many of the algorithms presented here are communication-optimal, and some are one to two orders of magnitude faster than algorithms commonly used in practice.

While most algorithm complexity analysis typically focuses on computational costs as measured by arithmetic or logical operations, there is also significant prior work in modeling communication costs and communication-avoiding algorithms. In addition, there are both automatic and manual approaches to minimizing communication. In the remainder of this chapter we will summarize the most relevant areas of prior work, starting with communication-avoiding algorithms, and then covering lower bounds and communication optimality. We then formalize the notion of all-to-all style algorithms and sparsity that are addressed in our work. Finally, we highlight some of the major contributions in this thesis and give a roadmap to the rest of the thesis.

## 1.1 Communication Avoidance

Computing speeds have been improving at a rate much faster than those of memory and network. The exponentially-growing performance discrepancy has been discussed for decades. There has been extensive research on maximizing data locality, both manually and automatically. This section gives a brief overview of previous efforts, starting from serial programs, to parallel shared-memory programs, and finally parallel distributed-memory programs, which are the domain of our work.

**Serial programs.** Typical memory optimizations include register tiling (loop unroll-and-jam), cache tiling (loop blocking), loop skewing, strip-mining, loop interchange, array padding, memory alignment, prefetching, etc. These processes are tedious and hardware-dependent. The research to delegate these tasks to *optimizing compilers* began as soon as the 1960s [72, 181, 7, 152, 36].

**Parallel shared-memory programs.** As compilers started to vectorize and parallelize codes [112, 115], the transformations were adapted to support shared memory programs as well. *Data parallelism* where processors split the data equally and update the part they own is the most common parallelization approach, but it is not always communication-efficient. *Iteration space tiling* [180, 179] performs dependency analysis and applies loop transformations and tiling to promote appropriate concurrency and data reuse. The name iteration space tiling reflects the fact that tiling consecutive loops corresponds to geometrically tiling (partitioning) the *iteration space* consisting of all possible tuples of loop indices. There are two main questions: what loop transformations to perform [124, 137], and what are the best tile sizes [114, 49, 157, 108]. While iteration space tiling can lead to algorithms with replication, such as those presented in this thesis, the prior work in this area did not include lower bound analysis or distributed-memory operations.

Current software tools are built on the premise that computing is the most expensive component [170], but the increasing need for data locality prompts the developments of many software extensions and also new programming models. OpenMP [53] does not have a native way to specify data distribution or give task scheduling guidelines based on locality, but there are extensions that do [140, 94]. Cilk [32] and Intel Threading Building Blocks (TBB) [156] are parallel runtime extensions to the C and C++ language, respectively. They both use work-stealing approach that balance the load well, but lack the consideration of data reuse. Locality supports are again being implemented by third parties as extensions [1, 125, 90].

**Parallel distributed-memory programs.** Distributed-memory compiler work came to attention in the late 1980s and is still ongoing [37, 155, 154, 85, 27], however, they have not been widely adopted yet. Most parallelizations and optimizations are still done by hand, using distributed parallel programming languages or runtimes such as MPI [89], UPC [39],

UPC++ [17], X10 [46], Chapel [43], Charm++ [101], etc., all equipped with features to manage data distribution and locality.

Optimization techniques are different for each type of applications, for example, tree-based N-body simulations [183] replicate the top levels of the tree structure, particle-in-cell simulations [128] replicate grid cells, stencil computations [55] replicate particles in the ghost zones, graph algorithms use graph partitioning to minimize communication [35, 78], sparse-matrix dense-vector multiplications and sparse-sparse matrix multiplications use hypergraph partitioning [42, 22], etc. Again, while the techniques are known to improve performance, they lack rigorous proofs of optimality and memory/performance tradeoffs presented in this thesis.

## 1.2 Communication Optimality

Last section described various approaches to *decrease* communication, this section discusses algorithms that *provably minimize* communication. There are two costs associated with communication: latency and bandwidth. To assess how well an algorithm avoids communication, we derive communication lower bounds on latency and bandwidth and compare to the algorithm costs.

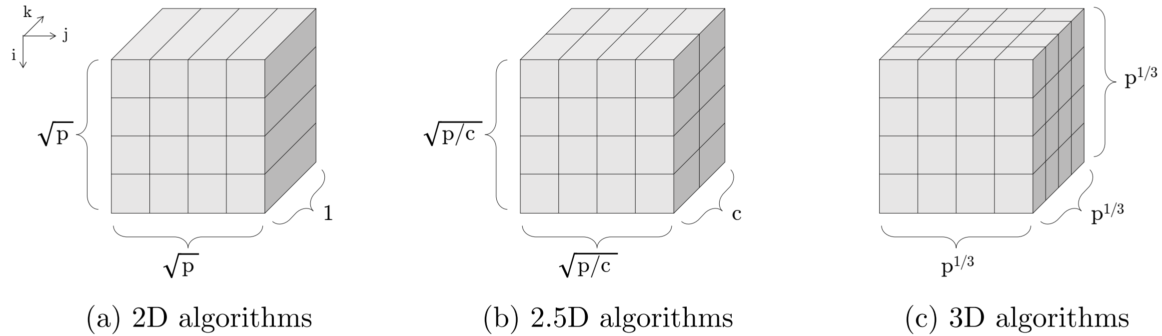
**Definition 1.2.1.** *Communication Optimality.* An algorithm is communication-optimal if its latency and bandwidth costs match their corresponding lower bounds for the problem.

Our work is motivated by recent developments of communication lower bounds and communication-optimal algorithms in linear algebra, starting with the matrix multiplication problem $C^{m \times n} = A^{m \times q} B^{q \times n}$,

$$\begin{aligned} &\textbf{for } i = 0 : m,\ j = 0 : n,\ k = 0 : q \\ &\quad c_{ij} \mathrel{+}= a_{ik} \times b_{kj}, \end{aligned}$$

where lowercase letters refer to matrix elements.

In 1981, Hong and Kung introduced communication lower bounds for sequential matrix multiplication [98]. Aggarwal et al. gave the parallel matrix multiplication lower bounds in 1990, assuming unlimited memory, together with an algorithm that attains the bounds [4]. These memory-independent bounds are reproduced separately by Irony et al. [96] in 2004, where they also presented memory-dependent lower bounds, although none of the algorithms at that time attained the lower bounds for all possible memory sizes. The existing algorithms are either optimal for *minimal* memory size, i.e., when each processor only has enough memory to store $1/p$ of data, such as Cannon's [38] and SUMMA [80] algorithms (also called 2D algorithms), or optimal for *maximal* memory size, requiring each processor to be able to store $1/p^{2/3}$ of each matrix such as the so-called 3D algorithms [4, 3]. In 2011, Solomonik and Demmel introduced the 2.5D *communication-avoiding* algorithm [167] which stores $c/p$ of data in memory. $c$ is a tunable parameter ranging between 1 to $\sqrt[3]{p}$, allowing the algorithm to interpolate between 2D ($c = 1$) and 3D ($c = \sqrt[3]{p}$) algorithms. By selecting the largest $c$




(a) 2D algorithms (b) 2.5D algorithms (c) 3D algorithms

Figure 1.1: How 2D, 2.5D, and 3D algorithms partition the matrix multiplication iteration space between $p$ processors. 2.5D algorithms interpolate between the 2D and 3D algorithms with its tunable parameter $c$ (replication factor).

that fits in memory, the algorithm was proven communication-optimal for any memory size and any number of processors. Compared to the non-replicating version (minimal memory, i.e., $c = 1$), the 2.5D algorithm decreases latency and bandwidth costs by factors of $c^{3/2}$ and $c^{1/2}$, respectively.

Replication can be interpreted as partitioning the iteration space rather than the data themselves among processors, similar to iteration space tiling in compiler optimization. Chamberlain et al. [44] discussed this in distributed-memory setting without being application-specific in 1999, although their goal was to expose more parallelism. Figure 1.1 shows how the 2D, 2.5D, and 3D algorithms partition (tile) the iteration space, each $(i, j, k)$ coordinate representing the calculation of $c_{ij} \mathrel{+}= a_{ik}b_{kj}$. 2D algorithms partition the $C$ matrix among the processors. Each processor is responsible for calculating all computations related to the part of $C$ that it owns. 2.5D and 3D algorithms partition the iteration space among the processors rather than the data.

**Definition 1.2.2.** *Communication-avoiding algorithms* are algorithms that replicate data and reformulate computation to *asymptotically (in c)* reduce communication.

The lower bounds for matrix multiplication were generalized to a larger class of dense linear algebra problems [23], and later to any loop nests with array subscripts that are affine functions of loop indices [47, 107, 48]. Many more *communication-avoiding* algorithms for linear algebra or three-nested-loop applications emerged, most are provably communication-optimal, achieving large improvements over common non-replicating algorithms, shown in Table 1.1. (See Ballard et al.'s survey [20] for more details on lower bounds and optimality.) This motivates us to apply the communication-avoiding techniques to other applications. Our work focuses on algorithms with *all-to-all* interactions since they require a significant amount of communication by nature.

| Computation | Algorithm | Communication Optimality | Largest speedup reported in the cited paper(s) |
|---|---|---|---|
| Dense matmul | 3D [3, 135, 44] | Optimal only in maximal memory case | N/A |
| | 2.5D [165] | Optimal for square matrices | 12× over non-replicating version |
| | CARMA [60] (recursive) | Optimal for all matrices | 141× over ScaLAPACK (rectangular multiplications) |
| Strassen's dense matmul | CAPS [21] | Optimal | 2.84× over non-replicating version |
| Sparse matmul | 3D-SpGEMM [25]+[14] | Optimal when matrix sparsities are similar (Erdős-Rényi matrices) | 3× over Trillinos |
| Cholesky | Block recursion [171] | Optimal | N/A |
| | 2.5D [81] | Optimal | ∼1× over non-replicating version |
| LU | Block recursion [172] | Optimal | N/A |
| | 2.5D [167] | Optimal | 1.37× over non-replicating version |
| Tall-skinny QR | TSQR-HR [24] | Optimal | 3.6× over ScaLAPACK |
| Symmetric band reduction | CASBR [19] | N/A (no known lower bounds) | 6× over PLASMA |
| Krylov subspace methods | CA-KSM [40] | N/A (no tight lower bounds) | 4.2× over non-replicating version |
| All-pairs shortest paths | 2.5D DC-APSP [166] | Optimal | 6.2× over non-replicating version |

Table 1.1: Communication optimality and empirical speedups of *communication-avoiding* algorithms, i.e., algorithms that can exploit memory larger than the minimal required ($1/p$ of data) to *asymptotically* communicate less. Dense matmul and sparse matmul refer to dense-dense and sparse-sparse matrix-matrix multiplication, respectively. Maximal memory means the maximum benificial memory usage according to the memory-independent lower bounds ($1/p^{2/3}$ of data for matmul). Speedups reported are compared to either non-replicating algorithms or existing software libraries. These numbers are from the cited papers only and are not meant to be comprehensive as later literature may report larger speedups. All other algorithms listed communication-optimal are optimal for all memory sizes, except for the 3D matmul algorithms in the first row.

## 1.3 All-to-all Algorithms

By all-to-all algorithms, we refer to nested loops where each output data point requires inputs subscripted with all possible values of all other loop indices that are not indexing the output.

**Definition 1.3.1.** *An all-to-all algorithm* is of the form,

$$\begin{aligned}
&\textbf{for } i_0 = 0 : I_0,\ i_1 = 0 : I_1,\ \ldots,\ i_{y-1} = 0 : I_{y-1}\\
&\quad \textbf{for } i_y = 0 : I_y,\ i_{y+1} = 0 : I_{y+1},\ \ldots, i_{z-1} = 0 : I_{z-1}\\
&\qquad Y(i_0, i_1, \ldots, i_{y-1}) = f\left(X_0(\phi_0(\mathcal{I})), X_1(\phi_1(\mathcal{I})), \ldots, X_{m-1}(\phi_{m-1}(\mathcal{I}))\right),
\end{aligned}$$

where

- $Y$ is a $y$-dimensional output array,
- $X_j$ is an $x_j$-dimensional input array,
- $i_0, i_1, \ldots, i_{z-1}$ are loop indices, $i_0, i_1, \ldots, i_{z-1}, I_0, I_1, \ldots, I_{z-1} \in \mathbb{N}$.

- $\mathcal{I} = (i_0, i_1, \ldots, i_{z-1})$ is a tuple of all loop indices,
- $\phi_j$: $\mathbb{N}^z \rightarrow \mathbb{N}^{x_j}$ is a function mapping a $z$-tuple of all indices to an $x_j$-tuple to index $X_j$,
- $f$: $\mathbb{R}^m \rightarrow \mathbb{R}$ is a function calculating an output from $m$ input values, assuming its output fully depends on all $m$ input values and is not trivially some constant for all arguments,
- $\mathbb{N}$ is the set of nonnegative integers (natural numbers), and
- $\mathbb{R}$ is the set of real numbers.

Many important scientific problems are all-to-all. The N-body problem, one of the seven most common scientific kernels to a large number of significant applications (a.k.a. "seven dwarfs" [10]), has an all-to-all interaction pattern. It simulates a system of $n$ particles (bodies) over time. In each timestep, it computes the total force $F(i)$ on each particle $i$ from all other particle $j$,

$$\begin{aligned} &\textbf{for } i = 0 : n,\ j = 0 : n \\ &\qquad F(i) \mathrel{+}= \text{interact}(Q(i), Q(j)), \end{aligned}$$

where $Q(i)$ represents the necessary information of particle $i$ such as position, charge, mass, etc.

The N-body simulation is widely used in molecular dynamics [150, 164, 161] and astrophysics [175, 91]. Its all-pairs interaction pattern also occurs in many other applications such as database join, collision detection in computer graphics, *all-against-all* genome comparison to study genome evolution in comparative genomics [116, 120, 83, 178, 8, 153], and many more. This thesis explores the N-body problem in the context of molecular dynamics, but our algorithms directly apply to all other similar applications.

All-pairs interactions have $O(n^2)$ complexity. Two common optimizations to save computation time are (1) interacting only particles within a limited distance from each other, and (2) utilizing force symmetry. (1) For short-range forces, far-away particles have little force contributions, and we can interact only particles within a cutoff distance from each other. This introduces sparsity to the interaction space. We call this **cutoff sparsity**. (2) According to Newton's third law of motion, the force from particle $i$ to $j$ is equal to the negative force from particle $j$ to $i$, $f_{ij} = -f_{ji}$. We do not need to compute $f_{ji}$ if we have already computed $f_{ij}$. This cuts the amount of work by a half. We call this **symmetry sparsity**. We derive communication-optimal algorithms for the direct 2-way interaction N-body problems without and with these sparsities in Chapter 3. Chapter 4 generalizes the algorithm to support $k$-way interactions,

$$\begin{aligned} &\textbf{for } i_0 = 0 : n,\ i_1 = 0 : n,\ \ldots,\ i_{k-1} = 0 : n \\ &\qquad F(i_0) \mathrel{+}= \text{interact}(Q_{i_0}, Q_{i_1}, \ldots, Q_{i_{k-1}}). \end{aligned}$$

The matrix multiplication problem is also all-to-all in a sense that each output $c_{ij}$ needs to go through all values of the index $k$ from $A$ and $B$. Even though matrix multiplication

has been thoroughly studied, there has been relatively little work on sparse-dense matrix-matrix multiplication. Sparse-dense matrix multiplication is an important kernel in increasing number of applications in many fields, including linear algebra [105, 76, 176, 68], graph algorithms [173], computer security [97, 84], and statistical and machine learning [106, 142]. Chapter 5 shows that the existing communication-optimal dense-dense and sparse-sparse matrix multiplications are not optimal in this case (or when the numbers of nonzeroes of the two operands are of different orders of magnitude). We propose new communication-avoiding "1.5D" algorithms that achieved up to 100× speedup from existing, more commonly used 2D/2.5D/3D algorithms. Chapter 7 uses our communication-avoiding sparse-dense matrix multiplication kernels to implement a large-scale sparse inverse covariance matrix estimator for graphical model learning in machine learning.

## 1.4 Contributions

This thesis makes three types of contributions: lower bounds, algorithms, and implementations. We give the communication lower bounds for the following problems:

- All-pairs 2-way interaction N-body problem.
- All-unique-triplets 3-way interaction N-body problem.
- All-unique-tuples $k$-way interaction N-body problem without and with large cutoff distance.
- Sparse-dense matrix-matrix multiplication.

All our algorithms presented in the thesis are provably communication-optimal or communication-efficient, i.e., perform asymptotically less communication than previous algorithms:

- A communication-optimal all-pairs 2-way interaction N-body algorithm.
- A communication-optimal all-unique-triplets 3-way interaction N-body algorithm.
- A communication-optimal all-unique-tuples $k$-way interaction N-body algorithm that handles large cutoff distance.
- A family of communication-avoiding sparse-dense matrix-matrix multiplication algorithms based on 1-dimensional matrix partitioning that is provably more communication-efficient than previously known algorithm when one matrix operand has much fewer nonzeroes than the other.
- Two novel communication-avoiding algorithm variants to implement a sparse inverse covariance matrix estimator based on a pseudo likelihood method, composed of our previous communication-avoiding sparse-dense matrix-matrix multiplication kernels.

Lastly, we give highly scalable implementations and performance results demonstrating their efficiency in practice:

- An all-pairs 2-way N-body implementation demonstrating near perfect strong scaling on two distributed-memory machines, attaining up to 11.8× speedup over its non-communication-avoiding version.
- 1- and 2-dimensional simulation space cutoff 2-body implementations demonstrating up to 2× speedups and good scalability on two distributed-memory machines.
- An all-unique-triplets 3-way N-body implementation demonstrating perfect strong scaling on a distributed-memory machine, attaining up to 42× speedup.
- An efficient sparse-dense matrix-matrix multiplication that outperforms existing, more commonly used communication-avoiding algorithms by up to a 100× speedup. The implementation is openly available on Bitbucket [12].
- Two large-scale sparse inverse covariance estimation implementations (HP-CONCORD) which, to the best of our knowledge, are the first distributed-memory regression-based pseudo likelihood optimization implementations. Our implementations outperformed an existing sparse inverse covariance method and demonstrated high scalability, allowing us to analyze high-dimensional data with arbitrary underlying structures at a scale that was previously intractable, e.g., 1.28 million dimensions (over 800 billion parameters) in under 21 minutes on 24,576 cores of a CrayXC30 machine. HP-CONCORD is publicly available on Bitbucket [12] and as a user-installed module at National Energy Research Scientific Computing Center (NERSC).
- A case study using HP-CONCORD to automatically recover the underlying human brain connectivity from a resting state fMRI dataset. Our analysis shows good agreement with a state-of-the-art clustering, which used manual intervention and significant domain knowledge, from the neuroscience literature.

## 1.5 Outline

The rest of this thesis is organized as follows: Chapter 2 explains common notation and terminology we will use throughout the thesis. Chapter 3 presents the 2-way interaction N-body problem, without and with cutoff distance and support for symmetry. Chapter 4 discusses many-body interactions, starting with the all-unique-triplets 3-way interaction N-body problem and generalizing the algorithm to handle the all-unique-tuples $k$-way interaction N-body problem with cutoff. Chapter 5 switches context to the sparse-dense matrix-matrix multiplication problem, to which we apply communication-avoiding techniques similar to those of Chapter 3 to gain large speedups. Chapter 6 generalizes the sparse-dense matrix multiplication kernels in Chapter 5 and gives more implementation details necessary for Chapter 7. Chapter 7 is a case study of sparse inverse covariance matrix estimation, where we compose multiple communication-avoiding 1.5D matrix multiplication kernels from Chapter 6 together and apply necessary changes to other intermediate operation to keep the data replicated throughout. Finally, we conclude in Chapter 8.

# Chapter 2

# Preliminaries

This chapter details our system assumptions, performance metrics, communication lower bounds, and terms that will be used throughout the dissertation.

## 2.1 Performance Model

We assume a distributed system of homogeneous processors connected through the network. Our metric is time. We model the total running time by the computation and communication costs on the critical path per processor, assuming that work must be performed on more than one processor so the communication costs cannot be trivially zero. We further divide communication costs into latency and bandwidth costs. A $w$-word message from one processor to another processor takes $\alpha + w\beta$ units of time, where $\alpha$ is the fixed cost for a message (latency cost) and $\beta$ is the time to send a word (reciprocal bandwidth). The total running time is therefore

$$T_{\text{total}} = F\gamma + S\alpha + W\beta, \tag{2.1}$$

where $F$ is the number of flops computed by a processor, $\gamma$ is the time per flop, $S$ is the number of messages, and $W$ is the total number of words sent in all messages combined. We assume $\gamma, \alpha$, and $\beta$ are fixed and usually count $F, S$, and $W$ to compare algorithms. This model can also be modified to support energy as a metric [61].

## 2.2 Communication Lower Bounds

To measure how well our algorithms *avoid* communication, we derive communication lower bounds on the number of messages ($S$) and words ($W$) for each particular problem. If our algorithms attain the lower bounds, then they are communication-optimal and require no further improvement, i.e., other algorithms that perform the same, or sufficiently similar, arithmetic operations cannot do better.

Throughout this thesis, we follow the framework first established for matrix multiplication by Irony et al. [96]. We sketch the idea of the lower bound here, see [23] for details. Let $Z$ be the total number of arithmetic operations a processor has to perform to solve a problem and $M$ be the size of memory available to one processor. Let $F$ be the upper bound of the number of useful arithmetic operations, i.e., those that make progress towards the goal of $Z$ operations, that each processor can do with the locally available $O(M)$ words, without communication. Then, the lower bound of the number of messages is $\Omega(Z/F)$ since, for each $F$ operations, each processor needs at least one message to fill the memory with new data. Each message can contain at most $O(M)$ words, so the lower bound of the number of words is $\Omega(Z/F \cdot M)$. Therefore,

$$S = \Omega\left(\frac{Z}{F}\right), \qquad W = \Omega\left(\frac{Z \cdot M}{F}\right). \tag{2.2}$$

$Z$ is trivially the problem's computational complexity divided equally among processors, i.e, $Z = O(n^3/p)$ for direct matrix multiplication of size $n$, where $p$ is the number of processors. $M$ depends on the hardware. The most challenging part is finding $F$. Ballard et al. [23] proved that $F = O(M^{3/2})$ for all direct-linear-algebra-like applications, which we will use to prove the lower bounds of sparse-dense matrix-matrix multiplication in Chapter 5. Christ et al. [47] generalized the idea to cover any loop nests with subscripts that are affine functions of loop indices and proved that $F = O(M^s)$ where $s$ is the solution of a linear program from loop index coefficients of all subscripts. Our N-body lower bounds in Chapter 3 and 4 can be reproduced with Christ et al.'s framework [47].

## 2.3 Processor Topology and Replication

Our algorithms logically arrange the processors into a $d$-dimensional torus, denoted with $P$. Let $p$ be the total number of processors and $p_i$ the number of processors in the $i^{\text{th}}$dimension. We use the Python ":" operator to refer to a range of array elements, i.e., $x : y$ refers to the range $[x, y)$. Our indexing is zero-based and cyclic, e.g.,

$$P(i_0, i_1, \ldots, i_{d-1}) = P(i_0 \bmod p_0, i_1 \bmod p_1, \ldots, i_{d-1} \bmod p_{d-1}).$$

Table 2.1 summarizes our notation.

In parallel algorithms, the minimum amount of data a processor can hold is $1/p$ of data. To asymptotically avoid communication, our algorithms store $c$ times more data, which we also refer to as *c layers*. Replicating $c$ copies means grouping processors into $p/c$ teams of $c$ each and splitting data between teams. All team members hold and cooperate on the same data that is $c/p$ times the size of the full data set.

Figure 2.1 shows how topologies partition data, without and with replication. Figure 2.1a-2.1c map $p = 8$ processors to a 1-dimensional data array of size $n$. (This can also be seen as a 1D block-column partitioning of a matrix of size $m \times n$.) Each processor holds $cn/p$

| Notation | Meaning |
|---|---|
| $P$ | Logical $d$-dimensional torus of processors. |
| $p$ | Total number of processors. |
| $p_i$ | Number of processors in the $i^{\text{th}}$dimension, $0 \leq i < d$. |
| : | Python range notation. $x : y$ means the range $[x, y)$. |
| $P(i, j, k)$ | Processor at coordinate $(i, j, k)$. |
| $P(x_1{:}x_2, :, k)$ | Processors $(i, j, k)$ where $x_1 \leq i < x_2$ and $0 \leq j < p_1$. |
| $c$ | Replication factor, i.e., number of processors in a team. |

Table 2.1: Common notation throughout the dissertation.

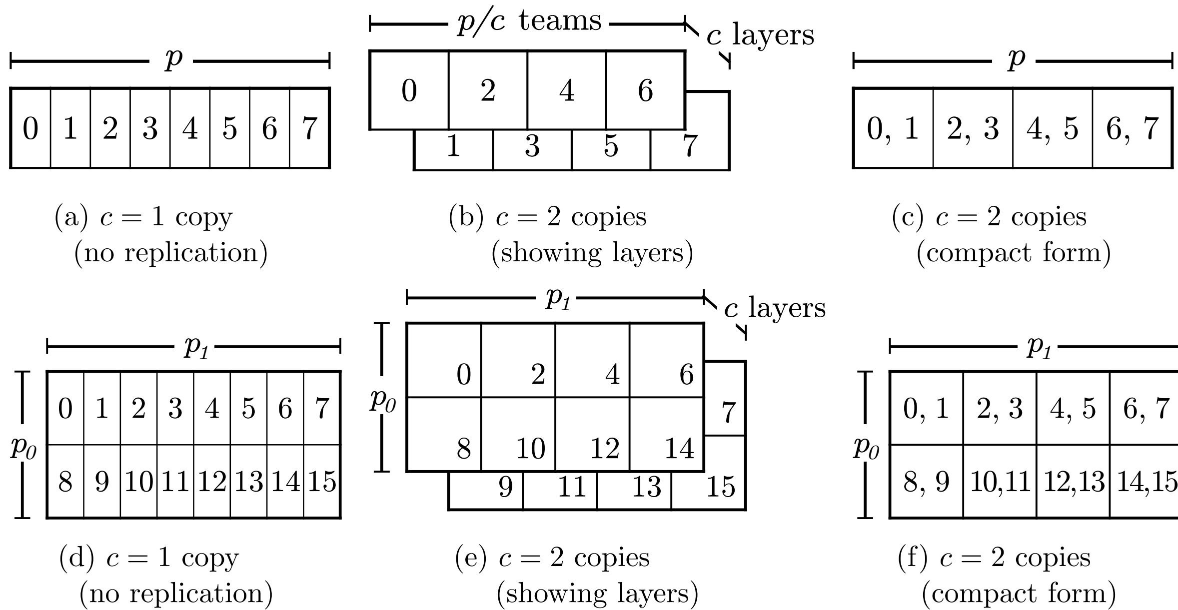



Figure 2.1: Mappings of processor meshes and data, without and with replication. Each submatrix is labeled with the processor rank(s) that it resides on. (a)-(c): $p = 8$ processors with 1-dimensional array data. (d)-(f): $p = 16$ processors with 2-dimensional array data (e.g., matrices).

elements. When $c = 1$, shown in Figure 2.1a, it becomes normal work partitioning with no replication. Figure 2.1b shows how we replicate $c = 2$ copies by logically arrange processors into 4 teams by 2 layers. All team members hold the same array part. In other words, each layer partitions the array equally. Figure 2.1c shows the same partitioning as Figure 2.1b, but draws just one copy of the array and lists all processor ranks that own that part together to save space. Similarly, Figure 2.1d-2.1f map $p = 16$ processors to a 2-dimensional array of size $m \times n$ (e.g., matrices). Figure 2.1d shows the non-replicating case where processors are arranged into a $2 \times 8$ mesh. Figure 2.1e and 2.1f shows the mesh with $c = 2$ copies (layers), where 8 processor teams are arranged into a $2 \times 4$ mesh.

| Operation | Description | S | W |
|---|---|---|---|
| Shift | Shifting by distance $d$ along the $k^{\text{th}}$ dimension means each $P(..., i_k, ...)$ sending $w$ words to $P(..., i_k + d, ...)$ and receiving $w$ words from $P(..., i_k - d, ...)$, simultaneously. | $O(1)$ | $O(w)$ |
| Broadcast | Sends $w$ words to all $p$ processors. | $O(\log p)$ | $O(w \log p)$ |
| Reduce | Sums a vector of $w$ words from each of $p$ processors in an element-wise manner to produce a vector of $w$ results. | $O(\log p)$ | $O(w \log p)$ |
| Gather | Gathers $w$ words from each of $p$ processors and creates $wp$ words on the gather root. | $O(\log p)$ | $O(wp \log p)$ |
| Alltoall | All $p$ processors exchange $w$ words with each other. | $O(\log p)$ | $O(wp \log p)$ |

Table 2.2: Common communication operations and their costs when operating on $w$ words.

## 2.4 Communication Operations

Table 2.2 lists common communication operations that occur in most chapters and their costs when operating on $w$ words. Because we assume a fully connected network without congestion, the latency and bandwidth costs are independent of the distance between processors. Each processor can only send a single message at a time, so we assume collective operations, such as broadcasts, are done in a log-depth tree [45].

## 2.5 Matrix Notation

We use capital letters to represent matrices. Let $X^{\{h \times w\}}$ denote a partitioning of a matrix $X$ into an $h \times w$ grid of equal size submatrices. The same tuple indexing applies here, for example,

$$X^{\{h \times w\}}(i, :) = [X^{\{h \times w\}}(i, 0)|...|X^{\{h \times w\}}(i, w-1)].$$

For brevity, we will drop the unnecessary submatrix index if the matrix is only partitioned in one dimension, i.e.,

$$X^{\{1 \times w\}}(i) = X^{\{1 \times w\}}(0, i).$$

We will refer to matrix elements as a corresponding lowercase letter with subscript. $x_{ij}$ means the element in row $i$ and column $j$ of matrix $X$.

Let $\text{nnz}(\cdot)$ denote the number of nonzeroes of a matrix. We use this notation for dense matrices as well because it allows us to simplify notation and compare asymptotic results for matrices with different sparsity and aspect ratios.

# Chapter 3

# 2-way N-Body

The gravitational N-body problem simulates the dynamics of a system of $n$ particles (bodies). It is widely used in many fields of science and engineering such as molecular dynamics, astrophysics, fluid dynamics, material science, etc. Its most compute-intensive part is the calculation of the total force, $F(i)$, on a given particle $i$,

$$F(i) = -\nabla \left[ \phi_1(Q(i)) + \sum_{i \neq j} \phi_2(Q(i), Q(j)) + \sum_{i \neq j \neq k \neq i} \phi_3(Q(i), Q(j), Q(k)) + \dots \right], \quad (3.1)$$

where $Q(i)$ refers to the position of particle $i$, $\nabla$ is the gradient operator, and $\phi_k$ is the $k$-body potential energy function. We call the calculation of the $k^{\text{th}}$ term over all $k$-tuples the $k$-body problem.[1] This series is believed to be rapidly convergent, i.e., each subsequent term has less effect than the previous term, despite its much higher computational complexity. Typical applications get sufficiently accurate results with just pairwise interactions [69] and the term N-body generally refers to this 2-body problem.

There are two approaches to computing 2-body interactions: direct and approximate. Direct methods calculate all required $O(n^2)$ interactions. Approximate or tree-based methods treat a group of sufficiently far away particles as one big particle and compute significantly fewer interactions. For example, the Barnes-Hut algorithm [28] processes $O(n \log n)$ interactions and the Fast Multipole Method (FMM) [87] computes $O(n)$ interactions. We will focus on the direct-interaction 2-body problem. Even for the approximate algorithms, the $O(n^2)$ calculations on nearby particles tend to dominate running time, so the direct problem is still of interest.

Using a framework developed by Ballard et al. [23], we give lower bounds on the communication in terms of the numbers of messages and words sent along the critical path. We show that previous work only attains the lower bound in specific cases, while our algorithm has a tunable parameter, a replication factor $c$, that can be tuned to always attain the

This chapter is based on joint work previously published in "A Communication-Optimal N-Body Algorithm for Direct Interactions" [65].

[1] Not to be confused with the many-body problem in quantum mechanics.

lower bounds. With enough memory for $c$ copies of the particles, we can theoretically reduce bandwidth and latency costs by factors up to $c$ and $c^2$, respectively. This effect is also observable in practice: performance results from two high-performance computing systems indicate that our algorithm achieves nearly perfect strong scaling with the right choice of $c$. We find that maximizing $c$ may not yield the best performance in practice because collective cost increases with $c$ and also sometimes fail to scale logarithmically as our model assumes, so $c$ should be treated as a tuning parameter.

In this chapter we will show:

- A derivation of the lower bounds on communication for an N-body simulation timestep. The bounds capture both the number of messages and the number of words sent along the critical path.
- An algorithm for particle interactions that achieves the lower bound for a fixed memory size. The algorithm allows finite cutoff distances beyond which interactions have constant or zero effect.
- Performance results demonstrating attainable speedups from the new algorithm on two distributed-memory machines.

This chapter is organized as follows. Section 3.1 derives the communication lower bounds for the N-body problem. Section 3.2 reviews related work. Section 3.3 gives the communication-avoiding N-body algorithm, proves its optimality, and presents performance results from experiments on two supercomputers. Section 3.4 generalizes the algorithm to include a cutoff radius beyond which particles have constant effect. Again, we give a proof of optimality and performance results from real systems. Section 3.5 extends the algorithm to utilize force symmetry. Section 3.6 describes further directions.

## 3.1 Communication Lower Bounds

Our communication-avoiding algorithm was motivated by an examination of communication lower bounds for the N-body problem. Recall the general lower bounds on latency $S$ and bandwidth $W$ from Equation (2.2),

$$S = \Omega\left(\frac{Z}{F}\right), \qquad W = \Omega\left(\frac{Z \cdot M}{F}\right). \tag{3.2}$$

**Theorem 3.1.** *The maximum number of force evaluations that can be computed with $M$ particles is $O(M^2)$.*

*Proof.* Figure 3.1 shows the interaction space of the N-body problem, sometimes called the *force matrix*. Each coordinate $(i, j)$ represents the interactions between particles $i$ and $j$. We would like to upper-bound the size of $V$, the set of interactions a processor can compute with $O(M)$ particles in memory. Let $V_i$ and $V_j$ be the set of indices $i$ and $j$ in the set $V$,

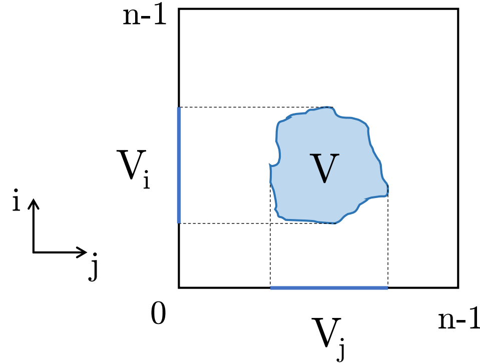



Figure 3.1: The force matrix (iteration space) for the all-pairs N-body problem. Each coordinate $(i, j)$ represents the interaction between particles $i$ and $j$. Vertical and horizontal axes show the indices $i$ and $j$, respectively. Each processor computes interactions in an area $V$. $V_i$ and $V_j$ are the projections of $V$ on axes $i$ and $j$, i.e., the sets of $i$ and $j$ indices.

i.e., the projection of $V$ onto axes $i$ and $j$ in Figure 3.1. Denote the cardinality of a set by $|\cdot|$. Then,

$$|V| \leq |V_i||V_j|,$$

since $|V_i||V_j|$ is the size of the bounding box that contains $V$ (a special case of Loomis and Whitney's inequality [127]). Because $V_i$ and $V_j$ must fit in memory, $|V_i|$ and $|V_j|$ can be at most $O(M)$, therefore, $|V| \leq O(M^2)$.

□

The number of force evaluations that can be computed with $M$ particles can be upper-bounded as $F_{\text{2-body}} = O(M^2)$. This upper bound, which represents the maximal amount of potential data-reuse, yields the communication lower bounds

$$S_{\text{2-body}} = \Omega\left(\frac{Z}{M^2}\right), \qquad W_{\text{2-body}} = \Omega\left(\frac{Z}{M}\right).$$

We use the terms *particle* and *word* interchangably here because we are analyzing asymptotic costs and each particle is of size $O(1)$ words.

## All-pairs interactions

If all interactions of $n$ particles are computed on $p$ processors in a load-balanced fashion, each processor must compute $O(n^2/p)$ force evaluations, yielding the following communication lower bounds

$$S_{\text{2-body}} = \Omega\left(\frac{n^2}{p \cdot M^2}\right), \qquad W_{\text{2-body}} = \Omega\left(\frac{n^2}{p \cdot M}\right). \tag{3.3}$$

$M$ being in the denominator of these lower bounds suggests that increased memory size allows less communication. In the parallel case, it turns out that given extra memory, data-replication can be used to lower the communication cost. In fact, many existing N-body algorithms already use data replication, as we detail below. The novelty of our algorithm is that we parameterize the number of data copies and minimize communication for any amount of memory, as done for 2.5D algorithms [167] for dense linear algebra. 2.5D algorithms are a memory-aware extension of 3D algorithms [57, 3, 4, 29, 99], which use $p^{1/3}$ copies of data on $p$ processors.

For convenience, we write $M$ as a multiple of $n/p$, the minimum data required for each processor. Let $M = O(c_M n/p)$, where $c_M \in \mathbb{Z}^+$. The lower bounds in Equation (3.3) become,

$$S_{\text{2-body}} = \Omega\left(\frac{p}{c_M^2}\right), \qquad W_{\text{2-body}} = \Omega\left(\frac{n}{c_M}\right). \tag{3.4}$$

Next, we derive the memory-independent lower bounds. Using Figure 3.1 again, let $V$ be the set of interactions the processor with the largest load has to compute. $|V|$ must be at least the total amount of work divided equally among processors,

$$|V| \geq \frac{n^2}{p}. \tag{3.5}$$

We relate $|V|$ to the total input size $|V_i| + |V_j|$ necessary to compute $V$,

$$\begin{aligned} |V| &\leq |V_i||V_j| \\ &\leq \frac{1}{2}|V_i|^2 + |V_i||V_j| + \frac{1}{2}|V_j|^2 = \frac{1}{2}(|V_i| + |V_j|)^2. \end{aligned} \tag{3.6}$$

Putting Equations (3.5) and (3.6) together, we have,

$$\begin{aligned} \frac{1}{2}(|V_i| + |V_j|)^2 &\geq \frac{n^2}{p} \\ |V_i| + |V_j| &\geq \sqrt{2}\frac{n}{\sqrt{p}}. \end{aligned}$$

Since $V_i$ can be the same as $V_j$, that processor must hold at least $(\sqrt{2}n/\sqrt{p})/2 = n/\sqrt{2p}$ particles. Assuming every processor starts with just $n/p$ particles in their memory, they must communicate at least $n/\sqrt{2p} - n/p$ words, using at least 1 messages. Therefore, the memory-independent lower bounds are,

$$S_{\text{2-body}} = \Omega(1), \qquad W_{\text{2-body}} = \Omega\left(\frac{n}{\sqrt{p}}\right), \tag{3.7}$$

equivalent to the memory-dependent lower bounds when $c_M = \sqrt{p}$. Any $c_M > \sqrt{p}$ gets the same lower bounds in Equation (3.7).

## Cutoff interactions

In molecular dynamics simulations, it is common to impose a cutoff on direct force interaction evaluations. Force interactions decay with distance, so their evaluation can be truncated by ignoring interactions between particles which lie beyond some cutoff distance. Typically, a correction term accounts for the contribution of long-distance interactions. Since the long-distance contribution to the potential is smooth, grid-based solvers are often employed to evaluate this correction. Instead of analyzing the costs associated with computing long-range interactions, we will instead focus on the analysis of direct interactions within the cutoff distance.

Our lower bounds extend trivially to the case where direct interactions are truncated within a cutoff. The only modification to the argument is a difference in the total number of computations which is now not $n^2$, but rather $F = nk$, where $k$ is the number of interactions necessary for each particle. The lower bounds on the number of messages and words that must be sent are

$$S_{\text{cutoff}} = \Omega\left(\frac{nk}{p \cdot M^2}\right), \qquad W_{\text{cutoff}} = \Omega\left(\frac{nk}{p \cdot M}\right). \tag{3.8}$$

Substituting $M = c_M n/p$, we get,

$$S_{\text{cutoff}} = \Omega\left(\frac{k}{n} \cdot \frac{p}{c_M^2}\right), \qquad W_{\text{cutoff}} = \Omega\left(\frac{k}{c_M}\right). \tag{3.9}$$

For the memory-independent lower bounds, consider the processor with the largest set of interactions $V$. Again, $|V|$ must be at least $1/p$ of the total amount of work,

$$|V| \geq \frac{nk}{p} \tag{3.10}$$

Combining Equations (3.10) and (3.6), we have,

$$\frac{1}{2}(|V_i| + |V_j|)^2 \geq \frac{nk}{p}$$
$$|V_i| + |V_j| \geq \sqrt{\frac{2nk}{p}}.$$

Similar to the all-pairs case, since $V_i$ and $V_j$ can be the same, the processor needs at least $\sqrt{2nk/p}/2 = \sqrt{nk/(2p)}$ particles to compute $V$. Assuming each processor starts with $n/p$ particles in memory, the processor must communicate at least $\sqrt{nk/(2p)} - n/p$ words, using at least 1 message. We assume that $\sqrt{nk/(2p)} - n/p > 0$ (in other words, $k > 2n/p$), otherwise, the problem would be trivial and does not require communication, which is not the focus of our work. The memory-independent lower bounds are,

$$S_{\text{cutoff}} = \Omega(1), \qquad W_{\text{cutoff}} = \Omega\left(\sqrt{\frac{nk}{p}}\right), \tag{3.11}$$

equivalent to the memory-dependent lower bounds when $c_M = \sqrt{kp/n}$. Any $c_M > \sqrt{kp/n}$ would yield the same lower bounds.

## 3.2 Previous Work

In this section, we review related algorithms, analyze their communication costs, and show that they are communication optimal only in extreme cases: when there is minimal memory available ($M = O(n/p), c_M = 1$) or maximal memory available ($M \geq O(n/\sqrt{p}), c_M \geq \sqrt{p}$). We discuss (1) all-pairs interaction algorithms (particle and force decompositions), (2) algorithms for interactions with cutoff radius (special decomposition and neutral territory methods), and (3) some of their applications.

### All-pairs interaction

**Particle decomposition.** The naïve approach for parallelizing N-body simulations is to assign each processor $n/p$ particles (*particle decomposition*) and make it compute all the interactions for its $n/p$ particles with all other particles [73]. One way to carry out the calculations is to use a simple ring/systolic scheme: All processors logically form a ring. Each processor stores its $n/p$ particles in two buffers. The first buffer is fixed. The other buffer is used to exchange particles with other processors. For $p$ rounds, each processor alternates between interacting all its particles (in the fixed buffer) with all particles in the exchange buffer and shifting the $n/p$ particles in the exchange buffer around to its right neighbor, receiving new particles from its left neighbor. (The shift direction does not matter and can be reversed.) The amount of communication required along the critical path is

$$S_{\text{particle}} = O(p), \qquad W_{\text{particle}} = O(n),$$

since each processor sends a message per round, each message containing $n/p$ particles. This algorithm stores $O(n/p)$ words in memory. (The actual storage requirement is $2n/p$, but we are using asymptotic costs so we drop the constant 2.) According to the lower bounds in Equation (3.4), this algorithm is communication-optimal only if there is minimal memory available ($c_M = 1$). If there is more memory available ($c_M > 1$), it is suboptimal.

**Force decomposition.** Plimpton pointed out that, by assigning each processor a block of force interactions rather than particles (*force decomposition*), communication is reduced [150], but did not prove if the algorithm is communication-optimal. In particular, $n^2$ total interactions need to be computed, and each of $p$ processors computes an $n/\sqrt{p}$-by-$n/\sqrt{p}$ block of the interactions. Thus, each processor requires only $2n/\sqrt{p}$ particles to compute its interactions and must return $2n/\sqrt{p}$ force contributions. Assuming the particle locations are not initially replicated and the forces must be collected at the end, a broadcast and a reduction is required to communicate these data sets. Thus the total communication costs are

$$S_{\text{force}} = O(\log(p)), \qquad W_{\text{force}} = O(n/\sqrt{p}).$$

Ignoring the $\log(p)$ factor, we get $S_{\text{force}} = O(1)$. With respect to the particle decomposition, force decomposition reduces latency by a factor of $p$ and bandwidth by a factor of $\sqrt{p}$. However, the memory usage goes up by a factor of $\sqrt{p}$ since each particle is replicated $\sqrt{p}$ times. According to Equation (3.7), this algorithm is communication-optimal when memory is at least the maximal effective amount ($c_M \geq \sqrt{p}$), but it cannot run at all otherwise ($c_M < \sqrt{p}$). It also requires that $p$ is a perfect square, which is quite restrictive in practice.

## Interactions with cutoff distance

**Spatial decomposition.** A spatial decomposition is a natural choice for parallelizing problems with a cutoff distance. In such a decomposition, each processor owns all particles in a region of the physical simulation domain. Consequently, processors must only communicate with their neighbors, with the number of neighbors given by the ratio of the cutoff with respect to the length of the simulation box and the dimensionality of the problem space. For instance, given some cutoff $\delta$ spanning $b$ processor boxes, each processor must communicate with $O(b^d)$ processors, where $d$ is the dimensionality of the simulation space. If processors form pairs to compute their particles interaction, a message of size $O(n/p)$ is required between each pair of interacting processors, assuming particles are uniformly distributed. This spatial decomposition algorithm has a communication cost of

$$S_{\text{spatial}} = O(b^d), \qquad W_{\text{spatial}} = O(nb^d/p). \tag{3.12}$$

To compare with the lower bounds in Equation 3.9, we find $k$, the expected number of interactions per particle. The ratio of the cutoff volume to the simulation volume is $b^d/p$. There are $n$ particles scattered uniformly across the simulation space, so $k = nb^d/p$. Rewriting Equation 3.12 in terms of $k$,

$$S_{\text{spatial}} = O\left(\frac{k}{n} \cdot p\right), \qquad W_{\text{spatial}} = O(k).$$

Therefore, the spatial decomposition is only communication optimal in the case of minimal memory $M = O(n/p)$. Is it suboptimal for any larger $M$'s ($c_M > 1$).

**Neutral territory methods.** Force decomposition can be done when a cutoff is imposed on interactions, but physical locality must now be considered in the algorithm. Hybrids between force and spatial decomposition can ensure necessary locality to provide communication optimality and yield the methods most commonly used in practice currently. They assume uniform particle density and bin particles to processors according to their positions. Generally, these methods can be defined as 'neutral territory' (NT), since force interactions are computed on processors which do not necessarily own either of the interacting particles in their assigned spatial territory. These algorithms achieve the communication costs

$$S_{\text{NT}} = O(1), \qquad W_{\text{NT}} = O\left(\frac{nb^{d/2}}{p}\right),$$

which is asymptotically optimal for the unlimited memory case $c_M \geq O(\sqrt{nk/p})$ according to the lower bounds for cutoff interactions in Equation 3.11 ($k = nb^d/p$).

Snir proposed a hybrid between a spatial and a force decomposition for cutoff interactions in 3-dimensional space in [164]. Snir's algorithm performs a multicast of particle locations to a set of nearby processors. Snir also gave lower bounds on the communication cost for this problem which showed the optimality of his algorithm asymptotically. However, Snir did not consider the limited-memory scenario.

The neutral territory method [161] was introduced by Shaw independently of Snir and achieves the same asymptotic cost. Shaw's algorithm selects a 3-dimensional import region that has a volume smaller than that of Snir's by a constant factor.

The midpoint method [33] is another interesting variation of neutral territory methods. In the midpoint method, a processor computes all interactions for which the midpoint of the interacting particles lies in the processor's territory. While the method is not asymptotically optimal, it has a smaller import region for small-to-medium number of processors, depending on the ratio of the cutoff volume to the simulation space volume. The method also has advantages in latency and throughput on torus networks, and can be generalized to many-body interactions.

## Existing applications

Existing molecular dynamics applications that employ neutral territory decompositions already utilize the knowledge that replication can lower communication costs, although they do not necessarily meet the lower bounds.

NAMD [148] is a parallel molecular dynamics simulation package which runs on top of the Charm++ runtime system [100]. Charm++ is an object-based parallel framework which decomposes work and data into 'chare' objects. Arrays of chare objects are dynamically mapped to processors by the runtime system. Chares communicate with each other via asynchronous message invocations.

NAMD employs an algorithm that decomposes particles spatially into an array of 'patch' chare objects and decomposes forces into 'compute' chare objects. Patches communicate particle data to compute objects and compute objects send force contributions back to patches. This algorithm employs data replication: The patch size does not depend on the number of processors but is selected so that one patch (sometimes two) is slightly wider that the cutoff distance, so each patch only need to interact with nearest patches [30]. Its dynamic scheduling aims to schedule nearby compute objects to the same processor as much as possible while maintaining load balance.

The neutral territory method as described by Shaw [161] is employed by the specialized supercomputer Anton [162]. This supercomputer is designed to run parallel molecular dynamics simulations at high efficiency. The architecture was co-designed with consideration for the 3D structure of the spatial decomposition and the neutral territory interaction algorithm.

Desmond [34] is a general-purpose molecular dynamics package developed in association with Anton. Desmond uses the midpoint method to evaluate direct interactions. This software achieves good absolute performance and scalability on modern architectures such as the BlueGene/P.

## 3.3 Interactions with No Cutoff Radius

This section addresses simulations in which every particle interacts with every other particle. We present a communication-optimal algorithm for computing interactions between all pairs of particles, provide a proof of its optimality, and present performance results from our implementation.

Our algorithm can use variable amounts of memory and is communication-optimal for a given machine configuration whenever the maximum available memory is used. It encompasses both the particle decomposition (optimal for minimal memory) and the force decomposition (optimal for unconstrained memory and can only be run with processor counts that are perfect squares) approaches, but we also prove it is communication-optimal for every constrained memory situation between those. Moreover, even in the unlimited memory scenario, our algorithm can outperform the force decomposition approach in practice because there is a collection cost that increases with memory usage. The sweet spot that minimizes the total running time lies between the minimal and maximal memory size and can only be attained by our algorithm, as we will show in the experimental results.

### The all-pairs interaction algorithm

The communication-avoiding algorithm, given in Algorithm 1, uses $p$ processors to compute interactions between a set $Q$ of $n$ particles. The algorithm also takes as input a replication factor $c$ describing the number of times the set of particles is replicated in available memory. The processors are logically arranged into a grid $P$ of size $p/c$ teams by $c$ layers (processors per team). Particles are distributed evenly among the teams into local subsets $Q_t$ (for team $t$), and teams are responsible for computing updates to their local subset. Each processor has two buffers: a fixed buffer, $B_f$, which remains stationary throughout the computation and an exchange buffer, $B_x$, which is shifted around to receive new particles from other processors.

The algorithm proceeds as follows. A team leader, $P(t, 0)$, reads $Q_t$ into $B_f$ and broadcasts $B_f$ to the rest of the team, $P(t, :)$. Each team member makes a second copy in $B_x$. Each processor $P(t, \ell)$ then shifts $B_x$ along the $0^{\text{th}}$ dimension of $P$, with the shift distance $\ell$ (its layer ID). Then, for $p/c^2$ steps, each processor shifts $B_x$ by $c$ and, upon receiving a new set of particles, updates $Q_t$ accordingly. Finally, sum-reductions within each team combine the updates. Figure 3.2 illustrates the algorithm with teams being columns and layers being rows.

---

**Algorithm 1** CA-All-Pairs-2-Body

---

**Input:** Replication factor $c$, the number of extra copies of the particles.
**Input:** $P$, a processor grid arranged into $p/c$ teams by $c$ layers. Each team $P(t,:)$ has $c$ members, the first one being a team leader.
**Input:** Set $Q$ of $n$ particles divided evenly among team leaders into local subsets $Q_t$.
**Output:** An updated set $Q$.

1: **for** $P(t, \ell)$ in parallel **do**
2: $\quad$ $P(t, 0)$ reads $Q_t$ into the fixed buffer $B_f$.
3: $\quad$ $P(t, 0)$ broadcasts $B_f$ to $P(t,:)$.
4: $\quad$ Copy $B_f$ to exchange buffer $B_x$.
5: $\quad$ Shift $B_x$ by $\ell$ along the $0^{\text{th}}$ dimension.
6: $\quad$ **for** $p/c^2$ **steps do**
7: $\quad\quad$ Shift $B_x$ by $c$ along the $0^{\text{th}}$ dimension.
8: $\quad\quad$ Update particles in $B_f$ based on effects of $B_x$.
9: $\quad$ **end for**
10: **end for**
11: Sum-reduce updates within $P(t,:)$.

---

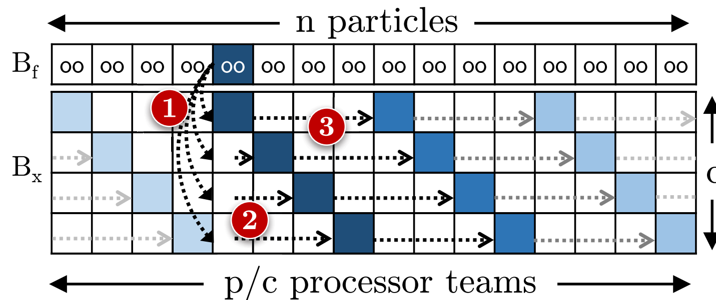



Figure 3.2: Illustration of Algorithm 1. Columns are the processor teams and rows are the layers. We only draw one row of the fixed buffer $B_f$ because every team member has the same content ($Q_t$ for team $t$). The exchange buffer $B_x$ is shown for all processors. The color boxes trace the path of the fifth team's particles. The labels show: (1) the initial broadcast step within a team; (2) the skewing of particles according to the row index; (3) the first of $p/c^2$ shifts and updates. Not shown is a final reduction within a column to combine updates.

Our algorithm "interpolates" between the particle decomposition and force decomposition algorithms of Plimpton [150]. In fact, either decomposition falls out at extreme values of $c$. When $c = 1$, the algorithm resembles a particle decomposition with simple point-to-point shifting, and each team of one processor is responsible for computing all forces on its subset of particles. Similarly, when $c = \sqrt{p}$, the algorithm uses a force decomposition.

## Communication optimality

We analyze the communication cost along the critical path to show communication optimality. The cost can be expressed as the sum of costs of three phases: the initial broadcast and skewing, the shifting steps, and the final reduction. First, each of $p/c$ teams executes a broadcast of $O(cn/p)$ words among $c$ processors. Assuming logarithmic collective performance, the broadcast can be completed in parallel with $\log(c)$ messages. Then, each processor skews its particles row-wise in parallel, sending $O(cn/p)$ words in $O(1)$ messages. Next, the processors perform $p/c^2$ shifting steps in which they send $O(1)$ messages of $O(cn/p)$ words, yielding a subtotal cost of $O(p/c^2)$ messages and $O(n/c)$ words. A final reduction moves $O(cn/p)$ words in $\log(c)$ messages. Asymptotically, the total cost is

$$S_{CA-allpairs} = O\left(\frac{p}{c^2}\right), \qquad W_{CA-allpairs} = O\left(\frac{n}{c}\right), \tag{3.13}$$

which matches the lower bounds in Equation (3.4) when we use the maximum memory available ($c = c_M$). Hence, our algorithm is communication-optimal regardless of the memory size.

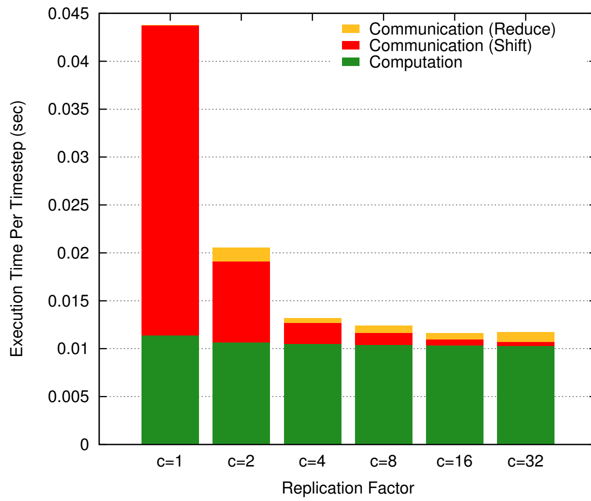
## Performance results

We built a simple particle simulation that uses our algorithm and measured its performance on two machines from different vendors. Our code simulates particles moving in a two-dimensional space with reflective boundary conditions. The particles exert a repulsive force on each other that drops off with the square of their distance. The force is symmetric, although we do not apply optimizations to exploit the symmetry. The particles are 52 bytes in size (one 32-bit integer for particle ID and six double-precision variables for position, velocity, and acceleration in $x$ and $y$ coordinates).

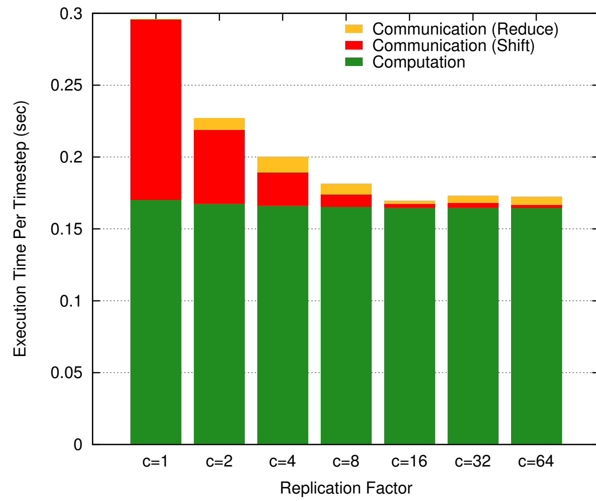
We ran our experiments on two platforms, Hopper and Intrepid. Hopper was a 6,384 node Cray XE-6 machine located at NERSC. Each node had two 12-core, AMD MagnyCours processors running at 2.1 GHz, yielding 24 cores per node and 153,216 cores in total.[2] Nodes were connected in a three-dimensional torus via the Cray Gemini interconnect. Intrepid [95] was a 163,480 core IBM Blue Gene/P machine located at ALCF. Each node consisted of one quad-core, 850 MHz PowerPC processor connected in a three-dimensional torus. We wrote all of these codes in C using MPI.

Since Blue Gene/P provided topology-aware partitions, we modified the code to utilize topology-aware collectives provided by the DCMF communication layer [71]. In doing so, we were able to fully exploit the bidirectional network links and minimize network contention. We found that replacing $P/c^2$ point-to-point shifts within the rows with $P/c^2$ broadcasts across the rows improved performance because the bidirectionality of the torus provides twice the bandwidth of a point-to-point send along a single link.

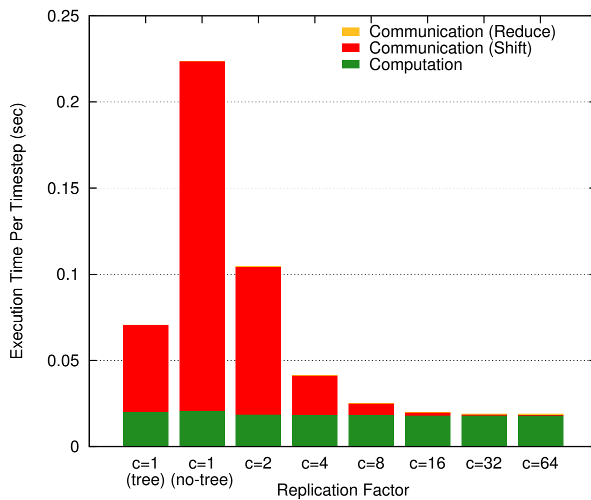
[2]Given Hopper had 24 cores per node, runs that saturate all cores often have factors of 3 in them that make our choice of experimental parameters seem odd. Powers-of-two numbers can be recovered by dividing by 3 in most cases.



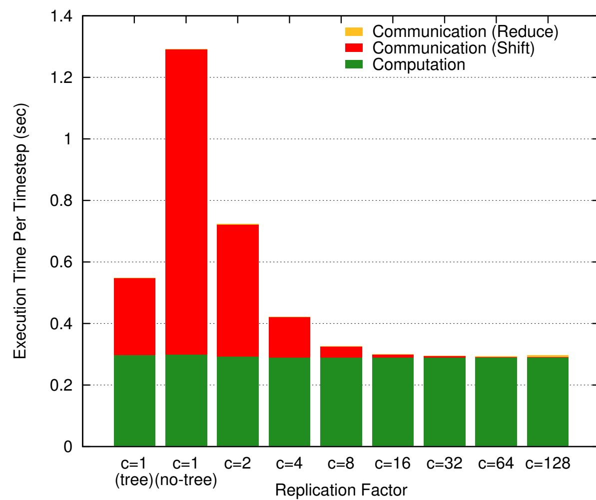

(a) Cray XE-6, 6,144 cores, 24,576 particles.




(b) Cray XE-6, 24,576 cores, 196,608 particles.




(c) Blue Gene/P, 8,192 cores, 32,768 particles.




(d) Blue Gene/P, 32,768 cores, 262,144 particles.

Figure 3.3: The effect of the replication factor $c$ on execution time for small and large problems on Cray XE-6 and Blue Gene/P. Figure 3.3a shows monotonically decreasing communication with increasing $c$, as predicted by the model. In contrast, Figure 3.3b shows the best performance when $c = 16$; this is the point at which the communication pattern strikes a balance between collective and point-to-point costs. Similar are the conclusions for Blue Gene/P on Figures 3.3c and 3.3d.

Our experiments sought to understand: (1) the effect of increased replication factors for fixed machine sizes and problem sizes, and (2) the strong scaling performance for all replication factors across a fixed problem size.

### Scaling the replication factor

The model predicts that communication cost should drop by factors between $c$ and $c^2$ for increased $c$. In practice, we find this to be accurate for small $c$. Figures 3.3a and 3.3b show the breakdown of communication and computation time for 24K-particle and 196K-particle simulations on 6K and 24K cores of Cray XE-6, respectively. When $c = 1$, communication costs represent significant portions of execution time. As $c$ increases, we see communication costs more-than-halving until $c = 16$. When $c = 64$ in the larger simulation, we see a greater cost than when $c = 16$, even though the model predicts better performance for the pure force decomposition. (Our algorithm is equivalent to force decomposition when $c = \sqrt{p}$, which cannot be run here because our processor counts are not perfect squares: $\sqrt{6{,}144} = 78.38$ and $\sqrt{24{,}576} = 156.77$. Judging from the increasing trend in collective cost, it is unlikely that the force decomposition will be faster.) We believe the communication pattern at this point best balances the costs of collective and point-to-point communication. We do not plot the initial broadcast cost because it is negligible. The computing time decreases slightly as we increase $c$ even though all replication factors compute the same amount of flops. This is because larger replication factors have more particles to interact locally in each round and can utilize cache more efficiently.

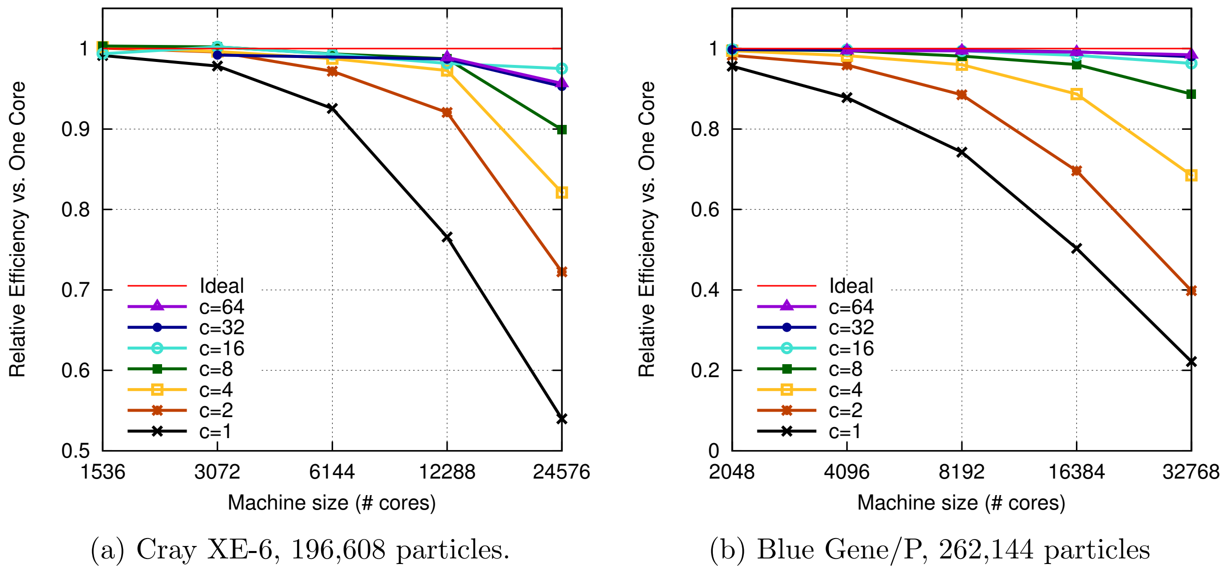
Figures 3.3c and 3.3d show the execution time breakdown for 32K-particle and 262K-particle simulations on 8K and 32K cores of Blue Gene/P, respectively. We plot two runs for $c = 1$. The *tree* bar represents a run which utilized a special network for collective operations involving the whole partition. Alternatively, we forced the use of the regular 3D-torus for the *no-tree* run. The specialized network is effective for the naïve implementation of the interaction algorithm, but our algorithm eventually outperforms the hardware-assisted variant by using the torus intelligently. For runs that just use the torus, we see a 99.5% reduction in communication time. We also see the slightly decreasing trend in computation time here.

### Strong scaling performance

We ran additional experiments to assess the strong scaling performance of the algorithm. Figure 3.4 shows the data from 196K and 262K particle runs on Cray XE-6 and Blue Gene/P, respectively. Our algorithm achieves nearly perfect strong scaling with the right choice of $c$.

## 3.4 Finite Cutoff Distance

Our communication-avoiding algorithm can be generalized to the case where particles have no effect beyond a cutoff radius $\delta$, or their effect can be approximated by a constant value. Like previous work, our algorithm is a hybrid between spatial decomposition and force decomposition, except that it is 'territorial' (as opposed to 'neutral territory'), i.e., processors compute forces on the particle sets they own. Again, our algorithm is communication-optimal for all memory sizes (not just the extreme cases as shown for spatial decomposition




(a) Cray XE-6, 196,608 particles.

(b) Blue Gene/P, 262,144 particles

Figure 3.4: Strong scaling performance on Cray XE-6 and Blue Gene/P. For the given problem sizes, our algorithm achieves nearly perfect strong scaling with the appropriate choice of replication factor.

and neutral territory methods) and is the only algorithm that can use intermediate memory size which can perform better that maximal memory size in practice. Here, we describe the algorithm in one-dimensional space, explain how it can be generalized to higher dimensional spaces, show that communication-optimality still holds, and give performance results from 1D- and 2D-problem-space experiments on Cray XE-6 and Blue Gene/P.

## The 1D interaction algorithm

The algorithm for distance-limited interactions in one dimension can be described as the algorithm for interactions with no cutoff plus two key refinements. First, we assume a spatial decomposition of particles among teams, i.e. each team is responsible for the particles in a particular region of the simulation space. Unlike the all-pairs interactions case which handles any particle distribution (all particles will interact with each other anyway), we assume a uniform distribution for load balance. If the particles are not uniformly distributed, some processors will have more particles to interact than others and take longer time to compute, causing all other processors idle and preventing good parallel efficiency. Second, the algorithm performs shifts modulo the cutoff distance, not the edge of the simulation space as before. Let $b$ be the number of processor teams $\delta$ spans and let $w$ be the width of the simulation space, assuming, without loss of generality, that the simulation space is square. Then, $b/(p/c) = \delta/w$ and therefore the total number of teams a team has to interact

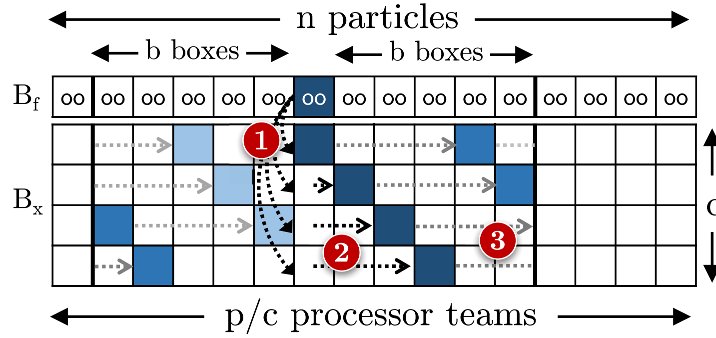



Figure 3.5: An illustration of Algorithm 2, the communication-avoiding algorithm for distance-limited interactions in one-dimensional simulation space. Labels (1) and (2) show the initial broadcast within a team and row-wise skewing. Label (3) shows the first of $2b/c$ shifts that "wrap around" at the cutoff radius.

---

**Algorithm 2** CA-1D-CUTOFF-2-BODY

---

**Input:** $b$, the number of processors spanned by the cutoff distance.
**Input:** Replication factor $c$, the number of extra copies of the particles.
**Input:** $P$, a processor grid of shape $p/c$-by-$c$.
**Input:** Set $Q$ of $n$ particles divided spatially among team leaders into local subsets $Q_t$.
**Output:** An updated set $Q$.

1: **for** $P(t, \ell)$ in parallel **do**
2: $P(t, 0)$ reads $Q_t$ into the fixed buffer $B_f$.
3: $P(t, 0)$ broadcast $B_f$ to $P(t, :)$.
4: Copy $B_f$ to exchange buffer $B_x$ of size $\lceil nc/p \rceil$.
5: Shift $B_x$ by $\ell$ along the $0^{\text{th}}$ dimension modulo the cutoff window.
6: **for** $2b/c$ **steps do**
7: Shift $B_x$ by $c$ along the $0^{\text{th}}$ dimension modulo the cutoff window.
8: Update particles in $B_f$ based on effect of $B_x$.
9: **end for**
10: **end for**
11: Sum-reduce updates within $P(t, :)$.

---

with is,

$$2b + 1 = 2\frac{\delta p}{wc} + 1 \tag{3.14}$$

Figure 3.5 illustrates the algorithm for simulations in 1D space. Algorithm 2 shows the pseudocode.

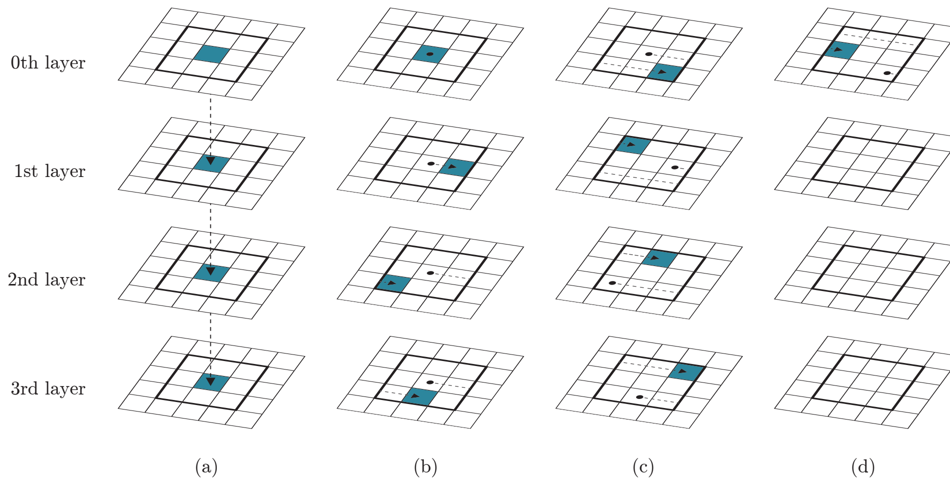



Figure 3.6: The communication-avoiding algorithm for distance limited interactions in two-dimensional simulation space. This example shows 100 processors arranged into 25 teams with 4 replication layers. The arrows indicate the transfer of particles between processors for different steps: (a) Broadcast among team members. (b) Initial skew. (c) Shifts during the second iteration. (d) Shifts during the third (final) iteration.

## Generalization for higher dimensional spaces

As in the 1D case, we assume a uniform particle distribution and a spatial decomposition of particles among processors teams. Broadcasts and reductions still occur along the $c$ dimension, and shifts of magnitude $c$ occur in the hyperplane perpendicular to the $c$ dimension. The easiest way to traverse the neighbors in the high-dimensional cutoff mesh is to linearize the space, calculate shifts in 1D, and map the shifting pattern back into the original space. Figure 3.6 shows shifts through a 2D space with 25 teams and a replication factor of 4. Communication avoidance becomes especially important in higher dimensions because the number of neighbors is exponential in the dimensionality of the problem space.

## Communication optimality

Analysis of the communication costs along the critical path is similar to that of the all-pairs interaction algorithm. The initial broadcast sends $O(cn/p)$ words to $c$ processors using $\log(c)$ messages, and the initial skewing sends $O(1)$ messages of size $O(cn/p)$ words. In a $d$-dimensional simulation space, each processor team has to interact with $(2b+1)^d = O(b^d)$ teams. Each team has $c$ processors, so this takes $O(b^d/c)$ shifting steps in which each processor sends $O(1)$ message of $O(cn/p)$ words, resulting in a total cost of $O(b^d/c)$ messages and $O(b^d n/p)$ words. The final reduction sends $O(cn/p)$ words in $\log(c)$ messages.

Dominated by the shifting costs, the total communication costs of the algorithm are,

$$S_{\text{CA-cutoff}} = O\left(\frac{b^d}{c}\right), \qquad W_{\text{CA-cutoff}} = O\left(\frac{b^d n}{p}\right) \tag{3.15}$$

Equation (3.9) gives the lower bounds for interactions with cutoff radius assuming each particle only needs to do $k$ interactions. Since $k/n = O(b^d/(p/c))$, rewriting Equation (3.15) with $b^d = O(kp/cn)$, we have

$$S_{\text{CA-cutoff}} = O\left(\frac{k}{n} \cdot \frac{p}{c^2}\right), \qquad W_{\text{CA-cutoff}} = O\left(\frac{k}{c}\right),$$

which match the communication lower bounds when $c = c_M$. This means our algorithms can always be communication optimal by using maximum memory available.

## Performance results

We extended our codes from Section 3.3 to support cutoff radius in 1D and 2D problem space with spatial decomposition. We ran our experiments on the same systems. Our setup ensures that the particle distribution remains nearly uniform throughout the simulation. We chose the cutoff radius to be 1/4 of the simulation space to allow reasonably many choices of $c$. Lastly, we did not utilize Blue Gene/P's topology-aware collectives because the communication pattern did not match the interconnect topology.

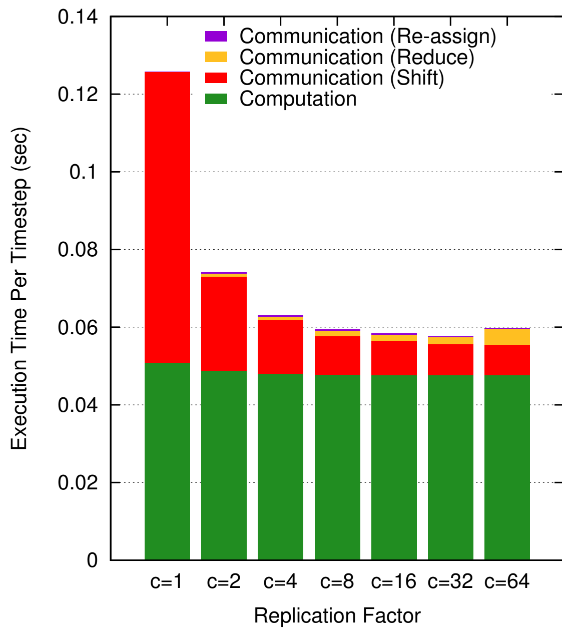
### Scaling the replication factor

Figure 3.7 shows the running time per timestep for varying replication factor $c$ in the 1D and 2D problem space. The additional cost of updating the spatial decomposition at every timestep is labeled as reassignment time in the plot.

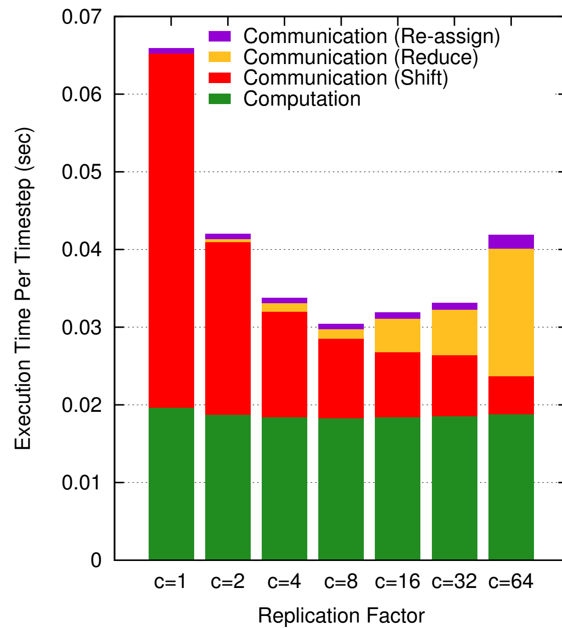
For small values of $c$, the plots show the expected decrease in communication time (between a factor of 1/2 and 1/4 as we double $c$). However, for large $c$ the cost of the reduction step grows considerably, yielding poorer overall performance than intermediate $c$ values. We believe this effect is primarily caused by collectives' inability to scale logarithmically when communication teams reach a certain size. Fortunately, our algorithm can be tuned to find the replication factor that provides the best balance between collective and point-to-point performance.

Costs due to shifting appear to stagnate after a few $c$ values, unlike in Section 3.3 where they approached zero. This is because of nontrivial load imbalance from our choice of boundary condition. Reflective boundary condition causes processors assigned to regions near the boundary of the simulation space have fewer interactions to compute than processors in the middle. Figure 3.8 shows an example in the first two rounds of calculation. This kind of load imbalance does not occur for interactions between all pairs because a spatial decomposition is not required in that case.

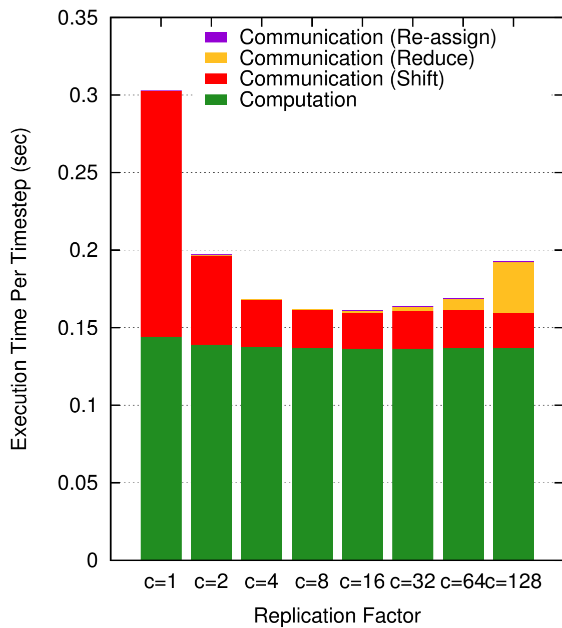
The average computation time is again lower at larger $c$'s because there are more particles to interact locally in each round, gaining better cache efficiency.



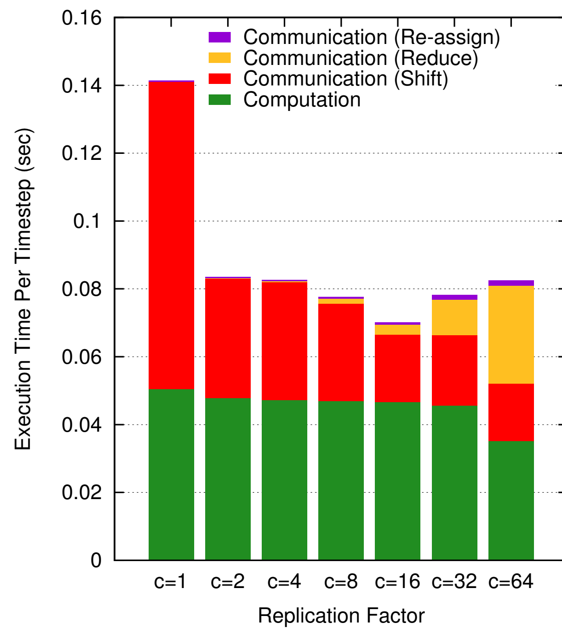

(a) 1D-cutoff, Cray XE-6,
24,576 cores, 196,608 particles.




(b) 2D-cutoff, Cray XE-6,
24,576 cores, 196,608 particles.




(c) 1D-cutoff, Blue Gene/P,
32,768 cores, 262,144 particles.




(d) 2D-cutoff, Blue Gene/P,
32,768 cores, 262,144 particles.

Figure 3.7: The effect of increased replication factors on execution time for 1D- and 2D-space simulations with a cutoff radius.

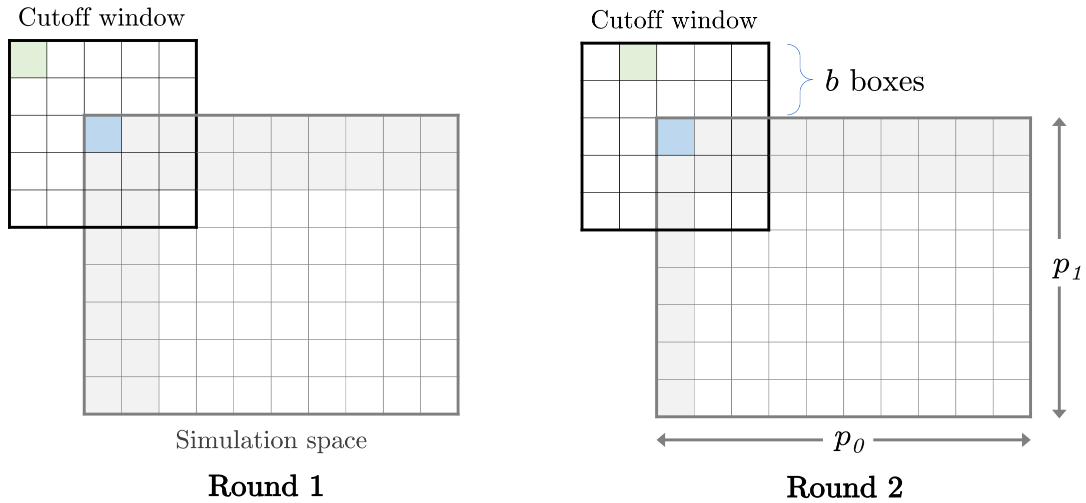
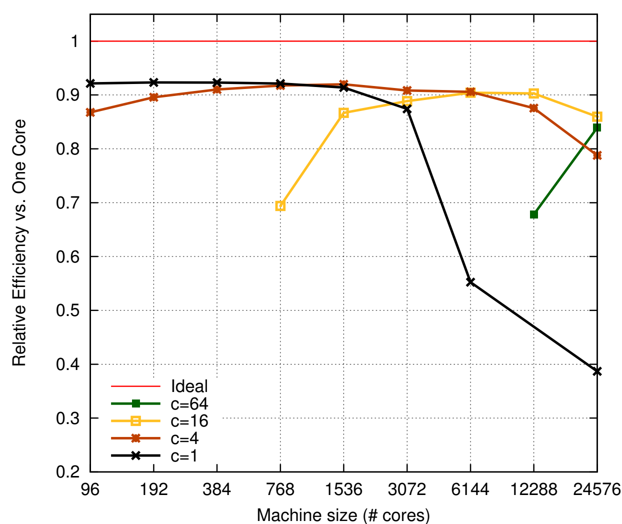



Figure 3.8: The first two rounds of a 2D space cutoff algorithm. The cutoff spans $b = 2$ boxes. The cutoff window shown is for the processor at the top left of the simulation space (in blue). Green color shows the position in the cutoff window that is being interacted with each processor. The area in gray indicates processors that are idle in that round.

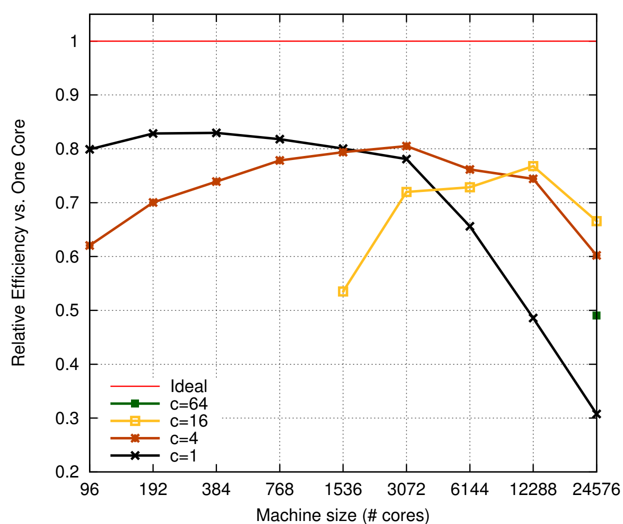
### Strong scaling performance

Figure 3.9 shows the strong scaling performance of 1D and 2D simulations with 196K particles on Cray XE-6 and 262K particles on Blue Gene/P. From the graphs, we make several observations. First, the largest available replication factor never gives best results. Second, there is a general trend that for a given replication factor, the algorithm exhibits sub-optimal performance on smaller machines due to load imbalance. Lastly, although we do not see clear benefits at small scale, the best replication of the communication-avoiding algorithm yields roughly double the efficiency of a non-replicating algorithm on the largest machine sizes (24K cores on Cray XE-6 and 32K cores on Blue Gene/P).

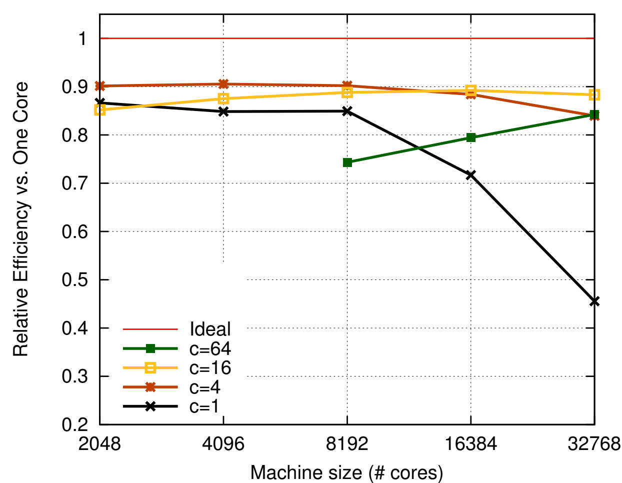
Our cutoff-distance results attain lower strong-scaling efficiency than the all-pairs case. We primarily attribute this fact to load imbalance caused by our choice of physical domain decomposition. More specifically, processors assigned to regions near the boundary of the simulation space have fewer interactions to compute than processors in the middle, leading to increased idle time and critical path length. Secondly, we did not use topology-aware collectives during the shift communication phase on Blue Gene/P; consequently, we utilized only half the bandwidth available to the experiment in Section 3.3 because we did not take advantage of the bidirectionality of the torus.



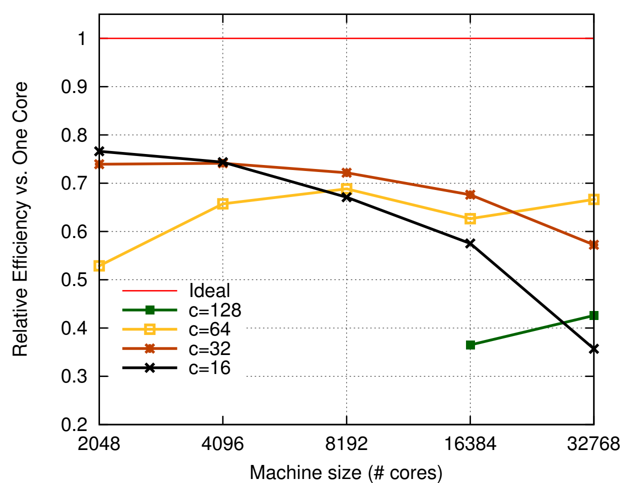

(a) 1D-cutoff, Cray XE-6, 196,608 particles.




(b) 2D-cutoff, Cray XE-6, 196,608 particles.




(c) 1D-cutoff, Blue Gene/P, 262,144 particles.




(d) 2D-cutoff, Blue Gene/P, 262,144 particles.

Figure 3.9: Strong scaling performance of 1D and 2D simulations with cutoff radius.

## 3.5 Taking Advantage of Symmetry

According to Newton's third law, the force on particle $i$ from particle $j$ is equal to the negative force on particle $j$ from particle $i$, $f_{ji} = -f_{ij}$. We can save the computation by a half if we only interact unique pairs of particles, i.e., $(i, j)$ and $(j, i)$ are the same pair. Our communication-avoiding algorithms only need two quick modifications to support this symmetry: (1) halving the number of rounds (assuming $2 \mid p/c^2$), and (2) making team leaders $P(t, 0)$ compute one additional round. Let $h = p/2c$ be half the number of processor teams. In this additional round, $P(i, 0)$ and $P(i+h, 0)$ for any $0 \leq i < h$ will have the same particle sets in their buffers: $Q_i$ and $Q_{i+h}$. So, we let $P(i, 0)$ compute the interactions of $Q_i$ with the first half of $Q_{i+h}$ and $P(i+h)$ compute the interactions of $Q_i$ with the latter half of $Q_{i+h}$. Algorithm 3 shows how to extend the communication-avoiding all-pairs algorithm in Algorithm 1. Figure 3.10 illustrates how each processor gradually fills the force interaction space. Similar modifications can be made for the cutoff version.

**Algorithm 3** CA-ALL-PAIRS-WITH-SYMMETRY

**Input:** Replication factor $c$, the number of extra copies of the particles.
**Input:** $P$, a processor grid arranged into $p/c$ teams by $c$ layers. Each team $P(t, :)$ has $c$ members, the first one being a team leader.
**Input:** Set $Q$ of $n$ particles divided evenly among team leaders into local subsets $Q_t$.
**Output:** An updated set $Q$.

1: **for** $P(t, \ell)$ in parallel **do**
2: $P(t, 0)$ reads $Q_t$ into the fixed buffer $B_f$.
3: $P(t, 0)$ broadcasts $B_f$ to $P(t, :)$.
4: Copy $B_f$ to exchange buffer $B_x$.
5: Shift $B_x$ by $\ell$ along the $0^{\text{th}}$ dimension.
6: **for** $1/2 \cdot p/c^2$ **steps do**
7: Interact particles in $B_f$ and $B_x$, updating both buffers.
8: Shift $B_x$ by $c$ along the $0^{\text{th}}$ dimension.
9: **end for**
10: **if** $t < p/2c$ **then**
11: $P(t, 0)$ interacts the first half of particles in $B_x$ with $B_f$, updating both buffers.
12: **else**
13: $P(t, 0)$ interacts the latter half of particles in $B_f$ with $B_x$, updating both buffers.
14: **end if**
15: Shift $B_x$ by $p/2c - \ell$ so $Q_t$ would be back in $B_x$ of team $t$.
16: **end for**
17: Sum-reduce updates within $P(t, :)$ from both buffers $B_f$ and $B_x$.

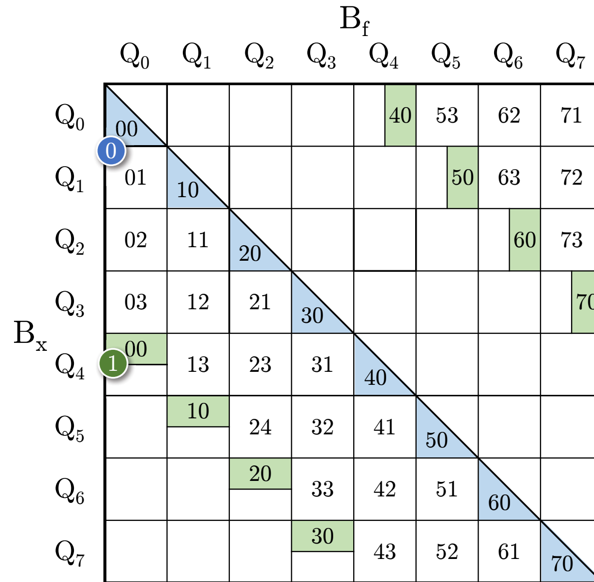



Figure 3.10: How processors partition the iteration space (force matrix) to take advantage of force symmetry. Here, $p = 32$ processors are divided into 8 teams of $c = 4$ team members. Each team $P(t,:)$ owns and is responsible for updating the particle set $Q_t$. Each number $t\ell$ inside each cell $(Q_x, Q_f)$ means processor $P(t, \ell)$ computes the interactions between $Q_x$ in the exchange buffer $B_x$ and $Q_f$ in the fixed buffer $B_f$. In round 0, each $P(t, \ell)$ interacts $Q_t$ with $Q_{t+\ell}$, covering the numbered cells in blue and white. $P(t, 0)$ then computes one additional round (the green cells), interacting half the interactions between $Q_t$ and $Q_{t+p/2c}$. The load is still balanced despite the additional round since, in round 0, $P(t, 0)$ computes interactions of particles in the same set $Q_t$ which is only half as many interactions as other team members.

## 3.6 Conclusions

We have presented an N-body algorithm for direct interactions that uses extra memory to replicate particles and asymptotically reduce communication. We analyzed the lower bounds on communication, and showed that the new algorithm is optimal in communication. Our new algorithm encompasses prior approaches, some of which also use replication, and degenerates to them in extreme cases. We also presented an experimental analysis on tens of thousands of cores for both BlueGene/P and Cray XE-6, which show that with the appropriate choice of replication factor, our algorithm achieves nearly perfect strong scaling by striking a balance between point-to-point and collective communication costs. One example shows a speedup of over 11.8× from communication avoidance. While the benefits of communication avoidance are best for small problems on large numbers of cores, the absolute communication overhead for the optimized algorithms is low, resulting in good absolute performance.

While the theoretical analysis would suggest maximizing the amount of replication to $\sqrt{p}$ if memory is available, we found this was not always optimal in practice. We, therefore, leave as open the question of how to select the replication factor $c$, which is typically close to the $\sqrt{p}$ limit and can be autotuned at runtime by trying multiple factors. Even using the maximum value of $c$ may be acceptable: for the all-pairs algorithm, the best value of $c$ differed by no more than 16% in any experiment, and most experiments revealed less than 2% difference.

At the time our work was first published, it was one of the early applications of communication-avoidance theory beyond numerical linear algebra. This suggests a general strategy for communication avoidance through replication, a technique used previously to increase parallelism or decrease synchronization [129]. Since then, the theory has been generalized to cover any loop nests with subscripts that are affine functions of the loop indices [47, 107, 48]. We also further applied the replication technique to the many-body problem, which is presented in Chapter 4.

# Chapter 4

# $k$-way N-Body

The parallel $k$-way N-body ($k$-body) problem is relatively unexplored compared to the 2-body problem discussed in Chapter 3. Molecular dynamics applications traditionally focused on pairwise interactions because they yield qualitatively valid results for most applications, and $k$-body's $O(n^k)$ complexity was infeasible to compute on supercomputers decades ago. However, some phenomena can only be captured through many-body potentials, and there has been increasing interest in $k$-body, especially 3-body, interactions. This chapter focuses on the 3-body problem then generalizes to the $k$-body problem with cutoff distance.

Many-body potentials are significant in both atomic systems and molecular systems [69]. They enable more accurate modeling of specific materials such as metals and ceramics [151]. 4-body potentials are used in protein folding [79, 41]. 3-body potentials are the most commonly used after 2-body potentials and are required to capture some properties such as phase equilibria in noble gases [131] and the second virial coefficient in systems such as water [133, 144].

The parallel 3-body problem poses a complicated symmetry challenge. It is trivial to exploit the force symmetry $f_{ij} = -f_{ji}$ in the 2-body problem, and the cost of ignoring symmetry is only a factor of two. However, in the 3-body problem, there are $n(n-1)(n-2)/6$ unique triplets, so there are roughly 6 times more triplet interactions if we do not eliminate redundant triplets. At $O(n^3)$ scale, this is costly. It is even worse for larger values of $k$ as the symmetry factor grows with $k!$. Multiple efforts have been put into avoiding redundancy and balancing work over processors, but none of these consider the communication complexity of the algorithms. Even though the problem is compute-intensive, communication takes time and energy and still comes into play when scaling aggressively. Our goal is to develop algorithms that are optimal in both computation (exploitation of symmetries and load balance) and communication.

We present two communication-avoiding algorithms that are both computation and communication optimal, one for all-triplets interactions and the other for interactions with cutoff

This chapter is based on work previously published in "A Computation- and Communication-Optimal Parallel Direct 3-Body Algorithm"[109].

distance. We derive lower bounds for communication along the critical path and prove that both algorithms achieve their bounds. By making $c$ replicas, they can reduce by factors of $c^3$ latency (number of messages) and $c^2$ bandwidth (number of words). We also present experimental results of the all-triplets algorithms on two large-scale machines.

Our contributions in this chapter are:

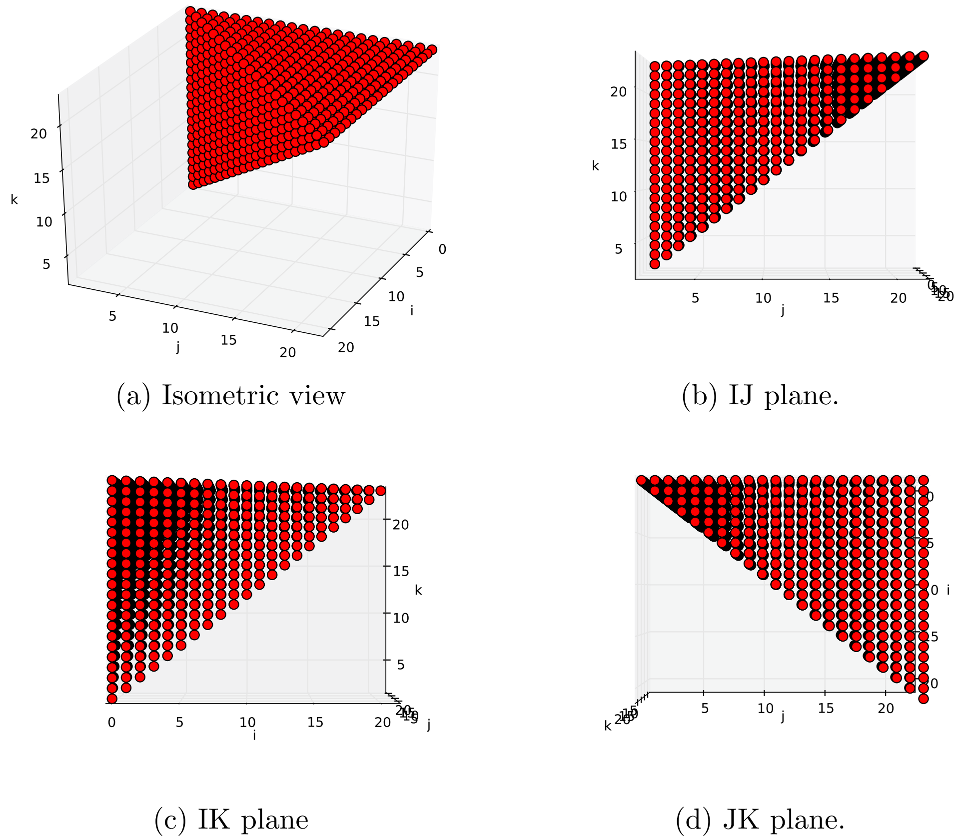
- Derivation of the communication lower bounds for $k$-body methods ($k \geq 2$) with or without cutoff.
- A new *all triplets* algorithm for 3-body calculations without cutoff that is provably optimal in computation and communication and has provably better load balance than previous work.
- A new class of algorithms for any $k$-body interaction ($k \geq 2$) with cutoff, where the cutoff limits interactions to less than 1/3 of the domain. These algorithms are also provably optimal in communication and computation.
- Application of replication to further avoid communication asymptotically.
- An implementation of the 3-body algorithm for massively parallel machines, including support for hybrid (shared and distributed memory) parallelism.
- Performance results showing near perfect strong scaling on tens of thousands of cores and the tradeoff between communication and load imbalance of the communication-avoiding technique

This chapter is organized as follows. Section 4.1 describes previous many-body algorithms. Section 4.2 derives communication lower bounds for the all-triplets problem. Section 4.3 gives the all-triplets algorithm and extends it to support replication. Section 4.4 shows performance results of the all-triplets algorithm. Section 4.5 adds cutoff, generalizes to any $k$-body interaction, and proves the communication lower bounds and optimality. Section 4.6 concludes and discusses future work.

## 4.1 Background and Previous Work

### 3-Body and other $k$-body Algorithms

The work on the parallel 3-body problem has begun rather recently, with the earliest papers in 1993 [141]. Similar to the 2-body problems, there are direct and approximate approaches, most of which took a direct approach. Nakano et al. [141] proposed a domain decomposition algorithm with multiple-time-step (MTS) to compute 2- and 3-body forces with cutoff distance. The MTS method divided the force on a particle into two components, primary and secondary. The primary force comes from interactions with particles within radius $r$ and is rapidly varying. The secondary force comes from interactions from particles outside the radius and is relatively slowly changing. The primary force is calculated every timestep, while the secondary force is calculated every $n$ timesteps where $5 \leq n \leq 15$, i.e., it has its




(a) Isometric view

(b) IJ plane.

(c) IK plane

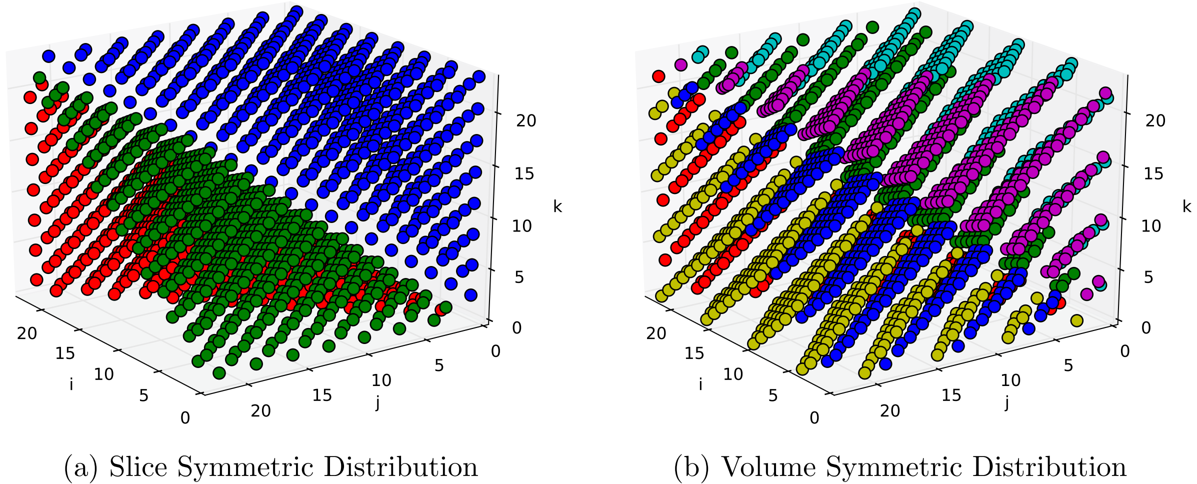
(d) JK plane.

Figure 4.1: Illustration of the 3-body force cube of a system of $n = 24$ particles. Shown in red is the set of all unique elements where $(i < j < k)$.

own *larger* timestep. They used a separable 3-body calculation, i.e., decomposing a 3-body force calculation $(i, j, k)$ to two 2-body interactions $(i, j)$ and $(i, k)$.

Li et al. [121, 122, 123] extended the 2-body's $n$-by-$n$ *force matrix* [150] to an $n$-by-$n$-by-$n$ *force cube*, as shown in Figure 4.1. The goal is to compute only unique triplets $(i, j, k)$ where $i < j < k$, shown in red circles forming a tetrahedron. They presented four ways to assign work to processors: Force, Cyclic, Balanced Cyclic, and Precise Decompositions. **Force Decomposition (FD)** [122] partitions the force cube into subcubes, prunes redundant subcubes from the job list, then assigns each subcube to a processor. Since not all subcubes have the same computation load – subcubes $(i, j, k)$ where $i \neq j \neq k$ have the most load and $i = j = k$ the least, – the FD algorithm suffered the most load imbalance among all the algorithms they considered.

**Cyclic Decomposition (CD)** [121, 122, 123] slices the force cube into $n$ planes and assigns planes cyclically to processors (processor $r$ gets planes $r, r + p, r + 2p$, etc.). It still incurs a slight load imbalance as processors that are assigned earlier planes always have less load than processors with later planes. This led to **Balanced Cyclic Decomposition**




(a) Slice Symmetric Distribution

(b) Volume Symmetric Distribution

Figure 4.2: Two ways to select unique triplets to process by Sumanth et al. [169]. Different colors are used for illustration purposes only and have no special meaning, i.e., they do not represent how processors partition work.

**(BD)** [122, 123], which cyclically assigns odd planes first, then cyclically assigns even planes in reverse order to balance out excess load.

**Precise decomposition (PD)** [122] counts exactly how many elements are to be computed in each plane and assigns consecutive planes that have closest to perfect load to each processor. All of the latter three algorithms performed marginally better than FD. Still, none of them achieved high parallel efficiency. PD, BD, and CD were at 50% parallel efficiency while FD was at 40% on a system with only 35 processors.

Also based on partitioning the force cube, Sumanth et al. [169] exploited the symmetry to select a different subset of triplets, rather than the tetrahedron-shaped subset, to compute so that there is an equal number of unique elements to compute in each plane (Slice Symmetric) or volume space (Volume Symmetric), as shown in Figure 4.2. Circles are the force elements to be processed. Colors are for illustration purpose only, i.e., they do not indicate how processors partition work. Figure 4.2a selects the triplets so each plane has the same number of triplets. Processors just need to process the same number of planes to be load balanced, regardless of how the planes are assigned to processors. Figure 4.2b selects the triplets that are uniformly distributed throughout the cubes. To maintain load balance, all processors only need to compute the same number of subcubes. Nevertheless, very few performance results are presented, most of which focused on load balance; none mentioned communication time. The communication is not likely to scale well given that each plane involves all particles. They stated that cutoff distance can be supported but did not provide details.

Kunaseth et al. [113] proposed a systematic way to compute $k$-body interactions with cutoff ($k$-tuple computation in their terminology). Their *Shift-Collapse* algorithm uses a cell-based method: (1) partition simulation space volume into disjoint cells (2) bin particles

geometrically into cells, (3) generate a cell search-space containing all tuples that must be computed based on the interaction/cutoff rules, and (4) prune out redundant $k$-tuples. Despite the excessive work of generating all possible $k$-level neighbor cell combinations, their algorithm performed very well at large scale.

Our work addresses these direct interaction methods, but there are also approximate algorithms for $k$-body scenarios [118], although none address parallelization.

## 4.2 Communication Lower Bounds

Recall the general lower bounds on latency S and bandwidth W from Equation (2.2),

$$S = \Omega\left(\frac{Z}{F}\right), \qquad W = \Omega\left(\frac{Z \cdot M}{F}\right).$$

Let $n$ be the total number of particles in the system and $p$ be the number of processors. The direct 3-body problem requires $O(n^3)$ interactions, so each processor has to do $Z = O(n^3/p)$ work. If each processor has a memory of size $M$ words, it can perform at most $F = O(M^3)$ useful work.

**Theorem 4.1.** *The maximum number of 3-body force evaluations that can be computed with $M$ particles is $O(M^3)$.*

*Proof.* The interaction space is a force cube where each coordinate $(i, j, k)$ represents the interactions between particles $i, j$ and $k$. We would like to upper-bound the size of $V$, the set of interactions a processor can compute with $O(M)$ particles in memory. Let $V_i$, $V_j$, and $V_k$ be the set of indices $i$, $j$, and $k$ in the set $V$. Denote the cardinality of a set by $|\cdot|$. Then,

$$|V| \leq |V_i||V_j||V_k|,$$

since that is the maximum number of Cartesian products possible (a special case of Loomis and Whitney's inequality [127]). Because $V_i$, $V_j$, and $V_k$ must fit in memory, $|V_i|$, $|V_j|$, $|V_k| \leq O(M)$, therefore, $|V| \leq O(M^3)$. □

Thus, the lower bounds for the 3-body problem are,

$$S = \Omega\left(\frac{n^3/p}{M^3}\right), \qquad W = \Omega\left(\frac{n^3/p}{M^2}\right). \tag{4.1}$$

We write $M$ as a multiple of the minimum number of particles each processor must store $(n/p)$: $M = c_M n/p$ where $c_M \in \mathbb{Z}^+$. The lower bounds become,

$$S = \Omega\left(\frac{p^2}{c_M^3}\right), \qquad W = \Omega\left(\frac{np}{c_M^2}\right). \tag{4.2}$$

For the memory-independent lower bounds, let $V$ be the set of interactions the processor with the largest load has to compute. $|V|$ cannot be less than the total amount of work divided equally among processors,

$$|V| \geq \frac{n^3}{p}. \tag{4.3}$$

We relate $|V|$ to the total input size $|V_i| + |V_j| + |V_k|$ necessary to compute $V$,

$$\begin{aligned} |V| &\leq |V_i||V_j||V_k| \\ &\leq \frac{1}{3!}(|V_i| + |V_j| + |V_k|)^3. \end{aligned} \tag{4.4}$$

Putting Equations (4.3) and (4.4) together, we have,

$$\begin{aligned} \frac{1}{3!}(|V_i| + |V_j| + |V_k|)^3 &\geq \frac{n^3}{p} \\ |V_i| + |V_j| + |V_k| &\geq \sqrt[3]{3!}\frac{n}{\sqrt[3]{p}}. \end{aligned}$$

Since $V_i$, $V_j$, and $V_k$ can all be the same, the processor must hold at least $(\sqrt[3]{3!}\, n/\sqrt[3]{p})/3$ particles. Assuming every processor starts with just $n/p$ particles in their memory, they must communicate at least $(\sqrt[3]{3!}\, n/\sqrt[3]{p})/3 - n/p$ words, using at least 1 messages. Therefore, the memory-independent lower bounds are,

$$S = \Omega\left(1\right), \qquad W = \Omega\left(\frac{n}{\sqrt[3]{p}}\right), \tag{4.5}$$

equivalent to the memory-dependent lower bounds when $c_M = p^{2/3}$. Any $c_M > p^{2/3}$ gets the same lower bounds in Equation (4.5).

## 4.3 Algorithms

Instead of visually partitioning the force cube, our approach starts from extending Algorithm 1 (without replication) from Chapter 3 or the systolic/ring algorithm in [92] by adding a buffer to hold the third body in a triplet. We start by forming a naïve algorithm that is load-balanced but does not utilize symmetries and is not communication-optimal. We then progressively improve the other two aspects in the next three versions. The first version (All-Unique Triplets) removes computational redundancy while retaining load balance. The second version merges some computations in the first version together to save a constant factor of communication. The third version applies replication to the second version to reduce the communication asymptotically. This last version achieves all three of our goals (load balance, redundancy freedom, and communication optimality).

**Algorithm 4** NAÏVE ALL-TRIPLETS

**Input:** $P$, a ring of $p$ processors.
**Input:** Set $Q$ of $n$ particles divided into $p$ equal subsets $Q_i$.
**Output:** The updated set $Q$.

```
 1: for P(i) in parallel do
 2:     Read Q_i into B_0 and copy them into B_1 and B_2.
 3:     for p steps do
 4:         for p steps do
 5:             Update particles in B_0 based on B_1 and B_2.
 6:             Shift B_2 by 1.
 7:         end for
 8:         Shift B_1 by 1.
 9:     end for
10: end for
```

## Naïve all-triplets algorithm

Let $Q$ be the set of all particles in the system. We divide $Q$ into $p$ equal disjoint subsets of $n/p$ particles, $Q_0, Q_1, \ldots, Q_{p-1}$ (no replication yet). We arrange the processors into a ring (1D torus). Processor $P(i)$ owns and updates the particles in $Q_i$, i.e., keeping track of forces applied to them and moving them accordingly in each timestep. Each processor has three buffers, $B_0$, $B_1$, and $B_2$, to hold the particle subsets $(Q_i, Q_j, Q_k), 0 \leq i, j, k < p$.[1] The algorithm simply mimics a sequential three-nested loops code. The buffer $B_0$ is fixed and $P(i)$ goes through all possible pairs of $(Q_j, Q_k)$ by shifting buffers $B_1$ and $B_2$ around. Particle interactions for each triplet of subsets $(Q_i, Q_j, Q_k)$ are computed by forming all possible triplets using one particle from each buffer, say, $B_0[x], B_1[y]$, and $B_2[z]$, calculating all 3-body forces on $B_0[x]$ from $B_1[y]$ and $B_2[z]$, and accumulating the forces within $B_0[x]$. The algorithm alternates between computing interactions and exchanging particle subsets with other processors, as shown in Algorithm 4.

Because all processors do the same amount of work in between every shift, the algorithm is load balanced. However, it ignores symmetries and computes all forces redundantly. Communication-wise, it shifts $O(n/p)$ words for $O(p^2)$ rounds with at most 2 messages per round. Therefore the costs are,

$$S_{\text{naïve}} = O(p^2), \qquad W_{\text{naïve}} = O(np), \tag{4.6}$$

achieving the lower bounds in equation (4.2) only when $c_M = 1$. In other words, it is not communication-optimal if the memory has space to store extra particles ($c_M \geq 2$).

[1] This is *not* replication. There are always 3 buffers for the 3-body problem and we treat this factor of 3 as a constant, i.e., each processor stores $O(n/p)$ when it does not replicate.

## All-unique-triplets algorithm

Next, we utilize symmetry by calculating all forces related to all particles in a triplet at once. For this, we need to form only unique subsets of particles. If our algorithm forms a triplet $(Q_i, Q_j, Q_k)$, it should never form any other five permutations, $(Q_i, Q_k, Q_j)$, $(Q_j, Q_i, Q_k)$, $(Q_j, Q_k, Q_i)$, $(Q_k, Q_i, Q_j)$, and $(Q_k, Q_j, Q_i)$.

First, we confirm that none of the processors can process the same triplet in the same round, with only one minor exception.

**Lemma 4.2.** *If $P(i)$ has $Q_x$ in a buffer $B \in \{B_0, B_1, B_2\}$, $P(i+d)$ has $Q_{x+d}$ in its $B$.*

*Proof.* Algorithm 4 starts with $P(i)$ holding $Q_i$ in all buffers, so the condition is true in the beginning. Assume that $P(i)$ has $Q_x$ and the condition is true before a shift. Shifting by a distance $s$ causes $P(i)$ to take the particle subset $Q_{x-s}$ from $P(i-s)$ and $P(i+d)$ to take $Q_{x+d-s}$ from $P(i+d-s)$. Since the condition holds after one shift, it holds throughout the algorithm by induction. □

**Lemma 4.3.** *At any point in time, all processors hold different triplets of particle subsets, except when $3 \mid p$ and the triplet is of the form $(Q_x, Q_{x+p/3}, Q_{x+2p/3}), 0 \leq x < p$.*

*Proof.* We will prove by contradiction. Assume that $P(i)$ and $P(j), j \neq i$ have the same triplet. Let the particle subset indices at $P(i)$ be $x_0, x_1$, and $x_2$ where $0 \leq x_0 \leq x_1 \leq x_2 < p$. Write $P(j)$ as $P(i+d), 0 \leq d < p$ then by Lemma 4.2, $P(j)$ has the subset indices $x_0+d, x_1+d$, and $x_2 + d$. Since the indexing wraps around, there are three cases:

**Case 1:** $(x_0 + d) \bmod p \leq (x_1 + d) \bmod p \leq (x_2 + d) \bmod p$
The following conditions must hold for $P(i)$ and $P(j)$ to have equivalent triplets,

$$x_0 + d \equiv x_0 \pmod{p},$$
$$x_1 + d \equiv x_1 \pmod{p},$$
$$x_2 + d \equiv x_2 \pmod{p}.$$

This means $d \in \{0, p, 2p, \ldots\}$, which contradicts the assumption that $0 < d < p$.

**Case 2:** $(x_2 + d) \bmod p \leq (x_0 + d) \bmod p \leq (x_1 + d) \bmod p$
The following conditions must hold for $P(i)$ and $P(j)$ to have equivalent triplets,

$$x_2 + d \equiv x_0 \pmod{p}, \tag{4.7}$$
$$x_0 + d \equiv x_1 \pmod{p}, \tag{4.8}$$
$$x_1 + d \equiv x_2 \pmod{p}. \tag{4.9}$$

Summing Equations (4.7), (4.8) and (4.9) together, we get,

$$3d \equiv 0 \bmod p,$$

meaning $d \in \{0, p/3, 2p/3, ...\}$. $d = p/3$ and $2p/3$ only happen when $3 \mid p$ and both imply that the triplet is of the form $Q_x, Q_{x+p/3}, Q_{x+2p/3}$, as stated in the lemma. The other values of $d$ contradict the assumption that $0 < d < p$.

**Case 3:** $(x_1 + d) \bmod p \leq (x_2 + d) \bmod p \leq (x_0 + d) \bmod p$
The following conditions must hold for $P(i)$ and $P(j)$ to have equivalent triplets,

$$x_1 + d \equiv x_0 \pmod{p}, \tag{4.10}$$
$$x_2 + d \equiv x_1 \pmod{p}, \tag{4.11}$$
$$x_0 + d \equiv x_2 \pmod{p}. \tag{4.12}$$

Summing Equations (4.10), (4.11) and (4.12) together, we get,

$$3d \equiv 0 \bmod p$$

The same analysis as Case 2 follows. □

Lemma 4.3 ensures that, if all processors start with unique particle subset triplets and all particle exchanges are done by shifting, no two processors can hold the same triplet in the *same* round. Next, we need to find a shifting pattern that avoids redundant triplets *across* rounds. Algorithm 5 makes two changes to Algorithm 4 to make this possible. First, instead of fixing $B_0$ and shifting just $B_1$ and $B_2$, it shifts all three buffers. Second, it changes the number of times a particular buffer is to be shifted before switching to another buffer. The

**Algorithm 5** ALL-UNIQUE-TRIPLETS

**Input:** $P$, a ring of $p$ processors.
**Input:** Set $Q$ of $n$ particles divided into $p$ equal subsets $Q_i$.
**Output:** The updated set $Q$.

```
 1: for P(i) in parallel do
 2:     Read Q_i into B_0 and copy them into B_1 and B_2.
 3:     b ← 2                          // The 2nd buffer is picked for illustration purpose.
 4:     for each phase d ∈ {0, 1, ..., ⌊p/3⌋ − 1} do
 5:         for p − 3d rounds do
 6:             Update all 3-body interactions in B_0, B_1 and B_2.
 7:             Shift B_b by 1.
 8:         end for
 9:         b ← (b + 1) mod 3          // Switch buffer to shift.
10:     end for
11:     if 3 | p then                  // Special case.
12:         Calculate the ⌊i/(p/3)⌋^th third of the interactions of the three buffers.
13:     end if
14: end for
```

new shifting scheme is surprisingly simple: shift $B_2$ by one $p$ times, shift $B_0$ by one $p-3$ times, shift $B_1$ by one $p-6$ times, then back to shifting $B_2$ by one $p-9$ times, keep shifting buffers in a round-robin manner until the number of times to shift is $p \bmod 3$. (The buffers can be picked in any order in the beginning, but must reoccur in the same order until the end.) If $p$ is a multiple of 3, one additional round is required at the end.

Figure 4.3 shows the complete system status of Algorithm 5 for $p = 9$ processors. The number at the beginning of each row is absolute round number. Nine column groups on the right side represent the nine processors. The three columns in each group list what particle subsets (the number $i$ of $Q_i$) are in $B_0, B_1$, and $B_2$, respectively. On the left are boxes and dashes depicting $(Q_i, Q_j, Q_k)$ that are in buffers $B_0, B_1$, and $B_2$ of $P(0)$. Red, green, and blue boxes represent $B_0, B_1$, and $B_2$, respectively. Dashes mean not in the buffer. For example, at round 10, the red, green, and blue boxes are at indices 8, 0, and 1, meaning $P(i)$ has $(Q_{i+8}, Q_{i+0}, Q_{i+1})$. We call triplets of distances like $(8, 0, 1)$ offset patterns (sometimes we will write just 801 for brevity). The algorithm consists of $\lfloor p/3 \rfloor$ phases. In phase $d$, a buffer is shifted by one $p - 3d$ times. At the end of each phase, a new buffer is shifted.

In the first round, each processor calculates interactions between its own particles. In rounds 2-9, it calculates interactions between two of its own particles and those of succes-

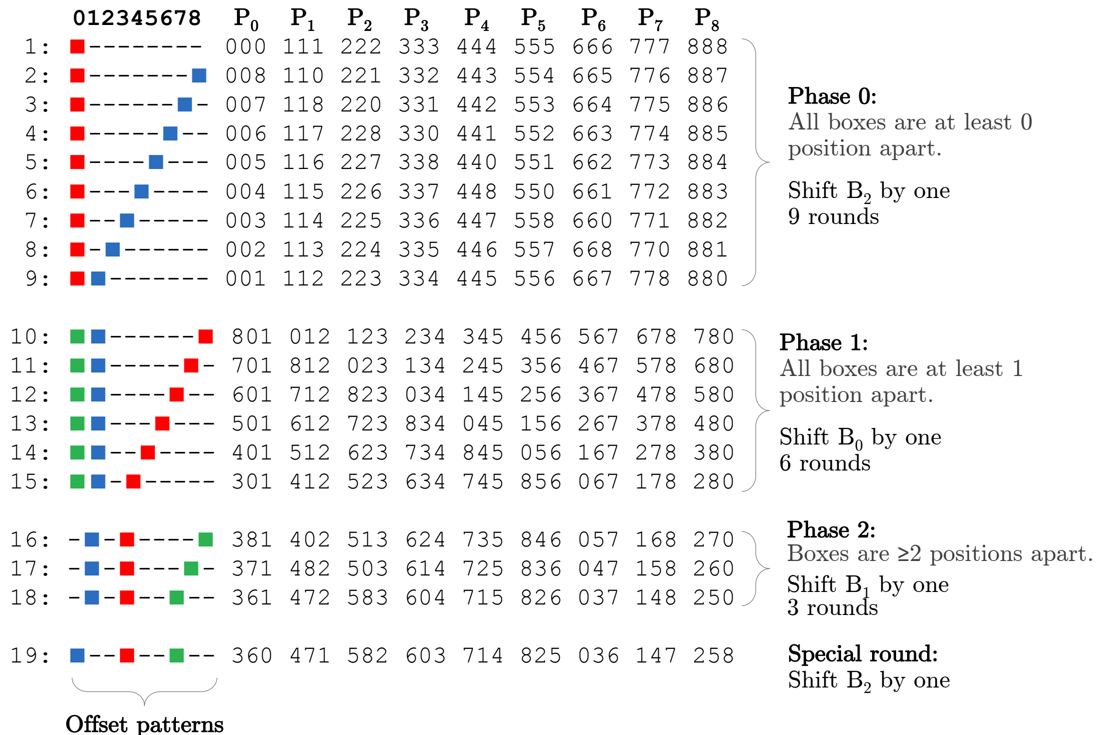



Figure 4.3: Illustration of the all-unique-triplets algorithm. There are 19 rounds; each row shows what parts of particles are in what buffer in each round. Offset patterns are drawn in read, green, and blue on the left side. In the first nine rounds, the red and green boxes are both in processor 0.

sive left neighbors, starting with the immediate left and moving further left cyclically until reaching its right neighbor. Then it switches to shifting the red buffer ($B_0$) to avoid getting a repeat offset pattern.

During the next $9 - 3 = 6$ rounds, interactions between itself and its right neighbor are calculated with the moving buffer, starting from its left neighbor and stopping short before reaching the right neighbor to avoid getting the same offset pattern as round 10 (recall that the permutation does not matter).

Then, the green buffer is shifted 3 times to compute interactions of particle subsets that are at least one neighbor apart. Again, it has to stop at round 18 or it will generate the same offset patterns as round 16. Finally, the blue buffer is shifted to cover the pattern with equally spaced boxes, which is the special case when $p$ is divisible by 3. Notice that $P(i)$ has the same triplet of particle subsets as those of $P(i+p/3)$ and $P(i+2p/3)$. To maintain load balance, let each third of the processors calculate each third of the interactions in this case. ($P(i)$ calculates the $\lfloor \frac{i}{p/3} \rfloor^{\text{th}}$ third of interactions.)

After computing all interactions, the particle subsets are sent back to their owner processors, which will combine all interactions and update their positions. See Algorithm 5 for pseudocode.

## Correctness

Here we provide a formal proof that Algorithm 5 generates every unique triplet of particle subsets $Q_0, Q_1, \ldots, Q_{p-1}$, and that no redundancy occurs apart from the special case when $3 \mid p$. Lemma 4.4 shows that phase $d$ generates all offset patterns that all boxes are at least $d$ apart, with two boxes exactly $d$ apart. We will look at this as a balls-and-bins problem, but the bins will be organized cyclically, i.e., as a ring.

**Lemma 4.4.** *Given a cyclic list of $p$ bins, $b_0, b_1, \ldots, b_{p-1}$, 3 balls (R, G, and B) and a fixed distance $d$, there are exactly $p - 3d$ possible ways of placing the balls into bins such that $R$ and $G$ are distance $d$ apart and $B$ is at least $d$ away from both $R$ and $G$.*

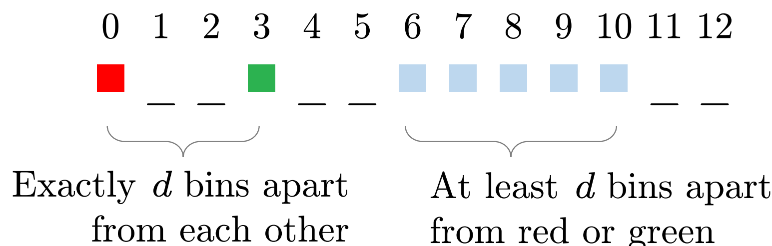



Figure 4.4: The number of ways to put $R$ and $G$ exactly $d$ apart and $B$ at least $d$ bins away from the first two. ($d = 2$ in this case.)

*Proof.* Figure 4.4 shows an example when $p = 12$. $R$, $G$, and $B$ are shown as red, green, and blue squares, respectively. Let us put $R$ at position 0, $G$ at position $d$, then the $B$ can be from position $2d$ to $p-1-d$. Any position greater than $p-1-d$ will make $B$ less than $d$ away from $R$. There are $(p-1-d)-2d+1 = p-3d$ positions, therefore there are $p-3d$ patterns. □

Next, we show in Lemma 4.5 that Algorithm 5 will not generate redundant patterns and prove the algorithm correctness in Theorems 4.6 and 4.7.

**Lemma 4.5.** *Throughout all rounds, Algorithm 5 does not generate redundant offset patterns.*

*Proof.* The algorithm generates all offset patterns following the proof of Lemma 4.4, starting from $d = 0$ to $d = \lfloor p/3 \rfloor - 1$. In the same phase $d$, one buffer is moving by the other two is fixed, so no offset pattern will occur twice. Between different phases $d_1 < d_2$, since all boxes in an offset pattern in phase $d_2$ are at least $d_2 > d_1$ apart, the offset pattern cannot be equivalent to those of phase $d_1$ in which at least two boxes are $d_1$ apart. □

**Theorem 4.6.** *No redundant triplets (2 or more triplets that are permutations of each other) will be produced by Algorithm 5, with the exception of the special case in lines 11-13.*

*Proof.* We will prove by contradiction. Assume that Algorithm 5 generates redundant triplets, then there are three possible cases:

**Case 1:** *The redundant triplets come from different processor during the same round (the same offset pattern).*
By Lemma 4.3, no redundancy *within* an offset pattern can occur, excluding the spacial case. So, this case cannot happen.

**Case 2:** *The redundant triplets come from the same processor across different rounds.*
By Lemma 4.5, this is not possible.

**Case 3:** *The redundant triplets come from different processors across different rounds.*
Let $P(i)$ and $P(j)$ be the two processors that have the same triplet $(Q_a, Q_b, Q_c)$. We can express the triplet indices as terms of offsets $(x, y, z)$ from $P(i)$ and $(m, n, o)$ from $P(j)$,

$$i + x \equiv j + m \pmod{p},$$
$$i + y \equiv j + n \pmod{p},$$
$$i + z \equiv j + o \pmod{p},$$

which gives us,

$$i - j \equiv m - x \pmod{p} \equiv n - y \pmod{p} \equiv o - z \pmod{p}.$$

This means $(x, y, z)$ and $(m, n, o)$ are the same offset pattern and that the algorithm will only pick one of them by construction. Hence, no redundancy occurs. □

**Theorem 4.7.** *Algorithm 5 generates all unique triplets.*

*Proof.* There are $p+p(p-1)+\binom{p}{3}$ unique triplets, $p$ for $(Q_i, Q_i, Q_i)$, $p(p-1)$ for $(Q_i, Q_i, Q_j)$ where $i \neq j$, and $\binom{p}{3}$ for $(Q_i, Q_j, Q_k)$ where $i \neq j \neq k \neq i$. We will show that the algorithm generates exactly this many unique triplets. In phase 0, there are $p$ rounds, the first round generating $p$ unique triplets of the form $(Q_i, Q_i, Q_i)$ and the rest $p-1$ rounds generating $p(p-1)$ unique triplets of the form $(Q_i, Q_i, Q_j)$. Now, we count the total number of box patterns generated by the algorithm from phase 1 onward, excluding the special pattern $(0, p/3, 2p/3)$, and show that they generate the rest $\binom{p}{3}$ unique triplets. Let $q$ and $r$ be the quotient and remainder of $p/3$, so $p = 3q + r$ where $q = \lfloor p/3 \rfloor$ and $r \in \{0, 1, 2\}$.

$$\begin{aligned}
\sum_{d=1}^{q}(p-3d) &= \sum_{d=1}^{q} p - 3\sum_{d=1}^{q} d \\
&= pq - \frac{3q(q+1)}{2} \\
&= p\left(\frac{p-r}{3}\right) - \frac{1}{2}\cdot(p-r)\left(\frac{p-r}{3}+1\right) \\
&= \frac{1}{6}\cdot(p-r)(p+r-3) \\
&= \begin{cases} \dfrac{p^2-3p}{6} & \text{if } 3 \mid p \\ \dfrac{(p-1)(p-2)}{6} & \text{otherwise} \end{cases}
\end{aligned}$$

Each pattern creates $p$ unique triplets. When $3 \nmid p$, the algorithm produces

$$\frac{(p-1)(p-2)}{6}\cdot p = \binom{p}{3}$$

unique triplets. When $3 \mid p$, the algorithm does one extra round and forms $p/3$ additional unique triplets. Therefore, it generates a total of

$$\frac{p^2-3p}{6}\cdot p + p/3 = p\left(\frac{p^2-3p+2}{6}\right) = \frac{p(p-1)(p-2)}{6} = \binom{p}{3}$$

unique triplets. This completes the proof. □

## Computation Optimality

There are two kinds of computation optimality we are looking for: avoiding non-redundant computation and ensuring load balance. Lemma 4.6 indicates that no triplets are repeated and therefore proves the first point.

There are three possible workload numbers, depending on the offset pattern. Let $m = n/p$ be the number of particles each processor owns. If all three buffers contain same particle

subsets, e.g., offset pattern 000, the load is $\binom{m}{3}$. If two buffers have same particle subsets and the other is different, e.g., offset pattern 001, the load is $m\binom{m}{2}$. Otherwise, the load is $m^3$.

However, since all processors are always working on the same offset pattern, the load will always be balanced, assuming $n$ is a multiple of $p$ for simplicity. If $p \nmid n$, there will be a negligible load imbalance as we make $n$ mod $p$ processors hold $\lceil n/p \rceil$ particles and the rest holds $\lfloor n/p \rfloor$ particles.

## Communication Optimality

The algorithm sends a message each round for $\sum_{d=1}^{q}(p-3d) = O(p^2)$ rounds, therefore a total of $O(p^2)$ messages. Each message contains $n/p$ words, so the bandwidth used is $O(p^2 \cdot n/p) = O(np)$. These costs match the lower bounds derived in equation (4.2) with $c = 1$ (no replication, $M = 1 \cdot n/p$) so the algorithm is communication optimal if and only if the memory is not large enough to replicate.

Still, the bounds only indicate optimality in an asymptotic sense. Algorithm 5 can save at least a constant factor more; we can calculate the case where more than one particle is from the same particle subset without really having to store them in multiple buffers, i.e., processing $(Q_i, Q_i, Q_j)$ when $(Q_i, Q_j, Q_k)$ are in the buffers. These computations can be *embedded* into some other rounds.

Algorithm 6 outlines one of the numerous ways to do so. The only change is to skip the first $p$ rounds in Algorithm 5 and do more computation at some specific rounds. $P(i)$ now starts with $(Q_{i-1}, Q_i, Q_{i+1})$ in the buffers instead of $(Q_i, Q_i, Q_i)$. The algorithm still alternates between interacting and shifting as before, but the number of shifts starts from $p-3$. Extra calculations are performed in the first $p-3$ rounds. Rounds 10-19 of Figure 4.3 also shows the complete system status of Algorithm 6. To see how this works, let us follow the interactions of $Q_0$. In the first round interactions are computed on these combinations of buffers: $(B_1, B_1, B_1), (B_1, B_1, B_2), (B_0, B_0, B_2)$, and $(B_0, B_1, B_1)$, in addition to the usual $(B_0, B_1, B_2)$, to get interactions of offset patterns 000, 001, 002, and 800. Offset patterns 700 to 300 are computed in the next p-4 rounds from the $(B_0, B_1, B_1)$ interactions.

Algorithm 6 has similar asymptotic costs as Algorithm 5 but sends $p$ fewer messages and $n$ fewer words in the exact costs. The load is still balanced since all processors are still computing the same distance patterns every round.

## Incorporating 1- and 2-body interactions

For simplicity, most previous 3-body algorithms compute 3-body potentials separately from those of 1- and 2-body potentials because the work is partitioned differently. With our approach, 1- and 2-body potentials can be computed together with 3-body the same way we *'embed'* the first $p$ rounds of Algorithm 5 into Algorithm 6 and the load will still be perfectly balanced without any extra effort.

**Algorithm 6** EMBEDDED ALL-UNIQUE-TRIPLETS

**Input:** $P$, a ring of $p$ processors.
**Input:** Set $Q$ of $n$ particles divided into $p$ equal subsets $Q_i$.
**Output:** The updated set $Q$.

```
1: for P(i) in parallel do
2:     Read Q_i into B_0 and copy them into B_1 and B_2.        // Has (Q_i, Q_i, Q_i).
3:     Shift B_0 by 1 and shift B_2 by -1.                      // Has (Q_{i-1}, Q_i, Q_{i+1}).
4:     b ← 0
5:     Calculate all (B_1, B_1, B_1), (B_1, B_1, B_2), and (B_0, B_0, B_2) interactions
6:     for each phase d ∈ {1, 2, ..., ⌊p/3⌋ − 1} do
7:         for p − 3d rounds do
8:             if d = 1 then
9:                 Calculate all (B_0, B_1, B_1) interactions.
10:            end if
11:            Update all 3-body interactions in B_0, B_1 and B_2.
12:            Shift B_b by 1.
13:        end for
14:        b ← (b + 1) mod 3                                    // Switch buffer to shift.
15:    end for
16:    if 3 | p then                                            // Special case.
17:        Calculate the ⌊i/(p/3)⌋^th third of the interactions of the three buffers.
18:    end if
19: end for
```

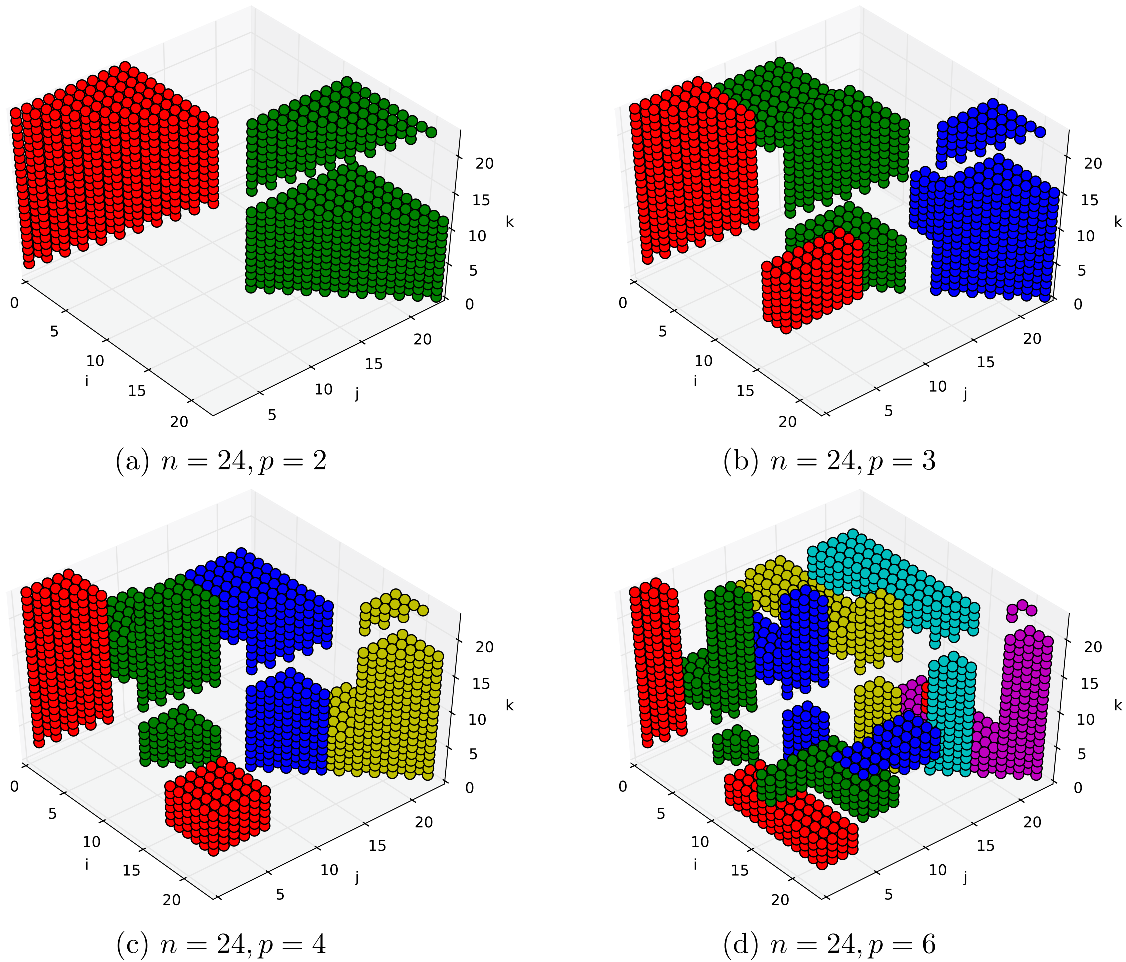
## Geometric Meaning

Since previous algorithms pick subsets of elements in the force cube to compute, it is interesting to see the subsets formed by our algorithm as well. For simplicity, let us focus on just Algorithm 5. (Algorithm 6 produces a slightly different subset because of the $(B_0, B_1, B_1)$ interactions.)

Let $C$ be the set of the elements in the force cube that is to be computed. In those related approaches [121, 122, 169], $C$ only depends on the number of particles, $n$, and is invariant to the number of processors, $p$. In contrast, our algorithm picks a new subset $C_p$ for every $p$. For example, Figure 4.5 shows these subsets for $n = 24$ particles and $p = 2, 3, 4,$ and 6. Red, green, blue, yellow, cyan, and magenta indicate the elements computed by $P(0), P(1), \ldots, P(5)$, respectively. All these subsets are equivalent to the big-tetrahedron-shaped subset a single processor would compute:

$$C_T = \{(i, j, k) \in \mathbb{Z} \mid 0 \leq i < j < k < n\}.$$

To provide the intuition for the shape of the subsets, we illustrate the case where $n = 24$ and $p = 6$ step-by-step from step 1 to 10 in Figure 4.6, 4.7, and 4.8. Subfigures in the left




(a) $n = 24, p = 2$ (b) $n = 24, p = 3$

(c) $n = 24, p = 4$ (d) $n = 24, p = 6$

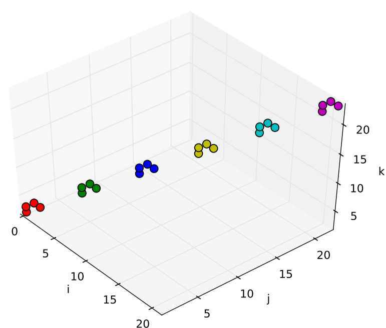
Figure 4.5: Subsets of computed elements for $n = 24$ and $p = 2, 3, 4, 6$ for Algorithm 5.

column show the actual force elements that are computed ($i$, $j$, and $k$ shows the particles from buffers $B_0, B_1$, and $B_2$, respectively.) while the ones in the right column show the equivalent force elements in the tetrahedron for $p = 1$, i.e., elements $(i, j, k)$ with $0 \leq i < j < k < n$.

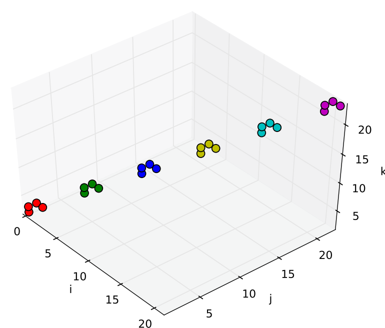
The force cube is partitioned into $p \times p \times p$ subcubes and each processor processes a subcube in each step. A processed subcube has three possible shapes depending on its coordinates: a tetrahedron at position $(i, i, i)$, a triangular prism at position $(i, i, j)$, and a cube at position $(i, j, k)$.

Shifting buffers $b_0, b_1$, and $b_2$ corresponds to moving periodically to process a new subcube along the $i$-, $j$-, and $k$-axis, respectively. Since the algorithm only shifts one buffer at a time before calculating interactions, the rays of processed subcubes extend in a one-dimensional manner, in parallel to the corresponding axis, and make a right-angle turn to move along another axis when another buffer is shifted.

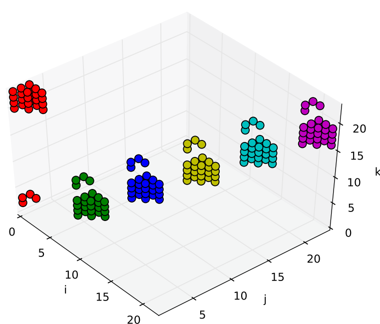
In this example, 6 rays of processed subcubes start at diagonal positions in the first step



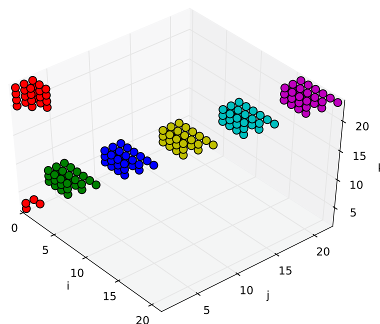

(a) round 1



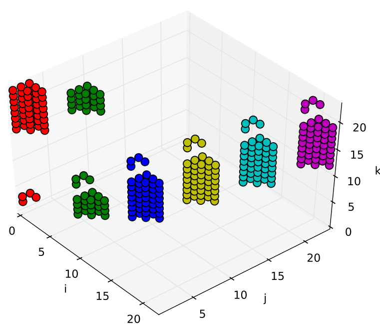

(b) round 1 (reflected)



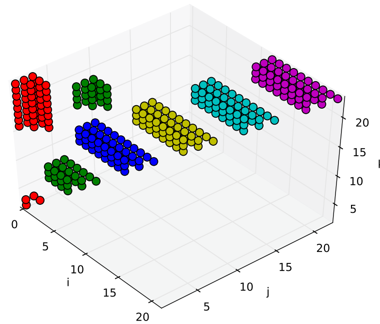

(c) round 2



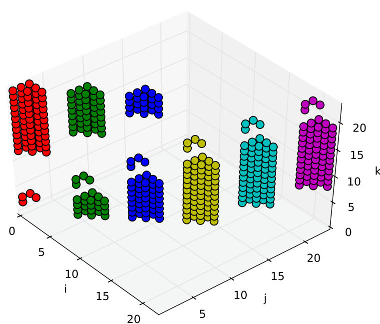

(d) round 2 (reflected)



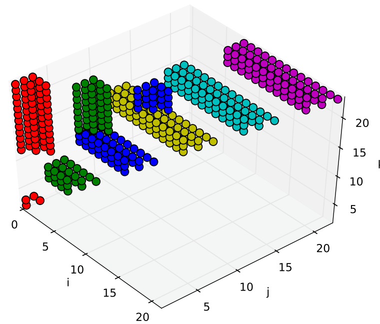

(e) round 3




(f) round 3 (reflected)

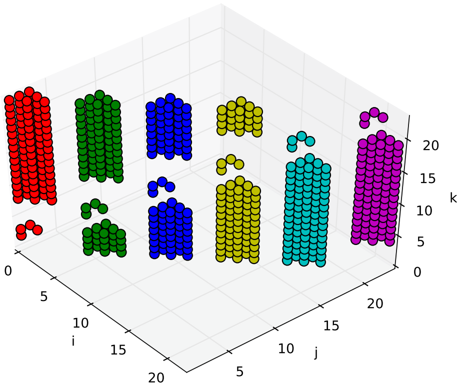



(g) round 4

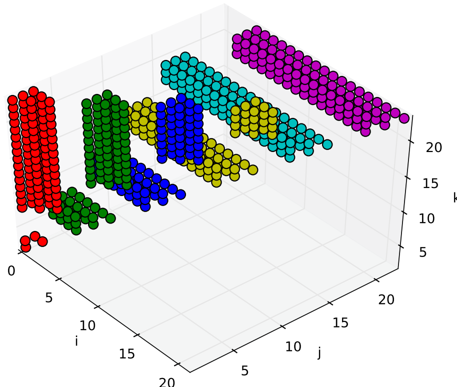



(h) round 4 (reflected)

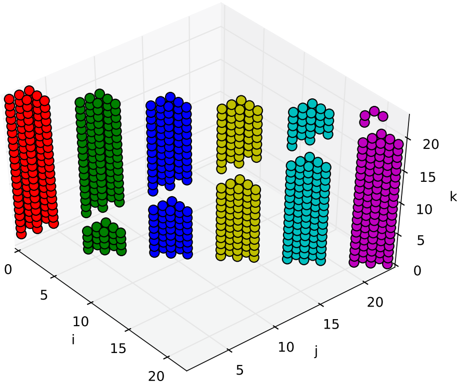
Figure 4.6: Computed elements for $n = 24$ and $p = 6$ for Algorithm 5 at rounds 1-4.



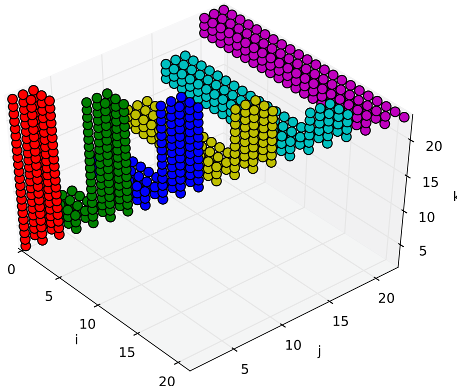

(a) round 5



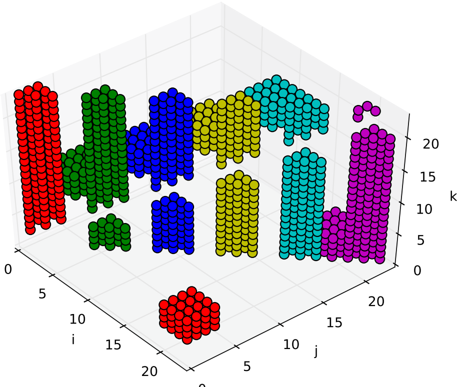

(b) round 5 (reflected)



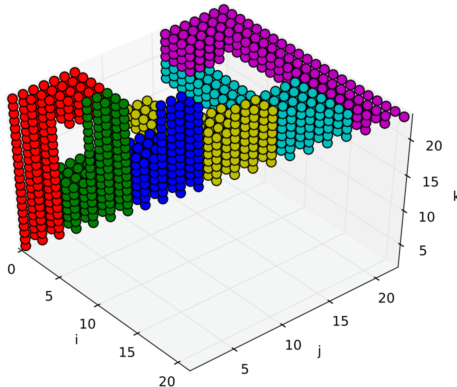

(c) round 6




(d) round 6 (reflected)

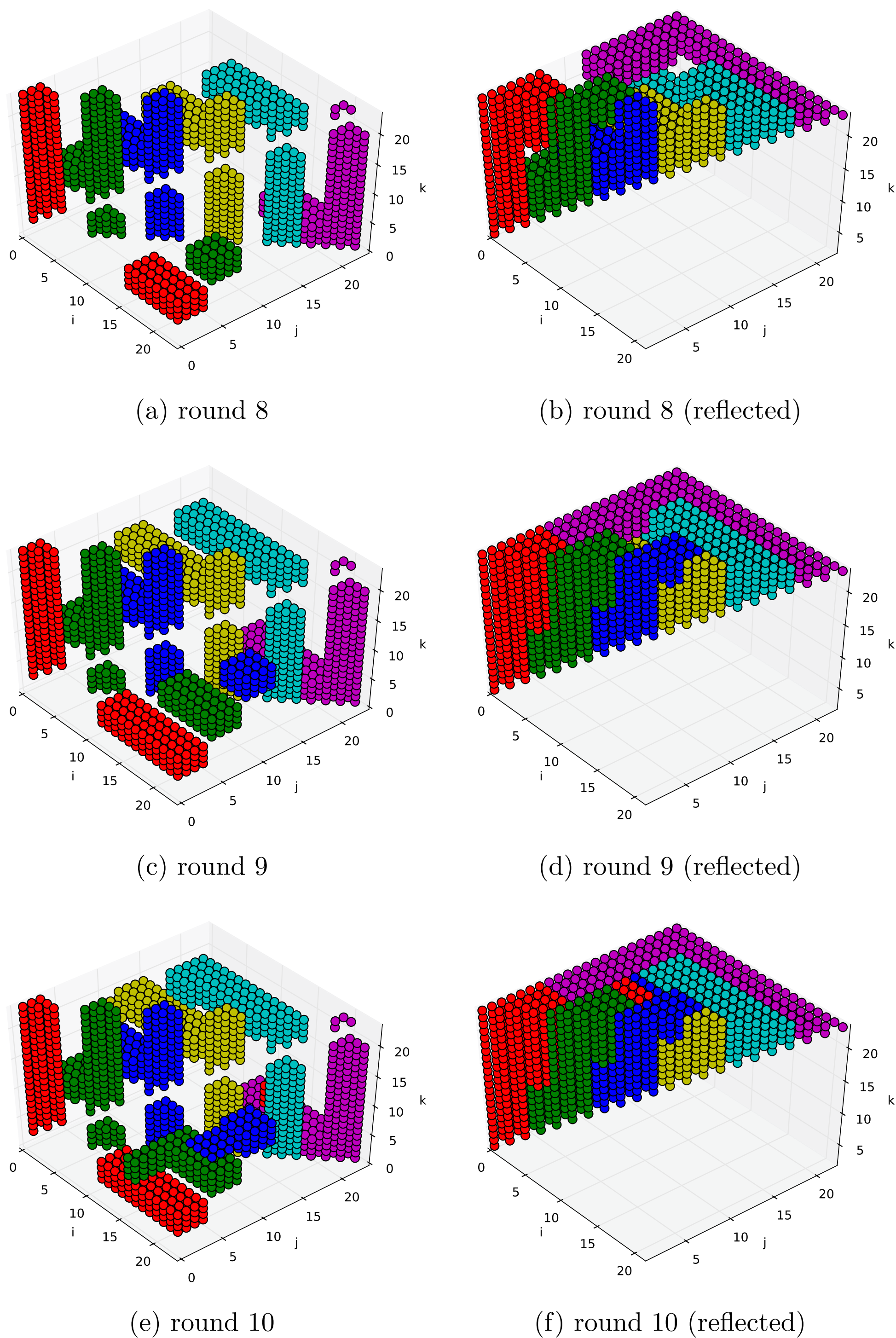



(e) round 7




(f) round 7 (reflected)

Figure 4.7: Computed elements for $n = 24$ and $p = 6$ for Algorithm 5 at rounds 5-7.



(a) round 8

(b) round 8 (reflected)

(c) round 9

(d) round 9 (reflected)

(e) round 10

(f) round 10 (reflected)

Figure 4.8: Computed elements for $n = 24$ and $p = 6$ for Algorithm 5 at rounds 8-10.

(Figure 4.6a), with the subcubes being filled in tetrahedron shape. Then they extend 5 subcubes along the $k$-axis because of 5 right-shifts on $b_2$ (Figure 4.6c to Figure 4.7c), turn and extend along the $i$-axis 3 times because of 3 right-shifts on $b_0$ (Figure 4.7e to Figure 4.8c). Finally, they make last turns and extend 1 subcube along $j$-axis because of one last right-shift on buffer $b_1$ (Figure 4.8e). Note that roughly a third of these last subcubes are filled due to the special case where $3 \mid p$.

## Communication-Avoiding All-Triplets Algorithm

This section applies the communication-avoiding (CA) technique to Algorithm 6. The main concept is that each processor stores more particles and cooperates on computing interactions with other processors who also have the same particle subsets.

Let us store $c$ times more particles and arrange $P$ logically into a $p/c$-by-$c$ 2D torus instead of a $p$-ring. We divide particles into $p/c$ equal subsets of $cn/p$ particles, $Q_0, Q_1, \ldots, Q_{p/c-1}$, and let processors $P(i,:)$ own $Q_i$ and cooperate on computing forces and updating them as a team.

The communication-avoiding algorithm behaves similarly to Algorithm 6 with $p/c$ processors; the only difference is that it divides all the rounds (lines in Figure 4.3) among processors on the same team, then performs a reduction on particles inside the teams first before sending particles back to owner processor teams.

Not all rounds have the same computation costs so round distribution has to be done wisely to avoid severe load imbalance. Currently, they are partitioned into consecutive rounds with accumulated load closest to the perfect load. Equation (4.13) predicts the cost to compute the $r^{\text{th}}$ round, where $m = cn/p$ represents the number of particles in each buffer. Again, it is trivial to support 1- and 2-body computation within the algorithm by updating the cost equation accordingly.

$$\text{cost}(r) = \begin{cases} m^3 + 3m\binom{m}{2} + \binom{m}{3} & \text{if } r = 0 \\ m^3 + m\binom{m}{2} & \text{if } 0 < r < p/c - 3 \\ m^3/3 & \text{if last round and } 3 \mid p \\ m^3 & \text{otherwise} \end{cases} \tag{4.13}$$

The communication-optimal 3-body algorithm is shown in Algorithm 7. The get_schedule function is introduced. It takes a processor's team number and layer ID (rank within its team) as inputs and indicates for each processor what rounds in the absolute round number of Algorithm 6 it should compute (variables *start* and *end*), what particle subsets should be in each buffer at the start (an array *srcs* containing three particle subset numbers), and what buffer should be shifted ($b$). The processor then handles the calculation of each round the same way as Algorithm 6 would. The function change_buffer takes in the absolute round number and the processor's layer ID (rank within team) and determine if it should switch the buffer being shifted.

**Algorithm 7** COMMUNICATION-AVOIDING ALL-TRIPLETS

**Input:** $P$, a 2D torus of $p/c$ teams by $\ell$ layers.
**Input:** Set $Q$ of $n$ particles divided into $p/c$ equal subsets $Q_i$.
**Output:** The updated set $Q$.

```
 1: for P(i, ℓ) in parallel do
 2:     Read Q_i into B_0 and copy them into B_1 and B_2.
 3:     (b, srcs, start, end) ← get_schedule(i, ℓ)
 4:     for buf ∈ {0, 1, 2} do
 5:         Shift B_buf by i − srcs_buf.     // Forms the starting (Q_srcs_0, Q_srcs_1, Q_srcs_2) triplet.
 6:     end for
 7:     for r ∈ [start, end) do
 8:         Calculate interactions between all three buffers.
 9:         Shift B_b by 1.
10:         if change_buffer(r) then
11:             b ← (b + 1) mod 3
12:         end if
13:     end for
14:     Send particles in each buffer back to the ℓ^th processor of the owner team.
15:     Add the forces on from the other 2 copies of Q_i in B_1 and B_2 to B_0.
16:     Do a sum-reduction of B_0 between P(i, :).
17: end for
```

Figure 4.3 can be used for illustration again; this time it is for 36 processors with replication factor $c = 4$. Processors are divided into 9 teams, resulting in a total of 10 rounds of interactions (rounds 10-19). According to equation (4.13), processors in rows 0, 1, 2, and 3 will compute rounds 10-11, 12-13, 14-15, and 16-19, respectively. Then processors on the same team do a column reduction to get total interactions computed by all processors in the team before sending particles back to their owners, which will proceed to update them as usual.

The overall communication is now carried in two dimensions. When processors separately calculate their shares of interactions in the team, they only need to shift horizontally in a ring as before. The additional direction comes from the column reduction at the end where team members need to communicate vertically. Regarding communication costs, there are $p/c$ columns so there will be $O(p^2/c^2)$ total rounds. Dividing the rounds to $c$ groups, each processor does approximately $O(p^2/c^3)$ rounds. A message is sent per round, so $O(p^2/c^3)$ messages are sent during shifting phases. Only $O(\log c)$ messages are required for column reduction so we can consider the total number of messages sent to be just $O(p^2/c^3)$. Each message is of size $cn/p$, therefore the bandwidth used is $O(p^2/c^3)\cdot cn/p = O(np/c^2)$. Since the costs match the lower bounds in (4.2), we conclude the communication-avoiding algorithm is communication optimal.

There is a constraint on the replication factor $c$. If there are more processors per team

than the number of rounds to compute, then some processor rows will be idle and the computation efficiency will decrease. Hence, we want

$$c \le \binom{p/c}{3} \div (p/c)$$
$$6c^3 \le (p-c)(p-2c). \quad (4.14)$$

Any $c$ that satisfies inequality (4.14) guarantees that all processor rows are utilized. Explicit solution of $c$ is omitted due to its length and complexity. Simplifying inequality (4.14) to $c^3 \le p^2$, we get the asymptotic upper bound of $c$,

$$c = O(p^{2/3}), \quad (4.15)$$

which is consistent with the maximum effective $c_M$ in Section 4.2.

### Extension to k-body interactions

The same offset pattern approach can be used to calculate all-k-tuple interactions and will preserve load balance. However, finding a way to generate all unique offset patterns is more complicated. We have not derived them because we do not have a use case for more than all-triplets interactions yet. Below are some of the challenges

- There are more offset patterns that create redundancy within themselves, which are tedious to handle. For example, in 4-body problem at $p = 8$ and $c = 1$, offset patterns $\square\square - - \square\square - -$, $\square - \square - \square - \square -$, and many others create redundancy because 4 is not a prime number.

- It is nontrivial to find an efficient shifting pattern where only one message is sent per round (only one buffer is shifted).

## 4.4 Performance Results

As a demonstration, we implemented Algorithm 7 with C and MPI. A particle is of size 80 bytes, consisting of one integer for particle ID, three double-precision variables for the $x$, $y$, and $z$ coordinates, three double-precision variables for $x$, $y$, and $z$ velocities, and three double-precision variables for $x$, $y$, and $z$ accelerations. The Axilrod-Teller potential [13] is used for 3-body interactions. The Newton's third law of motion still applies. The 3-body force symmetry properties are as follows: the force on particle $i$ from particle $j$ alone in a triplet $(i, j, k)$ is equal to the negative force on particle $j$ from particle $i$ alone, $f_{i(j)k} = -f_{j(i)k}$. There two other equalities: $f_{ij(k)} = -f_{k(i)j}$ and $f_{ji(k)} = -f_{ki(j)}$. The total force on particle $i$ from both particle $j$ and $k$, denoted by $f_{i(jk)}$ is simply the sum of $f_{i(j)k}$ and $f_{ij(k)}$. The sum of the total force on particles $i$ and $j$ is equal to the negative total force on particle $k$, $f_{i(jk)} + f_{j(ik)} = -f_{k(ij)}$. Our implementation computes each total force $f_{i(jk)}$, $f_{j(ik)}$ and $f_{k(ij)}$

directly all at once from many shared terms within a triplet, for examples, square distances, cube distances, and distances to the $5^{\text{th}}$ between all pairs of particles. We did not overlap communication since we wanted explicit communication time to fully measure the effect of the algorithm and because it is also a good indicator of energy usage. The correctness verification was done by comparing the algorithm's outputs to those of a sequential program at small problem sizes and confirming that the difference was no greater than a threshold relative to $n$ and $p$. (The difference is caused by floating-point round-off errors due to different summation orders and could be prevented by using reproducible summation [58].) We benchmarked the program on two platforms with different network topology, Mira at Argonne Leadership Computing Facility (ALCF) and Edison at National Energy Research Scientific Computing Center (NERSC).

**Mira** is a 10 PFLOPS IBM Blue Gene/Q supercomputer with a 5D torus network and topology-aware task mapping. There are 49,152 compute nodes, each has a 16-core PowerPC A2 1.6GHz processor with 4 hardware threads and 16 GB of memory.

**Edison** is a 2.57 PFLOPS Cray XC30 supercomputer consisting of 5,576 compute nodes. Each node is equipped with 2 sockets of 12-core Intel Ivy Bridge processor at 2.4GHz and 64 GB memory. The machine has Cray Aries interconnection with Dragonfly topology.

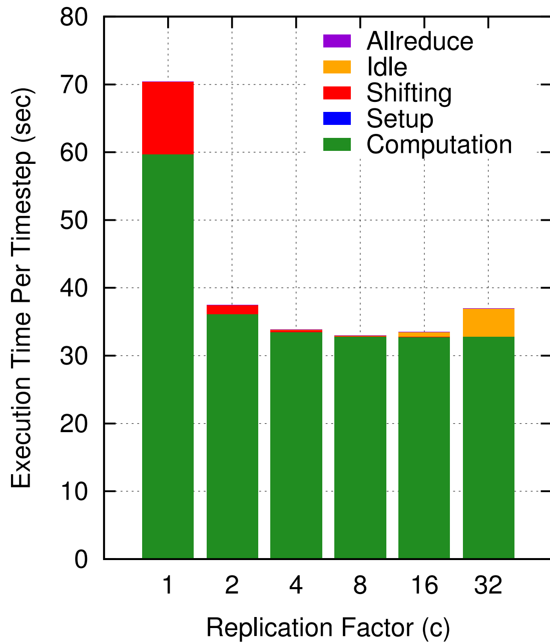
While we used flat MPI with one MPI process per core on Cray XC30, we implemented hybrid MPI/OpenMP version for Blue Gene/Q to fully utilize its 4 hardware threads per core. Using one MPI process per core, we ran 1, 2, and 4 OpenMP threads per MPI process and selected the best results.

## Effects of replication

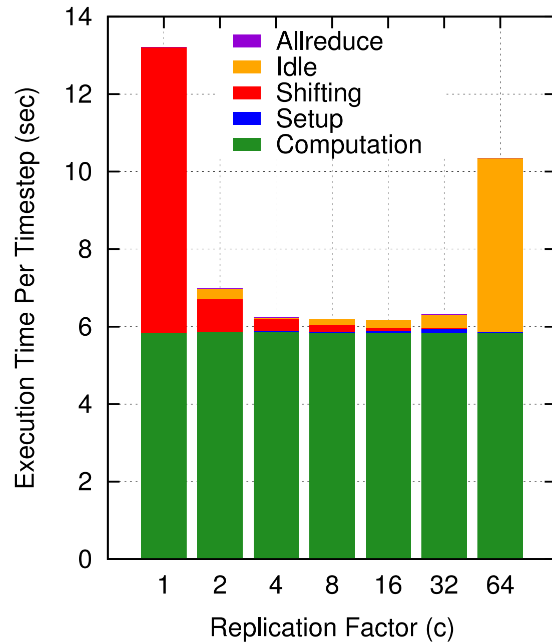
Figures 4.9 shows time breakdowns per timestep of the program as we vary the replication factor $c$. The green, blue, red, yellow, and purple bars are the computation, setup, shifting, idle, and allreduce times, respectively. Setup time is the time to load required particle subsets to buffers at the beginning of the timestep and the time to send particles back to their owners at the end of the timestep combined. Idle time is the average time each processor has to wait for its teammates to reach the reduction point. Allreduce time is the reduction time among column teams.

Figures 4.9a and 4.9b are small-scale results – 8K particles on 1K cores on Blue Gene/Q and 6K particles on 1.5K cores on Cray XC30. Figure 4.9c and 4.9d are large-scale results – 16K particles on 8K cores on Blue Gene/Q and the extreme case, 24K particles on 24K cores on Cray XC30. All four graphs demonstrate a same decreasing trend in shifting time, i.e., between 4 to 8 times reduction as $c$ doubles. This is consistent with the bounds in equation (4.2) which says the algorithm can save factors of $c^3$ in messages and $c^2$ in bandwidth.

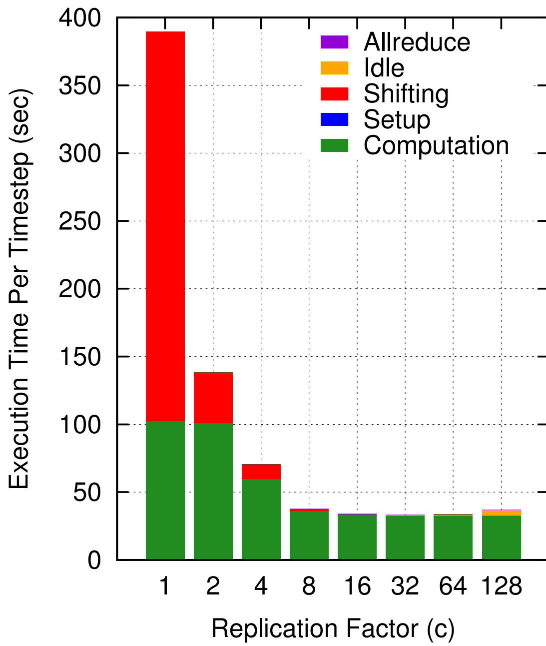
Up to 99.98% reduction in communication time, idle time included, is observed in the experiment. Maximum overall speedup occurs at the extreme scale at one particle per core, 22.13× on 16K cores on Blue Gene/Q (breakdown graph not shown) and 41.85× on 24K cores on Cray XC30.



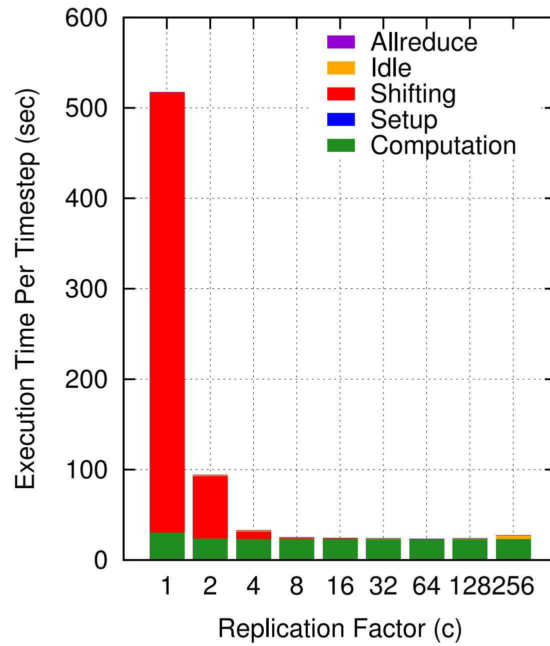

(a) Blue Gene/Q, 1,024 cores, 8,192 particles. (8 particles per core)




(b) Cray XC30, 1,536 cores, 6,144 particles. (4 particles per core)

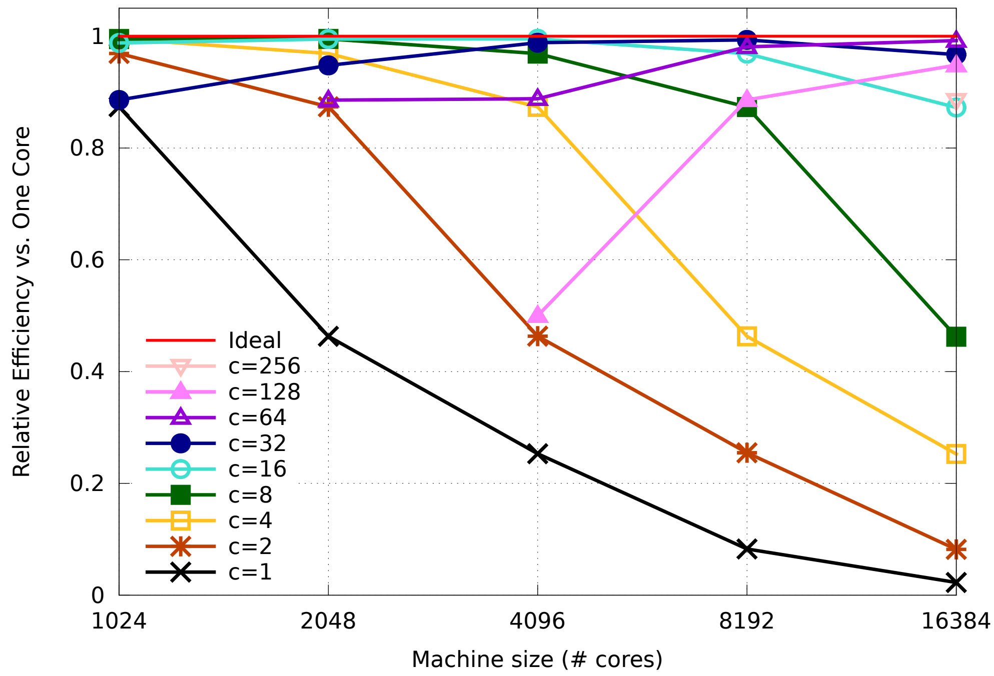



(c) Blue Gene/Q, 8,192 cores, 16,384 particles. (2 particles per core)

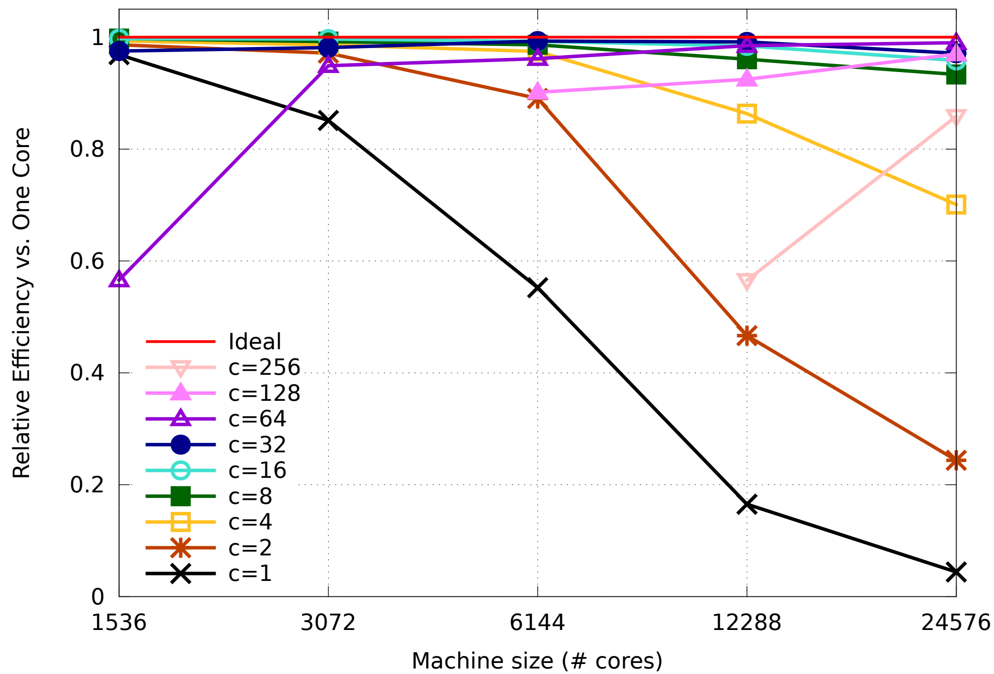



(d) Cray XC30, 24,576 cores, 24,576 particles. (1 particle per core)

Figure 4.9: Time breakdown for each replication factor in small- and large-scale experiments on Blue Gene/Q and Cray XC30.




(a) Blue Gene/Q, 16,384 particles




(b) Cray XC30, 24,576 particles.

Figure 4.10: Strong scaling on Blue Gene/Q and Cray XC30. Larger $c$'s perform better as the number of cores increases, vice versa for smaller $c$'s. Some large $c$'s are available only with sufficiently many cores.

Computation times are mostly equal as expected. For Blue Gene/Q, the computation time where $c \leq 4$ is significantly larger than others because each thread has too little work to do. As $c$ increases, there are more interactions to compute per thread so the computation times are equal from $c > 4$. This shows that replication also helps maintain computation efficiency.

Idle time indicates load imbalance. It increases with $c$ because storing more particles means more interactions per round and also fewer rounds to distribute among the processor rows (a.k.a. *team members*). Load balance is harder to maintain at coarser grain. However, this imbalance can be predicted and thus avoided since the work partitioning is static and does not depend on input data.

### Scalability

Figure 4.10 shows strong scaling on Blue Gene/Q and Cray XC30, with 16K and 24K particles, respectively. The $y$-axis is relative efficiency compared to one node, which is estimated from running the program on one node for 2,048 and 3,072 particles on Blue Gene/Q and Cray XC30, respectively. The red line at 1 indicates ideal efficiency and other lines are efficiencies for each replication factor $c$ from 1 to 256. The benchmark achieved perfect strong scaling at 99% efficiency for all machine sizes on both machines with the best $c$.

In general, larger $c$'s perform better than smaller $c$'s with more cores. Some replication factors are only available with sufficiently many cores since more memory is available with more cores.

## 4.5 Extension for Cutoff Distance

This section extends the all-triplets algorithm to support a cutoff distance. Unlike pairwise interactions in which there is only one obvious way to apply the cutoff distance $\delta$, it is more complicated in the 3-body case. There have been various ways of applying cutoff distance to 3-body interactions, from triplets in which at least a pair of particles is less than $\delta$ apart [50, 113], triplets where all particles are less than $\delta$ apart [130], to triplets where the sum of all distances from the center of mass is less than $\delta$ [144], etc. We opted to follow the second approach where all particles have to be less than $\delta$ from each other because the first approach does not preserve Newton's third law [130] and because of its simplicity.

The limited interaction distance helps simplify the process of forming unique $k$-tuples. We first discuss our $k$-body cutoff algorithm for 1-dimensional simulation space, then mention how to extend it to support higher dimensional space. We use arrange $P$ into a 1D torus and use domain decomposition, i.e., map $P$ to the simulation space, and let them own particles in their range. We assume that the particles are uniformly distributed and the cutoff distance $\delta$ is large enough to span $b \geq 1$ processor boxes, but smaller than half the simulation space ($b < p/2$). Define the cutoff window $w_i$ as a $2b+1$-wide window centered at $P(i)$,

$$w_i = \{i-b, \dots, i-1, i, i+1, \dots, i+b\} \bmod p.$$

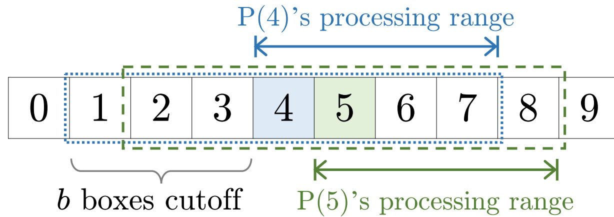



Figure 4.11: 10 processors in a 1D-space simulation. The cutoff distance spans $b = 3$ boxes and each processor process $k$-tuples by moving only *to the right* of its box.

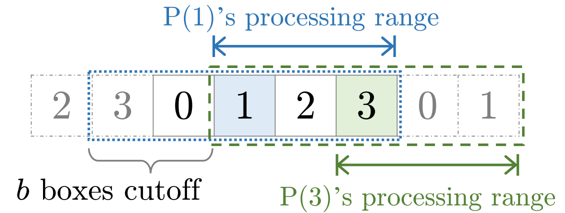



Figure 4.12: Redundancy happens if the cutoff distance is more than half the simulation space length, i.e., $(1, 3)$ for 2-body, $(1, 3, 3)$ for 3-body, and so on.

For example, Figure 4.11 shows $p = 10$ processors with cutoff distance $b = 3$ boxes. We draw two cutoff windows, $w_4$ in blue and $w_5$ in green. Let $\ominus$ be the periodic difference operator,

$$j \ominus i = \begin{cases} j - i & \text{if } |j - i| \leq p/2 \\ \text{sign}(i - j)(p - |j - i|) & \text{otherwise} \end{cases}$$

We use $\leq_i$ to compare processor ranks from $P(i)$'s point of view, $j \leq_i k$ if $j \ominus i \leq k \ominus i$.

Our algorithm makes each $P(i)$ process $k$-tuples of particle subsets $(Q_i, Q_{i_2}, \ldots, Q_{i_k})$ where $i \leq_i i_2 \leq_i i_3 \leq_i \ldots \leq_i i_k \leq_i (i + b) \bmod p$, i.e., starting from itself and moving only *to the right*, up to the *right* border of the cutoff window, when picking other $Q$s. The key to avoiding redundancy is to make sure that any processors within $b$ boxes from each other order all processors in their intersected cutoff windows the same way. For example, in Figure 4.12 there are 4 processors with cutoff distance spanning 2 boxes, long enough for $P(3)$ to be *on the right* of $P(1)$ $(1 \leq_1 3)$ and $P(1)$ to also be *on the right* of $P(3)$ $(3 \leq_3 1)$. In this case, $P(1)$ and $P(3)$ will produce redundant $k$-tuples. We will show that, by enforcing the cutoff distance to be less than half the simulation space length, this cannot happen.

**Lemma 4.8.** *Given $b < p/2$, for any $0 \leq i < p$, if $x \leq_i y$, then $x \leq_j y$ for all $x, y, j \in w_i$.*

*Proof.* Assume there exists some $x, y, j \in w_i$ where $x \leq_i y$ but $x >_j y$. Then the cutoff range must wrap around like in Figure 4.12. The farthest distance between $x$ and $y$ when $x \leq_i y$ is $b$ where $x = i$ and $y = (i + b) \bmod p$. Likewise, the farthest distance between $y$ and $x$ when $y \leq_j x$ is $b$ where $y = j$ and $x = (j + b) \bmod p$. This means if we keep walking to the right from $x$, passing $y$, and back to $x$ again, the distance cannot have been more than $b + b = 2b$. Since this distance is also $p$ (we just walked through the whole simulation space), we have $p \leq 2b$, which contradicts the assumption. □

Next, we prove that our algorithm produces all unique $k$-tuples with no repeats.

**Lemma 4.9.** *Any $k$-tuple of particle subsets $(Q_{i_1}, Q_{i_2}, \ldots, Q_{i_k}), 0 \leq i_1, i_2, \ldots i_k < p$, that all $k$ indices are within $b$ boxes from each other is processed by exactly one processor.*

*Proof.* Let $j_1, j_2, \ldots, j_k$ be the permutation of the indices such that $i_{j_1} \leq_{i_1} i_{j_2} \leq_{i_1} \ldots \leq_{i_1} i_{j_k}$, then, by Lemma 4.8, $i_{j_1} \leq_{i_m} i_{j_2} \leq_{i_m} \ldots \leq_{i_m} i_{j_k}$ for all $1 \leq m \leq k$. Therefore, $P(i_{j_1})$ is the only processor to compute this tuple, since $i_{j_1}$ appears to be on the left side of the cutoff window to all other $P(i_m), m \neq j_1$. Note that it is not possible for $P(i_{j_1})$ to miss this tuple since $i_{j_1} \leq_{i_{j_1}} i_{j_2} \leq_{i_{j_1}} \ldots \leq_{i_{j_1}} i_{j_k} \leq_{i_{j_1}} (i_{j_1} + b) \bmod p$. □

To simulate $k$-nested loops, each processor has $k$ buffers, $B_0, B_1, \ldots, B_{k-1}$ and alternates between shifting and computing interactions as shown in Algorithm 8. We shift each buffer by -1 to mimic the serial schedule. Consider a 1-dimensional problem space with $p \geq 7$ and $b = 3$ boxes, $P(0)$ will process particle subset triplets in this order: 000, 001, 002, 003, 011, 012, 013, 022, 023, 033. When a non-innermost loop changes, all loops nested in it also need to change its buffer. For example, from 003 to 011, or from 013 to 022. This requires more than one shifts per round. To retain one shift per round, we rely on the fact the all the loops nested inside need the same particle subset as the changed loop, so instead of shifting all of them, we just copy the particle subset over. Back to the 003-to-011 example, instead of shifting buffer $B_1$ by -1 and $B_2$ by -2, we just shift $B_1$ by -1 and copy the particle subset in $B_1$ to $B_2$.

## $d$-dimensional Space

Regardless of the dimensionality of the problem space, every processor always computes the same tuple positions relative to their cutoff window, for example, if $P(i)$ is computing $(Q_{i+x}, Q_{i+y}, Q_{i+z}) \bmod p$ then any other $P(j)$ is computing $(Q_{j+x}, Q_{j+y}, Q_{j+z}) \bmod p$. So everything from the 1D case extends naturally here. In fact, we can just map $b^d$ voxels of the $d$-dimension cutoff volume to 1D and use the schedule of the 1D version.

## Computation Optimality

Again, we are looking for two kinds of optimality – no redundant work and load balance. Lemma 4.9 indicates that no $k$-tuple is computed twice and thus proves the first property. The load balance depends on particle distribution. If particles are uniformly distributed, all processors will compute an equal number of interactions every round and the load is balanced. Otherwise, there will be load imbalance.

## Communication Optimality

Here we derive the communication lower bounds for the uniformly-distributed cutoff case. For a $d$-dimensional problem space, let $f$ be the ratio of the cutoff window volume to the problem space volume,

$$f = \frac{(2b_c + 1)^d}{p}.$$

**Algorithm 8** PARALLEL K-BODY CUTOFF ALGORITHM

**Input:** $P$, a ring of $p$ processors.
**Input:** Set $Q$ of $n$ particles divided and binned into $p$ equal subsets $Q_i$.
**Input:** $b$, the number of processor boxes the cutoff distance spans.
**Output:** The updated set $Q$.

```
1: for P(i) in parallel do
2:     Copy Q_i to B_0, ..., B_{k-1}.
3:     for i_2 = 0 : b do
4:         for i_3 = i_2 : b do
5:             ...
6:             for i_{k-1} = i_{k-2} : b do
7:                 for i_k = i_{k-1} : b do
8:                     Compute all interactions between particles in buffers B_0, ..., B_{k-1}.
9:                     if i_k < b then      // Do not shift if the outer loop is going to shift.
10:                        Shift B_{k-1} by -1.
11:                    end if
12:                end for
13:                if i_{k-1} < b then
14:                    Shift B_{k-2} by -1.
15:                end if
16:                Copy B_{k-2} to B_{k-1}.
17:            end for
18:            ...
19:            if i_3 < b then
20:                Shift B_3 by -1.
21:            end if
22:            Copy B_3 to B_4, B_5, ..., B_{k-1}.
23:        end for
24:        Shift B_2 by -1.
25:        Copy B_2 to B_3, B_4, ..., B_{k-1}.
26:    end for
27: end for
28: Update my particles.
```

This means a particle has about $fn$ particles in its cutoff volume and needs $O((fn)^{k-1})$ interactions. The total work is $O\left(n \cdot (fn)^{k-1}\right)$ and thus each processor has work $Z = O(f^{k-1}n^k/p)$. Let us continue to write $M$, the memory available to a per processor, as $c_M n/p$. The maximum useful work a processor can do with $M$ particles in memory is $O(M^k)$, therefore, $F = O(c_M^k n^k/p^k)$. Substituting this into equation (2.2), we get the general

communication lower bounds for $k$-body problem,

$$S = \Omega\left(\frac{f^{k-1}p^{k-1}}{c_M^k}\right), \qquad W = \Omega\left(\frac{nf^{k-1}p^{k-2}}{c_M^{k-1}}\right). \tag{4.16}$$

For example, let $f = 1.0$, $k = 3$ and we get the same lower bounds as in equation (4.2). The case where $f = 1.0$, $k = 2$ also matches the 2-body communication lower bounds previously derived in Chapter 3.

For the memory-independent lower bounds, let $V$ be the set of interactions the processor with the largest load has to compute. $|V|$ cannot be less than the total amount of work divided equally among processors,

$$|V| \geq \frac{f^{k-1}n^k}{p}. \tag{4.17}$$

We relate $|V|$ to the total input size $\sum_{i=0}^{k-1}|V_i|$ necessary to compute $V$,

$$\begin{aligned} |V| &\leq |V_0||V_1|\ldots|V_{k-1}| \\ &\leq \frac{1}{k!}\left(\sum_{i=0}^{k-1}|V_i|\right)^k. \end{aligned} \tag{4.18}$$

Putting Equations (4.17) and (4.18) together, we have,

$$\begin{aligned} \frac{1}{k!}\left(\sum_{i=0}^{k-1}|V_i|\right)^k &\geq \frac{f^{k-1}n^k}{p} \\ \sum_{i=0}^{k-1}|V_i| &\geq \sqrt[k]{k!}\, nf^{\frac{k-1}{k}}p^{-\frac{1}{k}}. \end{aligned}$$

Since $V_0, V_1, \ldots, V_{k-1}$ can all be the same, the processor must hold at least $\sqrt[k]{k!}\, nf^{\frac{k-1}{k}}p^{-\frac{1}{k}}/k$ particles. Assuming every processor starts with just $n/p$ particles in their memory, they must communicate at least $\sqrt[k]{k!}\, nf^{\frac{k-1}{k}}p^{-\frac{1}{k}}/k - n/p$ words, using at least 1 messages. Assuming $\sqrt[k]{k!}\, nf^{\frac{k-1}{k}}p^{-\frac{1}{k}}/k > n/p$, the memory-independent lower bounds are,

$$S = O(1), \qquad W = O\left(nf^{\frac{k-1}{k}}p^{-\frac{1}{k}}\right), \tag{4.19}$$

equivalent to the memory-dependent lower bounds when $c_M = f^{\frac{k-1}{k}}p^{\frac{k-1}{k}}$. Any $c_M > f^{\frac{k-1}{k}}p^{\frac{k-1}{k}}$ gets the same lower bounds in Equation (4.19).

Now we analyze the communication costs of Algorithm 8. Since particles are distributed evenly and processors are responsible of equal domain size, there are roughly $fp$ processor

cells within a processor's cutoff volume and $O(fp)$ rounds are required to go through all of them. With $k$-way interaction, a processor needs to loop through all cells in range for $k-1$ levels, totaling of $O\left((fp)^{k-1}\right)$ rounds. It sends a message per round. The program stores $n/p$ particles in memory. Therefore, the total communication costs are,

$$S = O\left(kf^{k-1}p^{k-1}\right), \qquad W = O\left(knf^{k-1}p^{k-2}\right),$$

which are optimal if the memory does not have enough space to replicate ($c_M = 1$).

### Communication-Avoiding Algorithm

A communication-avoiding algorithm can be derived the same way as it was for the all-triplets section. In brief, we divide processors into $p/c$ teams with $c$ team members each. Each team owns $cn/p$ particles. All processors in a team cooperate to compute interactions for the particle subset their team owns. Work should be partitioned in a way that each processor calculates close to $1/c$ of all interactions of the subset. Finally, processors participate in a team sum-reduce to merge all interactions together, then update the particles. Due to the lack of space, we will only prove its communication optimality.

Assume once again that particles are uniformly distributed and let $f$ be the ratio of the cutoff volume to the problem space volume. There are $p/c$ teams, each has to interact with approximately $O(fp/c)$ particle parts from other teams. $O\left((fp/c)^{k-1}\right)$ rounds of interactions are required in $k$-way interactions. If the work is partitioned perfectly, a processor has to compute $O\left((fp/c)^{k-1}/c\right)$ rounds. Assuming optimal shifting schedule (1 message per round) is used, the total communication costs are

$$S = O\left(\frac{f^{k-1}p^{k-1}}{c^k}\right), \qquad W = O\left(\frac{nf^{k-1}p^{k-2}}{c^{k-1}}\right),$$

which are communication-optimal when we choose $c = c_M$.

## 4.6 Conclusions

This chapter presented a direct long-range 3-body algorithm and proved that it is both computation and communication optimal. We also provided a communication-avoiding version that, by making $c$ replicas, decreases the total number of messages and bandwidth usage by $c^3$ and $c^2$, respectively.

The communication-avoiding algorithm introduces some load imbalance, but it is predictable based on a given replication factor, $c$, and grows with increasing values of $c$. Thus, there is a tradeoff between reducing shifting time through bandwidth and latency reductions and increasing idle time. Since we know the amount of load imbalance for each $c$ ahead of time and the increase in reduction time is insignificant compared to shifting time, we suggest picking large $c$'s with reasonably small load imbalance.

Large-scale experimental results on up to 16K cores on a Blue Gene/Q and 24K cores on a Cray XC30 were consistent with the communication costs predicted for the communication-avoiding algorithm and also exhibited strong scalability. The algorithm exhibited up to 99.98% reduction in communication time and 41.85× speedup relative to the version without replication, enabling strong scaling up to 99% efficiency.

Finally, we presented a generalized algorithm that supports $k$-body computations with a large cutoff distance that limits interactions to 1/3 of the total particles. We derived the communication lower bounds for this $k$-body algorithm and showed that it is also both computation and communication optimal.

We believe that our algorithmic framework, which represents a class of algorithms for various values of $k$ and varying amounts of memory for replication is flexible enough to be useful in multiple application settings. The algorithms were provably optimal in communication and computation, and had bounded load imbalance. Overall, this work has shown the importance of both communication avoidance and computation avoidance for scalable $k$-body algorithms in both a theoretical and experimental setup.

# Chapter 5

# Sparse-Dense Matrix-Matrix Multiplication

Based on prior work on avoiding communication in matrix multiplication algorithms, we make several contributions in this chapter. We first provide communication lower bounds for parallel sparse-dense matrix-matrix multiplication (SpDM$^3$) algorithms. We then introduce efficient parallel algorithms, together with a rigorous analysis of their communication costs. We also provide performance results on up to ten thousands of cores using sparse matrices with both uniform and skewed nonzero distributions. Finally, we analyze the SpDM$^3$ problem within an iterative algorithm.

## 5.1 Background and Previous Work

Computing the product of a sparse matrix with a dense matrix is an understudied primitive in numerical linear algebra, but is important in several higher level algorithms.. In this chapter, we focus on the case where both matrices have fairly large dimensions, in contrast to the well-studied case in iterative methods where the dense matrix can be viewed as a small set of column vectors, typically fewer than 100 columns. [5].

Sparse-dense matrix-matrix multiplication, or SpDM$^3$ in short, has applications in a diverse set of application domains involving both scientific simulations and data analysis problems. Examples include the all-pairs shortest-paths problem [173] in graph analytics, non-negative matrix factorization [105] for dimensionality reduction, a novel formulation of the restriction operation [136] in Algebraic Multigrid, quantum Monte Carlo simulations for large chemical systems [159], interior-point methods for semidefinite programming [76], the siting problem in terrain modeling [146], block eigenvalue problems where the number of

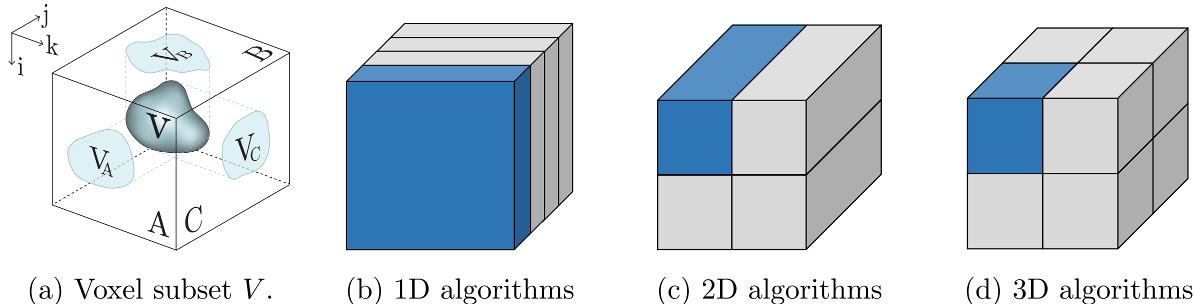
This chapter is based on joint work previously published in "Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication"[111].

eigenvalues to be estimated is large (more than 1,000) [176, 68], attribute inference attacks in computer security [97, 84], and deep convolutional networks [106] in machine learning.

Another application in statistics and machine learning is sparse inverse covariance selection (ICS) and related problems [62, 182, 75, 147, 103, 174]. ICS estimation is used for data analysis to identify the most important probabilistic relationships between various features of a given population. Its computational burden can be cubic in the number of dimensions [75, 147, 103] per iteration and, hence, easily becomes intractable as dimensionality increases beyond a few thousand. To improve scalability, a divide-and-conquer approach has been developed for a shared memory architecture [93]; however, we know of none that take advantage of parallel distributed memory computing environments. More recently, an algorithm based on sparse linear algebra was developed [142] for computing the CONCORD estimator [103]. The running time of this ICS estimation algorithm, CONCORD-ISTA, is dominated by the solution of two SpDM$^3$ problems at every iteration. Hence, a fast parallel SpDM$^3$ would significantly increase CONCORD-ISTA's scalability and improve its running time. CONCORD-ISTA and additional examples of computational algorithms that can benefit from SpDM$^3$ are further discussed in Section 5.5.

There has been relatively little work on the SpDM$^3$ problem. Bader and Heinecke [18] presented cache-oblivious algorithms based on space-filling curves, together with their high-performance shared memory implementations. Greiner and Jacob [88] presented I/O-efficient serial algorithms and related lower bounds. Ortega et al. [143] provided an efficient GPU implementation. In terms of multi-node parallelism, the literature is even sparser. Pietracaprina et al. [149] gave lower bounds on the number of rounds it takes to compute the sparse matrix product in MapReduce. The only multi-node implementation we are aware of is published in 2016 [2], which was after we had published our work. The scope is also different. They focus on structured matrices using hypergraph partitioning and only test scaling up to 2,048 cores. Our approach focuses on general matrices, including those arising in machine learning problems that often lack any clear structure, and replication, demonstrating scaling results up to over ten thousand cores.

Communication-avoiding algorithms aim to reformulate linear algebra operations to minimize the communication costs [23, 59]. In the case of dense-dense matrix multiplication, all 2D, 2.5D, and 3D algorithms are optimal, given two different assumptions about available memory [96]. 2.5D and 3D algorithms, however, further minimize the cost of communication relative to 2D algorithms, at the expense of more memory usage [167]. Sparse-sparse matrix multiplication is more complicated due to different sparsity patterns. Lower bounds are only known for Erdős-Rényi matrices, for which optimal 2.5D/3D algorithms have also been proposed [25]. The efficient implementation of the 2.5D/3D sparse-sparse matrix multiplication algorithm on distributed-memory architectures has been done only recently [14].




(a) Voxel subset $V$. (b) 1D algorithms (c) 2D algorithms (d) 3D algorithms

Figure 5.1: The matrix multiplication computational cuboid. We call algorithms $k$D if they split $k$ dimensions of the cube.

## 5.2 Communication Lower Bounds

We compute the parallel matrix-matrix multiplication $C = A \times B$. $A$ is a sparse, $m \times q$ matrix, $B$ is a dense, $q \times n$ matrix, and $C$ is an $m \times n$ matrix which is usually dense, depending on the sparsity pattern of $A$ and the size of $q$. For theoretical analysis and lower bounds, we assume that the nonzeroes in $A$ are uniformly distributed, as in the Erdős-Rényi model [70], and that there are $d$ nonzeroes per row on average. For experimental analysis, we also use more realistic sparsity patterns. Note that all analyses can be easily extended to the reverse case where $A$ is dense and $B$ is sparse.

The iteration space for the problem can be seen as an $m \times n \times q$ cuboid where each voxel $(i, j, k)$ represents the computation $c_{ij} \mathrel{+}= a_{ik} \cdot b_{kj}$. All $p$ processors partition the cuboid and compute the voxels in their subsets. Figure 5.1a shows the computational cuboid, an example voxel subset $V$, and its projections onto the $A$, $B$, and $C$ planes.

For simplicity of presentation, we assume that all matrices are square with length $n$ in this section, although our analysis and implementations will handle the more general case. Following our earlier notation, let $M$ be the size in words of the fast memory each of our $p$ processing elements has. Let $F$ be the total number of flops to multiply $A$ and $B$, $F = O(dn^2)$. The general lower bounds for communication along the critical path from [23],

$$S = \Omega\left(\frac{F}{p\sqrt{M^3}}\right), \quad W = \Omega\left(\frac{F}{p\sqrt{M}}\right), \tag{5.1}$$

trivially apply here. To relate $M$ to our problem parameters, we assume that $M$ can fit at most $c$ copies of all three matrices, i.e., $M = O(cn^2/p)$. Substituting for $F$ and $M$ into Equation (5.1) gives us the lower bounds,

$$S_{\text{compute}} = \Omega\left(\frac{d\sqrt{p}}{nc^{3/2}}\right), \quad W_{\text{compute}} = \Omega\left(\frac{dn}{\sqrt{pc}}\right).$$

Adding the bandwidth cost of reading the input matrices and writing the output matrices,

the communication lower bounds are

$$S = \Omega\left(\frac{d\sqrt{p}}{nc^{3/2}}\right), \quad W = \Omega\left(\frac{dn}{\sqrt{pc}} + \frac{n^2}{p}\right). \tag{5.2}$$

## 5.3 Algorithms

Here we discuss existing parallel algorithms and present a few new variants. The algorithms are categorized based on how they partition the computational cuboid [25]. We call an algorithm a $k$D algorithm if it partitions $k$ dimensions of the cuboid. (See Figures 5.1b-5.1d.) The constants on the leading order terms can be compared across all analyzed algorithms, so we are going to write the constants inside the big-O notation even though they do not affect the asymptotic values.

### 1D algorithms

1D algorithms partition only one dimension of the cuboid, i.e., slicing the cuboid into planes. They logically arrange processors into a ring (1D torus) topology and partition matrices along a single dimension. Only one matrix needs to be passed around. The algorithms alternate between shifting the matrix and computing a local multiplication based on the newly received matrix part. We only analyze communicating $A$ since it is asymptotically cheaper than communicating $B$ or $C$, given our assumptions that the dimensions of $B$ are relatively large. Recall that we will use $\text{nnz}(\cdot)$ to represent the number of elements in a matrix, whether it is sparse or dense.

**1D block column:**

All matrices are in block column layout. Processor $P(i)$ has $A^{\{1\times p\}}(i)$ and $B^{\{1\times p\}}(i)$, and is responsible for computing $C^{\{1\times p\}}(i)$. For $p$ rounds, each processor multiplies local $A$ and $B$, accumulates to result to the local $C$, then shifts $A$ to get the new part of $A$. In round $k$, $P(i)$ calculates $A^{\{1\times p\}}(i+k)B^{\{p\times p\}}(i+k,i)$. After $p$ rounds, we have the full multiplication,

$$\begin{aligned} C^{\{1\times p\}}(i) &= \sum_{k=0}^{p-1} A^{\{1\times p\}}(i+k)B^{\{p\times p\}}(i+k,i) \\ &= AB(i). \end{aligned} \tag{5.3}$$

See Figure 5.2b (with $c = 1$) and Algorithm 9 for illustration and pseudocode. Each round a processor sends one message of size $\text{nnz}(A)/p$ words. Therefore, the communication costs are,

$$S = O(p), \quad W = O(\text{nnz}(A)).$$

**Algorithm 9** 1D block column or inner product

**Input:** $P$, a 1-dimensional processor mesh of size $p$.
**Input:** $A$, partitioned in block-row or block-column layout to $p$ equal parts so $A(i)$ is in buffer $\tilde{A}$ on $P(i)$.
**Input:** $B$, distributed so $B^{\{1\times p\}}(i)$ is in buffer $\tilde{B}$ on $P(i)$.
**Output:** $C = AB$, distributed so $C^{\{1\times p\}}(i)$ is in buffer $\tilde{C}$ on $P(i)$.

1: **for each** $P(i)$, in parallel, **do**
2: $\quad\tilde{C} \leftarrow 0$.
3: $\quad$**for** $k \in \{0, 1, \ldots, p-1\}$ **do**
4: $\quad\quad$**if** 1D block column **then**
5: $\quad\quad\quad\tilde{C} \leftarrow \tilde{C} + \tilde{A}\tilde{B}^{\{p\times 1\}}(i+k)$. ▷ Multiplies $A$ with an appropriate block of $\tilde{B}$.
6: $\quad\quad$**else if** 1D block inner product **then**
7: $\quad\quad\quad\tilde{C}^{\{p\times 1\}}(i+k) \leftarrow \tilde{A}\tilde{B}$. ▷ Stores the result into an appropriate block of $\tilde{C}$.
8: $\quad\quad$**end if**
9: $\quad\quad$Shift $\tilde{A}$ by 1.
10: $\quad$**end for**
11: **end for**

**1D block inner product:**

$A$ is in block row layout. $B$ and $C$ are in block column layout. $P(i)$ owns $A^{\{p\times 1\}}(i)$ and $B^{\{1\times p\}}(i)$, and must compute $C^{\{1\times p\}}(i)$. We shift $A$ to get $C$ in block column layout. In round $k$, $P(i)$ computes,

$$C^{\{p\times p\}}(i+k, i) = A^{\{p\times 1\}}(i+k)B^{\{1\times p\}}(i).$$

After $p$ rounds, the whole matrix $C$ is filled. See Figure 5.2d (with $c = 1$) and Algorithm 9 for illustration and pseudocode. Each round a processor sends one message of size $\text{nnz}(A)/p$ so the communication costs are the same as the 1D block column variant,

$$S = O(p), \quad W = O(\text{nnz}(A)).$$

**1D block row** and **1D block outer product** require passing dense matrices $B$ or $C$ around, so they are omitted.

## 2D algorithms

2D algorithms split two dimensions of the computational cuboid, e.g., into pencils of length $q$. They logically arrange processors into a 2D grid of size $p_m \times p_n$. There are many variants including Cannon's algorithm [4] and SUMMA [80]. Since both algorithms have similar costs, we will only discuss the stationary-$C$ SUMMA algorithm because it is more generalizable and more widely used. We assume $p_m = p_n = \sqrt{p}$ for simplicity.

**2D SUMMA** calculates $b$ outer products, where $b$ is a blocking factor. Here we assume $b = \sqrt{p}$. All matrices are stored in 2D block layout; each is of size $n/\sqrt{p} \times n/\sqrt{p}$. Processor $P(i,j)$ owns $A^{\{\sqrt{p}\times\sqrt{p}\}}(i,j)$ and $B^{\{\sqrt{p}\times\sqrt{p}\}}(i,j)$ and computes $C^{\{\sqrt{p}\times\sqrt{p}\}}(i,j)$. In round $k$, $P(i,k)$ broadcasts its $A$ to $P(i,:)$ and $P(k,j)$ broadcasts its $B$ to $P(:,j)$. All processors then calculate

$$A^{\{\sqrt{p}\times\sqrt{p}\}}(i,k)B^{\{\sqrt{p}\times\sqrt{p}\}}(k,j),$$

and accumulate the product in their local $C$'s. After $\sqrt{p}$ rounds, we have,

$$\begin{aligned} C^{\{\sqrt{p}\times\sqrt{p}\}}(i,j) &= \sum_{k=0}^{\sqrt{p}-1} A^{\{\sqrt{p}\times\sqrt{p}\}}(i,k)B^{\{\sqrt{p}\times\sqrt{p}\}}(k,j) \\ &= A(i)B(j). \end{aligned}$$

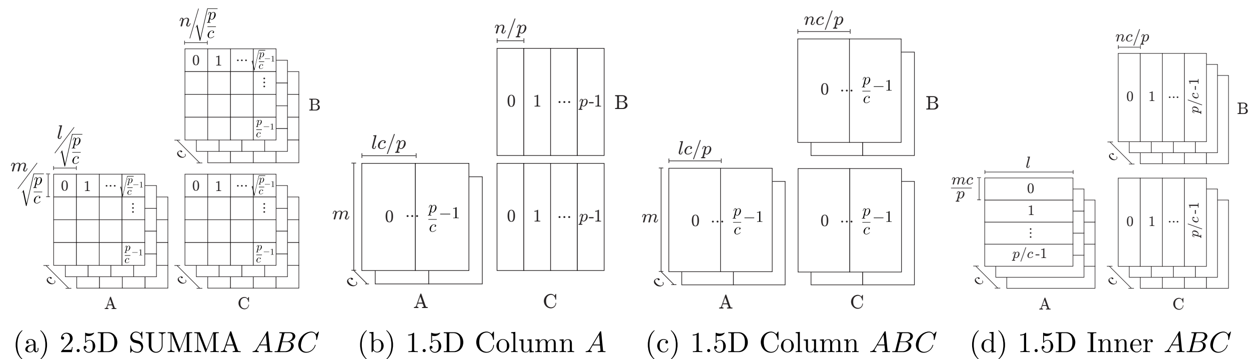
See Figure 5.2a and Algorithm 10 (with $c = 1$) for illustration and pseudocode. There are $\sqrt{p}$ broadcasts of $A$ ($\log\sqrt{p}$ messages and $\text{nnz}(A)/p$ words each), and $\sqrt{p}$ broadcasts of $B$ ($\log\sqrt{p}$ messages and $\text{nnz}(B)/p$ words each). Therefore,

$$S{=}O(2\sqrt{p}\log\sqrt{p}), \quad W{=}O\left(\frac{\text{nnz}(A){+}\text{nnz}(B)}{\sqrt{p}}\log\sqrt{p}\right).$$

## 2.5D and 3D algorithms

2.5D and 3D algorithms partition all three dimensions of the computational cuboid, e.g., into subcubes, length-$n/2$ pencils, etc.

**2.5D and 3D SUMMA algorithms** [3, 167] utilize replication to avoid communication. It logically arranges processors into a 3D $p_m \times p_n \times c$ mesh. In essence, it is $c$ layers of 2D




(a) 2.5D SUMMA $ABC$ (b) 1.5D Column $A$ (c) 1.5D Column $ABC$ (d) 1.5D Inner $ABC$

Figure 5.2: Processor mesh layouts for 1.5D and 2.5D algorithms. Matrix names at the end indicate that they are being replicated. (Substituting $c = 1$ gives corresponding 1D and 2D algorithms.)

**Algorithm 10** 2.5D SUMMA (Substitute $c = 1$ for 2D SUMMA)

**Input:** $P$, a processor grid of size $\sqrt{p/c} \times \sqrt{p/c} \times c$.
**Input:** $A$, distributed so $A^{\{p/c \times p/c\}}(i, j)$ is on $P(i, j, :)$.
**Input:** $B$, distributed so $B^{\{p/c \times p/c\}}(i, j)$ is on $P(i, j, :)$.
**Output:** $C = AB$, distributed so $C^{\{p/c \times p/c\}}(i, j)$ is in buffer $\tilde{C}$ on $P(i, j, 0)$.

1: **for each** $P(i, j, \ell)$, in parallel, **do**
2: $\tilde{C} \leftarrow 0$.
3: **for** $k \in \{\ell d, \ell d + 1, \ldots, \ell d + d - 1\}$ **do**
4: $P(i, k, \ell)$ broadcasts $A(i, k)$ to buffer $\tilde{A}$ of $P(i, :, \ell)$.
5: $P(k, j, \ell)$ broadcasts $B(k, j)$ to buffer $\tilde{B}$ of $P(:, j, \ell)$.
6: $\tilde{C} \leftarrow \tilde{C} + \tilde{A}\tilde{B}$.
7: **end for**
8: $P(i, j, 0)$ sumreduces $\tilde{C}$ from $P(i, j, :)$. ▷ Stores output.
9: **end for**

SUMMA algorithm with $p_m \times p_n$ processor grids, except that each layer only computes $1/c$ of the outer products. We assume $p_m = p_n = \sqrt{p/c}$ for simplicity, as illustrated in Figure 5.2a. Algorithm 10 shows the pseudocode. The 2D and 3D algorithms are special cases of the 2.5D algorithm where $c = 1$ and $\sqrt[3]{p}$, respectively. Therefore, from now on we will use the word 2.5D algorithms to represent all 2D, 2.5D, and 3D algorithms.

Converting from a 2D layout to a 2.5D layout requires preprocessing. We call this *replication* since it also makes $c$ processors hold the same blocks of each matrix. This can be done by exchanging blocks within a group of $c$ processors and on each processor concatenating into larger blocks. The algorithm partitions all matrices into $\sqrt{p/c} \times \sqrt{p/c}$ equal parts and arranges the processors so that $A(i, j)$, $B(i, j)$, and $C(i, j)$ are on $P(i, j, :)$. These $P(i, j, :)$ cooperate as a team on computing $C(i, j)$. There are a total of $\sqrt{p/c}$ outer products. Each layer is responsible for $(\sqrt{p/c})/c = d$ outer products. $P(i, j, \ell)$ calculates

$$C^{\{\sqrt{\frac{p}{c}} \times \sqrt{\frac{p}{c}}\}}(i, j) = \sum_{k=\ell d}^{\ell d + d - 1} A^{\{\sqrt{\frac{p}{c}} \times \sqrt{\frac{p}{c}}\}}(i, k) B^{\{\sqrt{\frac{p}{c}} \times \sqrt{\frac{p}{c}}\}}(k, j).$$

In round $k$, $P(i, k, \ell)$ broadcasts its $A$ to $P(i, :, \ell)$ and $P(k, j, \ell)$ broadcasts its $B$ to $P(:, j, \ell)$, then each processor computes the product. After $d$ rounds, $P(i, j, :)$ do a sum reduction on $C$ to get the final result. This costs $d$ broadcasts of $A$ ($\log \sqrt{p/c}$ messages and $c \cdot \mathrm{nnz}(A)/p \log \sqrt{p/c}$ words each), $d$ broadcasts of $B$ ($\log \sqrt{p/c}$ messages and $c \cdot \mathrm{nnz}(B)/p \log \sqrt{p/c}$ words each), and one reduction of $C$ ($\log c$ messages and $c \cdot \mathrm{nnz}(C)/p \log c$ words).

As for the replication cost, we will model it as $P(i, j, 0)$ gathering all matrices from $P(i, j, :)$ then broadcasting the concatenated matrices back to them. Replicating A takes one gather ($\log c$ messages and $c \cdot \mathrm{nnz}(A)/p \log c$ words) and one broadcast ($\log c$ messages and $c \cdot \mathrm{nnz}(A)/p \log c$ words). Replicating B takes one gather ($\log c$ messages and $c \cdot \mathrm{nnz}(B)/p \log c$

words) and one broadcast ($\log c$ messages and $c \cdot \text{nnz}(B)/p \log c$ words). Let $W_{\text{rep}}$ represent the bandwidth costs of replication and reduction combined,

$$W_{\text{rep}} = \frac{2\,\text{nnz}(A) + 2\,\text{nnz}(B) + \text{nnz}(C)}{p} c \log c.$$

The total communication costs are,

$$S = O\left(2\frac{\sqrt{p}}{c^{3/2}} \log \sqrt{\frac{p}{c}} + 5 \log c\right),$$
$$W = O\left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{pc}} \log \sqrt{\frac{p}{c}} + W_{\text{rep}}\right).$$

## 1.5D algorithms

2.5D algorithms communicate at least one dense matrix while 1D algorithms can limit the movement to just the sparse matrix $A$. This section applies replication to 1D algorithms to avoid more communication. The first algorithm simply increases the size of matrix $A$ that is stored locally. The latter two algorithms are similar to the communication-avoiding $N$-body algorithms in Chapter 3 which also operate on a ring topology. We will still use $c$ for the replication factor.

### 1.5D block column replicating $A$ (ColA):

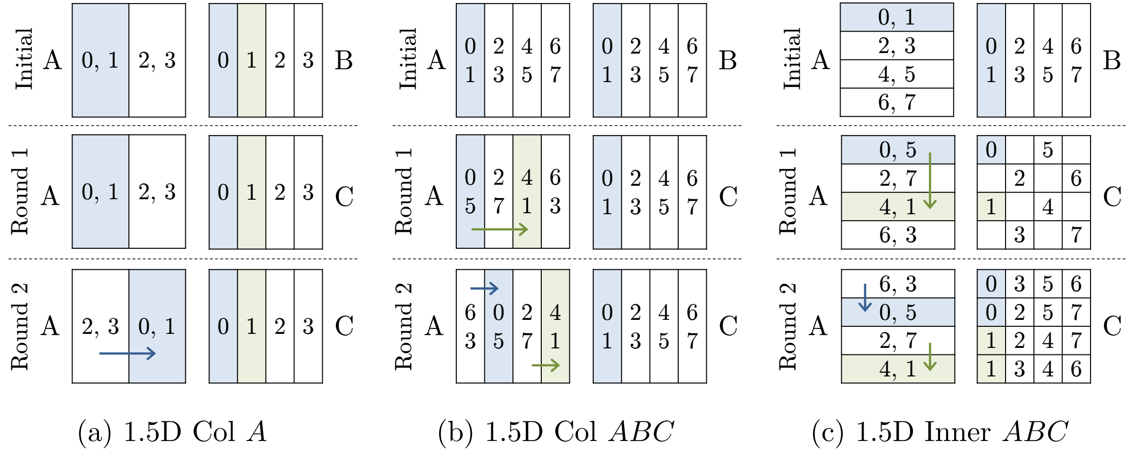
We start by replicating just the sparse matrix $A$ $c$ times, $1 \leq c \leq p$. As shown in Algorithm 11, processor $P(i)$ has $A^{\{1\times p/c\}}(i/c)$ and $B^{\{1\times p\}}(i)$, and computes

$$C^{\{1\times p\}}(i) = \sum_{k=0}^{p/c-1} A^{\{1\times p/c\}}(i+k) B^{\{p/c\times p\}}(i+k, i),$$

**Algorithm 11** 1.5D block column replicating $A$ (ColA)

**Input:** $P$, a 1-dimensional processor mesh of size $p$.
**Input:** $A$, distributed so $A^{\{1\times p/c\}}(\lfloor i/c \rfloor)$ is in buffer $\tilde{A}$ on $P(i)$.
**Input:** $B$, distributed so $B^{\{1\times p\}}(i)$ is in buffer $\tilde{B}$ on $P(i)$.
**Output:** $C = AB$, distributed so $C^{\{1\times p\}}(i)$ is in buffer $\tilde{C}$ on $P(i)$.

```
1: for each P(i), in parallel, do
2:     C̃ ← 0.
3:     for k ∈ {0, 1, ..., p/c − 1} do
4:         C̃ ← C̃ + ÃB̃^{p/c×1}(⌊i/c⌋ + k).
5:         Shift Ã by c.
6:     end for
7: end for
```




(a) 1.5D Col $A$ (b) 1.5D Col $ABC$ (c) 1.5D Inner $ABC$

Figure 5.3: Example computations of *1.5D block column A*, *1.5D block column ABC*, and *1.5D block inner product ABC* on 8 processors. Numbers in the grid are processor ranks. $B$ is fixed so it is drawn only once.

where $k$ is the round number. Figure 5.2b illustrates the layout. See Figure 5.3a for an example with $p = 8$ and $c = 2$. For $p/c$ rounds, each processor computes the product and shifts $A$ among processors in the same layer by one. Latency cost is improved by a factor of $c$, but the total bandwidth cost stays the same since the message size is also increased by $c$. Replicating $A$ takes $2 \log c$ messages and $2c \cdot \text{nnz}(A)/p \log c$ words. The total costs are,

$$S = O(2 \log c + p/c),$$
$$W = O\left(\frac{2c \cdot \text{nnz}(A)}{p} \log c + \text{nnz}(A)\right).$$

**1.5D block inner product replicating $A$ (InnerA):** InnerA also replicates just the sparse matrix $A$, but has $A$ in a block row layout and $B$ in a block column layout, forming an inner product. Processor $P(i)$ has $A^{\{p/c \times 1\}}(i/c)$ and $B^{\{1 \times p\}}(i)$. InnerA has the same communication costs and output layout as ColA so we omit further analysis.

### 1.5D block column replicating all matrices (ColABC):

Next, we investigate paying an extra cost of also replicating the dense matrices $B$ and $C$ in an attempt to reduce more shifting costs asymptotically. This algorithm groups $p$ processors into a $p/c \times c$ grid. See Figure 5.2c for illustration. $P(i, :)$ have $A^{1 \times p/c}(i)$ and $B^{1 \times p/c}(i)$, and work as a team to compute

$$C^{\{1 \times p/c\}}(i) = \sum_{\kappa=0}^{p/c-1} A^{\{1 \times p/c\}}(\kappa) B^{\{p/c \times p/c\}}(\kappa, j).$$

All $c$ team members split these $p/c$ summation terms equally. $P(i, \ell)$ computes $p/c^2 = d$ terms from $\kappa = ld$ to $(l + 1)d - 1$. This computation pattern can be done by first shifting

---

**Algorithm 12** 1.5D block column replicating all matrices (ColABC)

---

**Input:** $P$, a 2-dimensional processor mesh of size $p/c \times c$.
**Input:** $A$, distributed so $A^{\{1\times p/c\}}(i)$ is in buffer $\tilde{A}$ on $P(i,:)$.
**Input:** $B$, distributed so $B^{\{1\times p/c\}}(i)$ is in buffer $\tilde{B}$ on $P(i,:)$.
**Output:** $C = AB$, distributed so $C^{\{1\times p/c\}}(i)$ is in buffer $\tilde{C}$ on $P(i,:)$.

1: **for each** $P(i,\ell)$, in parallel, **do**
2: $\quad d \leftarrow p/c^2$. ▷ The number of rounds each processor needs to do.
3: $\quad$ Shift $\tilde{A}$ by $\ell d$. ▷ Initial shift so $P(i,\ell)$ has $A(i+\ell d)$.
4: $\quad \tilde{C} \leftarrow 0$.
5: $\quad$ **for** $k \in \{\ell d, \ell d+1, \ldots, \ell d+d-1\}$ **do**
6: $\quad\quad \tilde{C} \leftarrow \tilde{C} + \tilde{A}\tilde{B}^{\{p/c\times 1\}}(i+k)$.
7: $\quad\quad$ Shift $\tilde{A}$ by 1.
8: $\quad$ **end for**
9: $\quad P(i,0)$ sumreduces $\tilde{C}$ from $P(i,:)$.
10: **end for**

---

$A$ in the same layer by distance $\ell d$ (to jump to the starting point), then all processors can alternate between multiplication and shifting by one as usual for $d$ rounds, and reduce $C$ at the end. In other words: ColABC takes $d$ rounds. In round $k$, $P(i,\ell)$ calculates,

$$C^{\{1\times p/c\}}(:,i) = \sum_{k=\ell d}^{\ell d+d-1} A^{\{1\times p/c\}}(k) B^{\{p/c\times p/c\}}(k,j).$$

Figure 5.3b shows an example with $p=8$ and $c=2$. Algorithm 12 shows the pseudocode.

Replication and reduction cost the same as 2.5D SUMMA's. The matrix $A$ is shifted $p/c^2$ times, so it takes $p/c^2$ messages and $p/c^2 \cdot c \cdot \text{nnz}(A)/p = \text{nnz}(A)/c$ words. The total communication costs are,

$$S = O\left(5\log c + \frac{p}{c^2}\right), \quad W = O\left(\frac{\text{nnz}(A)}{c} + W_{\text{rep}}\right).$$

**1.5D block inner product replicating all matrices (InnerABC):**

Next, we apply replication to the inner product algorithm. This algorithm also groups $p$ processors into $p/c \times c$ grid, except this time $P(i,:)$ have $A^{\{p/c\times 1\}}(i)$ and $B^{\{1\times p/c\}}(i)$, and compute $C^{\{1\times p/c\}}(i)$ together as a team. See Figure 5.2d and Algorithm 13 for illustration and pseudocode.

There are $p/c$ block inner products to do, and each team member does $p/c^2 = d$ of them. $P(i,\ell)$ computes

$$C^{\{p/c\times p/c\}}(i+k,i) = A^{\{p/c\times 1\}}(i+k) B^{\{1\times p/c\}}(i)$$

for $\ell d \le k < (\ell+1)d$. It does so by initially shifting $A$ by $\ell d$ to start at the required offset then alternating between multiplication and shifting by one for $d$ rounds. Finally,

**Algorithm 13** 1.5D block inner product replicating all matrices (InnerABC)

**Input:** $P$, a 2-dimensional processor mesh of size $p/c \times c$.
**Input:** $A$, distributed so $A^{\{p/c\times 1\}}(i)$ is in buffer $\tilde{A}$ on $P(i,:)$.
**Input:** $B$, distributed so $B^{\{1\times p/c\}}(i)$ is in buffer $\tilde{B}$ on $P(i,:)$.
**Output:** $C = AB$, distributed so $C^{\{1\times p/c\}}(i)$ is in buffer $\tilde{C}$ on $P(i,:)$.

```
1: for each P(i, ℓ), in parallel, do
2:     d ← p/c².                          ▷ The number of rounds each processor needs to do.
3:     Shift Ã by ℓd.                     ▷ Initial shift so P(i, ℓ) has A(i + ℓd).
4:     for k ∈ {ℓd, ℓd + 1, ..., ℓd + d − 1} do
5:         C̃^{p/c×1}(i + k) ← ÃB̃.
6:         Shift Ã by 1.
7:     end for
8:     P(i, 0) gathers C̃ from P(i, :).
9: end for
```

the algorithm gathers the final matrix $C$ to $P(i,0)$ on the first layer. Figure 5.3c shows an example with $p = 8$ and $c = 2$.

Shifting $A$ costs $p/c^2$ messages and $\text{nnz}(A)/c$ words. Gathering $C$ costs the same as reduction asymptotically, so the total communication costs are,

$$S = O\left(5\log c + \frac{p}{c^2}\right), \quad W = O\left(\frac{nnz(A)}{c} + W_{\text{rep}}\right).$$

## Comparison

We compare our 1.5D algorithms, ColA, ColABC, and InnerABC, with the classic 2.5D SUMMA algorithm which will be called SummaABC from now on, with *ABC* indicating that it replicates all three matrices. Table 5.1 summarizes all communication costs of all replicating algorithms. The costs of 1D and 2D algorithms can be obtained by substituting $c = 1$ into 1.5D and 2.5D algorithms, respectively. None of the presented algorithms obtained the communication lower bounds, although SummaABC has quite similar costs.

Communication consists of three phases, replication, propagation, and collection. **Replication** is the gathering of neighboring matrices and the broadcasting of the concatenated matrix and is only used in .5D algorithms. **Propagation** is the communication within the multiplication steps to get the necessary blocks for each local multiplication. It corresponds to the shiftings of $A$ in 1D and 1.5D algorithms, and the broadcastings of $A$ and $B$ in 2.5D algorithms. **Collection** refers to reduction or gathering of $C$ at the end after all multiplications are done which only occurs in .5D algorithms.

Even though ColABC and InnerABC have the same *asymptotic* costs, InnerABC uses gather in the collection phase which can be significantly faster than ColABC's reduction in practice. They also store matrices in different layouts, which can have different local matrix

| Algorithms | #messages = $S$ | | | #words = $W$ | | |
|---|---|---|---|---|---|---|
| | Replication | Propagation | Collection | Replication | Propagation | Collection |
| 1.5D Col A | $2\log c$ | $\frac{p}{c}$ | - | $2\frac{\text{nnz}(A)}{p}c\log c$ | $\text{nnz}(A)$ | - |
| 1.5D Col ABC | $4\log c$ | $\frac{p}{c^2}$ | $\log c$ | $2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$ | $\frac{\text{nnz}(A)}{c}$ | $\frac{\text{nnz}(C)}{p}c\log c$ |
| 1.5D Inner ABC | $4\log c$ | $\frac{p}{c^2}$ | $\log c$ | $2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$ | $\frac{\text{nnz}(A)}{c}$ | $\frac{\text{nnz}(C)}{p}c\log c$ |
| 2.5D SUMMA ABC | $4\log c$ | $2\frac{\sqrt{p}}{c^{3/2}}\log\sqrt{\frac{p}{c}}$ | $\log c$ | $2\frac{\text{nnz}(A)+\text{nnz}(B)}{p}c\log c$ | $\frac{\text{nnz}(A)+\text{nnz}(B)}{\sqrt{pc}}\log\sqrt{\frac{p}{c}}$ | $\frac{\text{nnz}(C)}{p}c\log c$ |

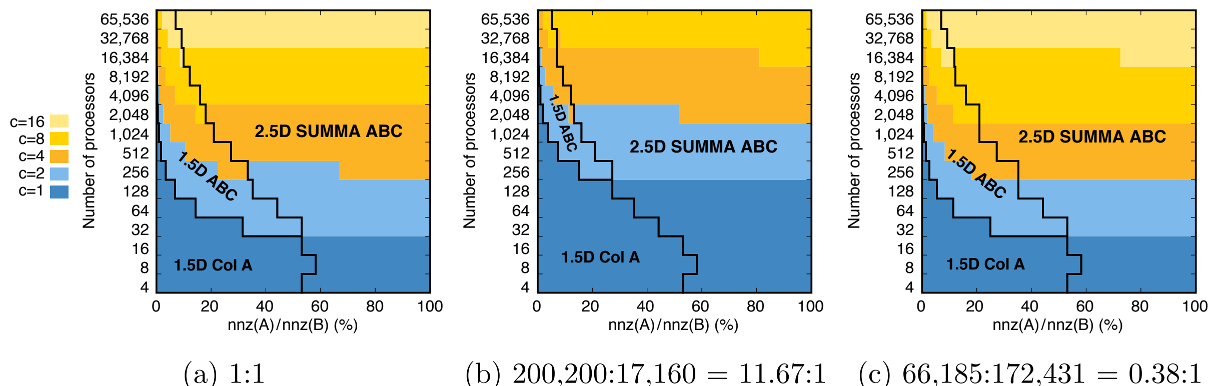
Table 5.1: Algorithm communication costs. Uppercase letters at the end of algorithm names indicate the matrices being replicated.

multiplication efficiencies. The storage format for C is key to performance in some context as well. We will discuss this in Section 5.5.

There are limits to the effective replication factors for each algorithm. For ColA, $c = p$ corresponds to replicating the whole matrix $A$. Therefore, $c \leq p$ is the limit. ColABC and InnerABC have $c \leq \sqrt{p}$, since when $c = \sqrt{p}$, each processor layer only computes one round of local matrix multiplication – any larger $c$'s would leave some layer(s) idle. The same reasoning applies to SummaABC algorithm whose upper limit is $c \leq \sqrt[3]{p}$ in which case each layer only computes one outer product.

**Latency costs** are not dependent on matrix inputs, but purely on the number of processors $p$ and the replication factor $c$. Out of all three phases, the propagation cost grows fastest with $p$ and is the dominating cost. The best latency cost for propagation is $O(1)$ and is attainable by all algorithms with their highest effective $c$'s. However, higher $c$'s mean more memory requirement and increased bandwidth costs for replication and collection. For a fixed $c$, SummaABC achieves lowest latency costs.

**Bandwidth costs** are based on the number of words sent. It can be computed from latency costs (number of messages sent) and message size (number of words sent in each message). Most analysis in prior work for dense-dense or sparse-sparse cases assumes the message sizes for matrices $A$ and $B$ are the same, which is not reasonable in many cases, especially for sparse-dense matrix multiplication. In that case, SummaABC also minimizes bandwidth costs altogether and is the best algorithm overall. This assumption does not hold in our case, and one can utilize less bandwidth by moving $A$ more and moving $B$ less often than SummaABC does, at the expense of higher latency costs. For example, ColA only moves the sparse matrix $A$ around. It is most likely to have the lowest overall bandwidth cost, but it has the highest latency cost. ColABC and InnerABC opt to replicate the dense matrix $B$ to achieve asymptotically lower propagation latency and bandwidth than ColA. They also have to move the dense matrix $C$ in the collection phase. In other words, they send $(2\,\text{nnz}(B) + \text{nnz}(C))/p \cdot c\log c$ more words to reduce the propagation bandwidth from $\text{nnz}(A)$ to $\text{nnz}(A)/c$. In most practical scenarios, $\text{nnz}(A) \ll \text{nnz}(B), \text{nnz}(C)$, so $p$ has to be considerably large for this trade-off to pay off. When ColABC and InnerABC are not




(a) 1:1 (b) 200,200:17,160 = 11.67:1 (c) 66,185:172,431 = 0.38:1

Figure 5.4: The areas that each algorithm has the theoretically lowest overall bandwidth cost. The X-axis is the ratio of nnz($A$) versus nnz($B$). The Y-axis is the number of processors. There are three subgraphs for three different nnz($C$) : nnz($B$) ratios which are delineated by the jagged lines within each plot. *1.5D ABC* stands for both Col ABC and InnerABC. The area for *1.5D ABC* includes the area for *1.5D Col A*. The best replication factors for the best algorithm at each data point are shown in colors. ColA is best for sparser matrices or lower concurrency while SummaABC is the opposite. *1.5D ABC* algorithms help improve the scalability of ColA.

replicating, they have equal overall bandwidth costs to ColA. SummaABC moves the dense matrix $B$ in every phase, so it is unlikely to beat any of the 1.5D algorithms in terms of bandwidth when $A$ is very sparse. It will be preferable again when nnz($A$) becomes closer to nnz($B$), decreasing the message-size imbalance, or when the number of processors grows large (since it minimizes latency).

It is best to obtain hardware parameters to determine this latency-bandwidth trade-off across the various algorithms. However, we can gain some intuition over where each algorithm is most suitable for without being specific to any particular machine. We found that the bandwidth costs are more prominent in our experiments, so we focus our analysis on those costs for simplicity. To eliminate one variable, we divide the bandwidth costs in Table 5.1 with nnz($B$) and represent the nonzero ratios nnz($A$)/ nnz($B$) and nnz($C$)/ nnz($B$) with $f$ and $g$, respectively. Knowing $g$, we can plot with $p$ and $f$ as axes and search for the best algorithm over all possible $c$'s at each point. We picked three different values of $g$, 1:1 in Figure 5.4a, 11.67:1 in Figure 5.4b, and 0.38:1 in Figure 5.4c. For an SpDM$^3$ problem, nnz($C$) : nnz($B$) $\approx m : q$ and can be interpreted as the *tallness* of matrix $A$. For example, 1:1 means a square $A$, 11.67:1 applies to a tall $A$, and 0.38:1 refers to a short-wide $A$. We draw black lines to separate regions in which different algorithms are optimal and use colors to show the best replication factors for the optimal algorithm. The best replication factor for ColA is always 1 because increasing $c$ does not reduce bandwidth. The area in which ColA wins is a subset of the area that ColABC and InnerABC win because ColABC at $c = 1$ is equivalent to ColA. The graphs confirm the intuition from the earlier analysis that ColA is most suitable for very sparse matrices or small-scale runs. ColABC and InnerABC can help improve scalability to some level, but eventually SummaABC wins as we move towards

larger concurrency or denser matrices.

Since this analysis is based on just nnz($A$), nnz($B$), and nnz($C$), it is trivially applicable to sparse-sparse matrix-matrix multiplication (of different sparsities and/or sizes) or even dense-dense matrix-matrix multiplication (of different sizes).

## 5.4 Performance Results

We implemented all four algorithms listed in Table 5.1 using C++ and MPI. $A$ is stored in zero-based indexing Compressed Sparse Row (CSR) format. $B$ and $C$ are stored in row-major format, except where noted. We used the multi-threaded Intel® Math Kernel Library (MKL) for local sparse-dense matrix-matrix multiplication (*mkl_dcsrmm*). (The Compressed Sparse Column (CSC) format would scale better in terms of storage for the blocked column algorithms, but we found MKL's multiplication routine for the CSC format (*mkl_dcscmm*) significantly slower than the CSR's (*mkl_dcsrmm*), so we used CSR format in all implementations.) We ran our experiments on Edison, a Cray XC30 machine at the National Energy Research Scientific Computing Center (NERSC). Edison has a Cray Aries interconnect with a Dragonfly topology and consists of 5,576 compute nodes, each with 2 sockets of 12-core Intel Ivy Bridge processors running at 2.4GHz and with 64 GB memory. We used Intel's C++ compiler (icpc) version 15.0.1, Intel MKL version 11.2.1, and Cray MPICH version 7.3.1. All benchmarks are run with 2 MPI processes per node and 12-way multi-threaded MKL operation per process (hybrid configuration). We did not utilize Intel's Hyper-Threading Technology nor Turbo Boost Technology since these would have led to high performance variance.

### Trends in Communication Costs

Figure 5.5 shows the cost breakdown of all algorithms running on 3,072 processors (256 MPI processes). $A$ is an Erdős-Rényi matrix with $n$=65,536 and 41 nonzeroes per row on average (0.0625% nonzeroes). The first two bars on the left belongs to SummaABC where all three matrices are replicated 1 (i.e., not at all) and 4 times, respectively. The next group is the ColA algorithm in which $A$ is partitioned into block columns and replicated with the factors ($c$) shown above the algorithm's name. The last two groups are ColABC and InnerABC with similar replication factors ($c$) shown in each label. All costs in the stacked bars are average costs over all processors. We exclude the bar for ColABC at $c = 16$ from Figure 5.5 because it is too tall.

The computation times in green are unequal even though all algorithms do the same amount of work. This is because the local MKL matrix-multiplication routine has varying efficiency for different shapes of input matrices. Figure 5.6 shows MKL performance for all of the relevant shapes and explains the variability in computation time in our algorithms. In general, MKL performs better on larger matrices, since they have higher computational intensity, although there is a dropoff in one case, perhaps due to suboptimal blocking. Sum-

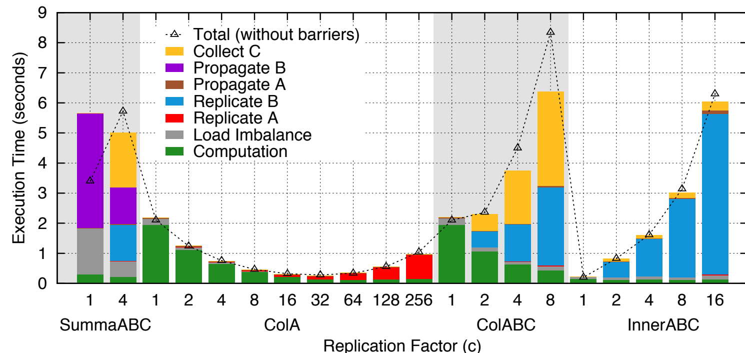



Figure 5.5: Cost breakdown for multiplying a square Erdős-Rényi matrix of size $n$ = 65,536 with 41 nonzeroes per row on average, with a dense matrix of the same size on $p$ = 3,072 cores of Cray XC30 (256 MPI processes). All three 1.5D algorithms, with their best replication factors, outperform SummaABC which is the only algorithm that moves the dense matrix $B$. InnerABC has the best total running time. We inserted barriers to get accurate breakdown costs. The total running time without barriers is shown in dotted line. InnerABC at $c = 1$ has the best overall running time.

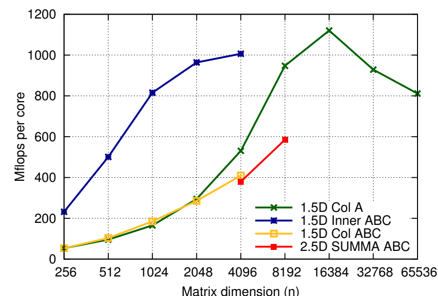



Figure 5.6: Single-node MKL matrix multiplication efficiency for different matrix shapes used in Figure 5.5. $n$ refers to different matrix-multiplication shapes for each algorithm: $n \times n \times n$ for SummaABC, $65K \times n \times 256$ for ColA, $65K \times n \times n$ for ColABC, and $n \times 65K \times n$ for InnerABC. For these problems, n=4096c for SummaABC and n=256c for the rest.

maABC performs $4K \times 4K \times 4K$ local matrix multiplications when $c = 1$, and $8K \times 8K \times 8K$ when $c = 4$. ColA does $64K \times 256c \times 256$ local matrix multiplications for $1 \leq c \leq 256$. ColABC does $64K \times 256c \times 256c$ local matrix multiplications for $1 \leq c \leq 16$. Finally, InnerABC does $256c \times 64K \times 256c$ local matrix multiplications for $1 \leq c \leq 16$.

Sometimes we found nontrivial variability across processors within an algorithm, even though all are performing roughly the same number of local multiplications on the same shape of matrices. We believe this is due to differences in the nonzero pattern of $A$ that lead to different cache effects. To separate idle time due to load imbalance from useful computation or communication, there is an extra barrier after the computation phase for these time breakdown graphs. The barrier time can be substantial and is shown in gray on top of the computation time. The total height of the stacked bars is the average total runtime of the run with barriers. We also show the maximum total runtime across all processors from similar runs without barriers in black dotted line.

For any of the algorithms with $c > 1$ for $A$ or $B$, the time to replicate those matrices is shown in bright red and blue, respectively. Replication times increase linearly with $c$ as predicted, although barrier costs decrease with $c$ since the set of processors involved in a barrier is smaller.

In each step within the multiplication algorithm, the local matrices are broadcast or shifted right after local matrix multiplication. The time to propagate $A$ (shift or broadcast), and propagate $B$ (broadcast) are in brown and purple, respectively. ColA reduces latency by a factor of $c$ but does not reduce bandwidth, and its shift time stays the same but with moderate variance, which could be because it sends many small messages. Both ColABC

and InnerABC reduce latency by a factor of $c^2$ and bandwidth by a factor of $c$, so we expect between two to four times reduction in *shift* or *broadcast* time as we double $c$. This trend might not be apparent in the graph since the duration is very short and there might be some overheads introduced. The SummaABC algorithm reduces latency by a factor of $c^{3/2}$ and bandwidth by a factor of $\sqrt{c}$, and the graph shows a decrease in communication time by a factor of between $\sqrt{4} = 2$ to $4^{3/2} = 8$ as $c$ is quadrupled. Since the factor seems closer to 2, it means that bandwidth is more prominent than latency on Edison, for our problem.

ColABC and SummaABC require a reduction of matrix $C$ while InnerABC gathers $C$ at the end. All of these are shown in yellow as collection cost. Even though gather asymptotically costs the same as reduce, in practice, it can cost much less, because each processor only sends a message of size ranging from $n^2/p$ to $cn^2/p$, depending on its position in the gather tree, instead of all $cn^2/p$ in reduction. InnerABC has the best overall cost. Comparing the timing without barriers, it is 16.83 times faster (at $c = 1$) than the best running time of SummaABC (also at $c = 1$), whose communication time is the worst because it also propagates the dense matrix.

## Scalability

Figure 5.7 shows strong scaling performance on 384 to 12,288 cores for 65,536 × 65,536 Erdős-Rényi matrices with 1% nonzeroes for $A$. All our non-cost-breakdown graphs were run without barriers. For each algorithm at each number of cores, we report the best speedup over all available replication factors ($c$), so the graphs are not expected to be smooth or monotonic. Since the problem cannot fit into one node, we timed the multiplication on 2 nodes (48 cores) with the same hybrid MPI configuration and excluded communication time for a baseline. We estimated the serial running time by multiplying this measured time by

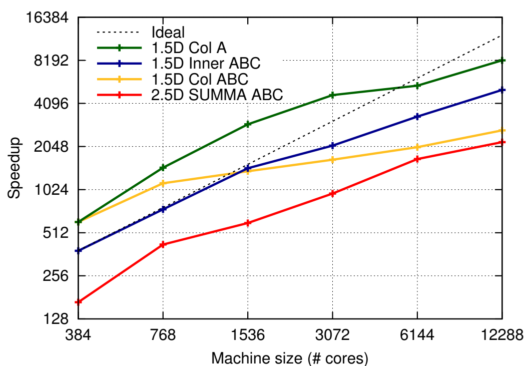



Figure 5.7: Strong scaling of an Erdős-Rényi matrix of size $n$ = 65,536 with 1% nonzeroes on Cray XC30. Each point represents the best running time across all available replication factors ($c$).

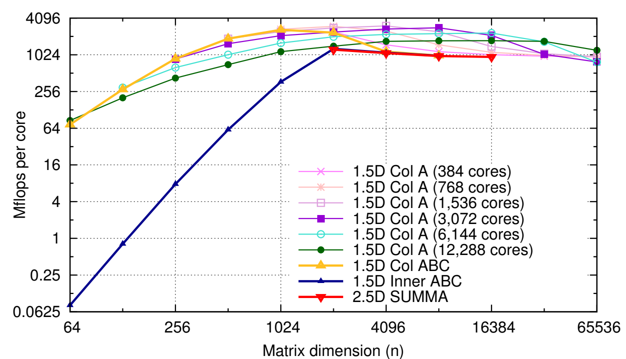



Figure 5.8: Single-node MKL matrix multiplication efficiency for Figure 5.7. $n$ refers to different matrix-multiplication shapes for each algorithm: $n \times n \times n$ for SummaABC, $n \times 65{,}536 \times n$ for InnerABC, 65,536 $\times n \times n$ for ColABC, and $65{,}536 \times n \times k$ for ColA, where $k = 2\text{K}, 1\text{K}, \ldots, 64$ for $p = 384, 768, \ldots, 12\text{K}$ cores.

| Fig. | Number of processors | | | | | |
|---|---|---|---|---|---|---|
| | 384 | 768 | 1,536 | 3,072 | 6,144 | 12,288 |
| 5.7 | 2,①,1,1 | 1,1,1,② | 2,1,1,⑧ | 1,2,1,⑯ | 2,2,2,㉜ | 4,1,4,⑧ |
| 5.9a | 2,1,①,1 | 1,1,①,2 | 2,1,①,4 | 1,1,1,⑧ | 2,2,2,⑯ | 4,2,2,㉜ |
| 5.9b | 2,1,1,② | 1,1,1,② | 2,1,1,④ | 1,1,1,⑧ | 2,2,2,⑯ | 4,2,2,㉜ |
| 5.11a | 2,1,1,② | 1,1,1,⑧ | 2,1,1,⑧ | 1,2,1,⑯ | 2,2,2,⑧ | - |
| 5.12a | - | 1,1,①,64 | 2,1,①,64 | 1,2,①,64 | 2,2,①,64 | - |
| 5.13a | 2,1,①,32 | 1,1,①,32 | 2,1,①,32 | 1,1,①,32 | 2,1,1,㉜ | - |

Table 5.2: The best replication factors ($c$) for each strong or weak scaling graph in this chapter. Each cell lists the replication factors of the algorithms in the following order: SummaABC, ColABC, InnerABC, and ColA. The winning algorithm in each cell is circled. A dash means we did not run an experiment for that configuration.

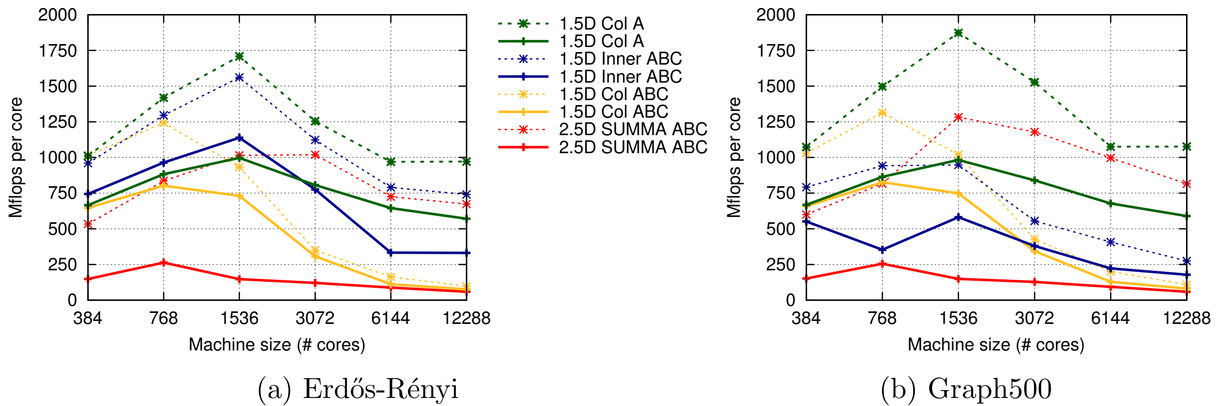
48. The black dotted line indicates the ideal speedup.

The superlinear speedup of ColA and ColABC at the beginning was because of significantly faster computation time (again due to different MKL performance with different matrix sizes, see the trends in Figure 5.8). They both fell to sublinear speedup at larger scales where the edge in computation time was gone and the increasing communication time dominated. ColABC was also faster than InnerABC because of the faster computation time. SummaABC was outperformed by ColA by factors from 3.25× to 4.89×, but it still has decent scalability.

We report the best replication factors in Table 5.2. This experiment maps to a line in Figure 5.4a at $\text{nnz}(A)/\text{nnz}(B) = 1\%$ from 32 to 1,024 MPI processes. The figure predicts any of the 1.5D algorithms could win with $c = 1$ (or any $c$ in case of ColA), which is true because ColABC wins at 384 cores with $c = 1$, then ColA wins the rest with $c > 1$. We also see larger replication factors as the number of cores increases, consistent with the trend in Figure 5.4. Most algorithms have best $c$'s greater than 1 on 6,144 cores onwards.

## Non-uniform Distribution

Next, we experiment with matrices with non-uniform nonzero distribution. A Graph500 matrix $A$ is generated with RMAT parameters $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$ [86]. Using these parameters, RMAT is known to create a matrix with skewed distribution (approximating a power-law distribution if some noise is added [160]) of nonzero row and column counts. We deviated from the average edge factor (nonzero row/column count) suggested by the Graph500 benchmark to stay consistent with the density of Erdős-Rényi matrices we used. We also modified our 1.5D algorithms to partition work based on equal number of nonzeroes (using greedy algorithm) instead of number of rows or columns to mitigate the expected load imbalance. We still partition matrices based on equal numbers of rows and columns for SummaABC.




(a) Erdős-Rényi (b) Graph500

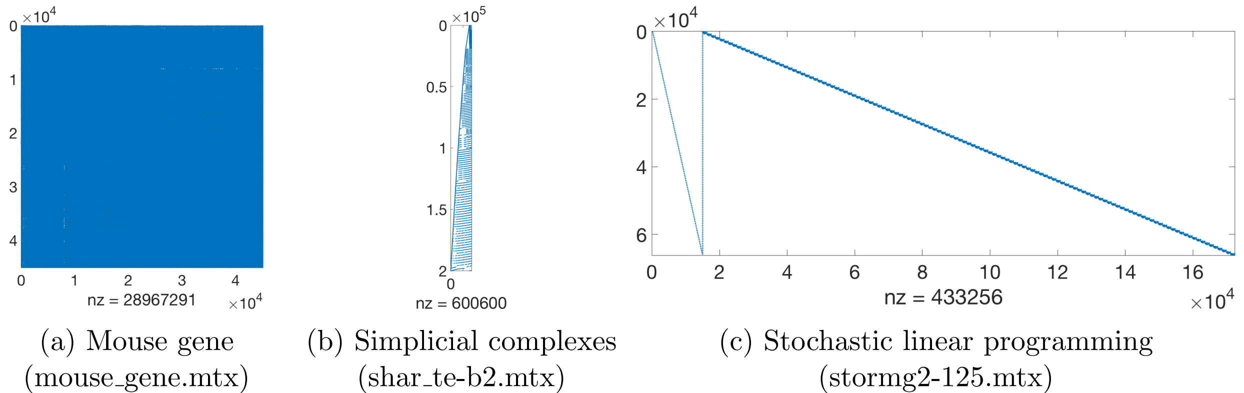
Figure 5.9: Weak Scaling of square Erdős-Rényi and Graph500 matrices with fixed $d = 164$ nonzeroes per row on $p = 384, 768, 1536, 3072, 6144$, and 12288 cores of Cray XC30 with corresponding matrix sizes of $n = 4096, 8192, 16384, 32768, 65536$, and 131072 (4%, 2%, 1%, 0.5%, 0.25%, and 0.125% nonzeroes), respectively. Flops rates from only the computation time are shown in dotted lines and explain the performance jump of the algorithms around 768 and 1,536 cores. The skewed distribution for this particular Graph500 matrix does not introduce substantial load imbalance, yielding similar performance to Erdős-Rényi's. It also increases the computation efficiency in some cases. Due to a problem with the default MKL routine we used, InnerABC in (b) was run with a different routine and shouldn't be compared to any other lines in our graphs.

Figure 5.9 compares weak scaling performance of Erdős-Rényi versus Graph500 matrices. We fix the number of nonzeroes per row to $d = 164$ and vary the number of cores from 384 to 12,288 cores. We start with 4,096 × 4,096 matrices (4% nonzeroes) on 384 cores and double the matrix size as we double the number of cores, ending with 131,072 × 131,072 (0.125% nonzeroes) matrices on 12,288 cores. Some points from Graph500 have better performance than Erdős-Rényi because of faster computation time. This might be due to the different structures of nonzeroes. Dotted lines show the weak scaling of the actual computation times of each algorithm. Due to problems running the zero-based CSR, row-major version of the MKL routine (*mkl_dcsrmm*), the data for InnerABC in Figure 5.9b were collected with one-based CSR, column-major version of the same routine which is slower, and therefore should not be compared to other algorithms in this figure.

We still observe the same performance trend for all algorithms in the Graph500 results without any significant load imbalance. ColA has the highest speedup over SummaABC at 12,288 cores, with 9.64× speedup for Erdős-Rényi matrix and 9.94× speedup for Graph500 matrix.

## Real-world Matrices

Our final experiments test on three real-world matrices of different shapes from the University of Florida Sparse Matrix Collection [56]. Each of these sparse matrices ($A$) is multiplied by




(a) Mouse gene (mouse_gene.mtx)

(b) Simplicial complexes (shar_te-b2.mtx)

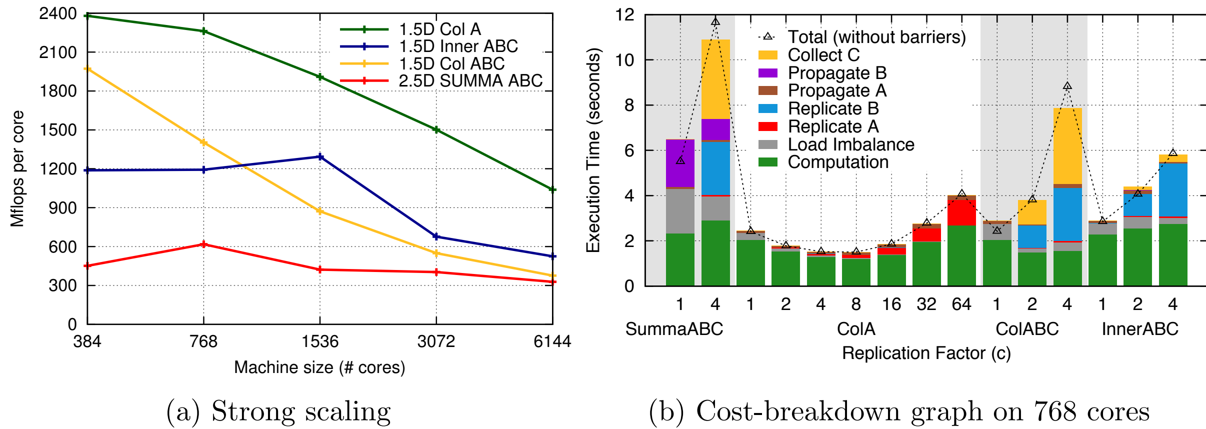
(c) Stochastic linear programming (stormg2-125.mtx)

Figure 5.10: Nonzero structures of the real-world matrices.

a generated dense matrix $B$ of the same size as $A^T$.

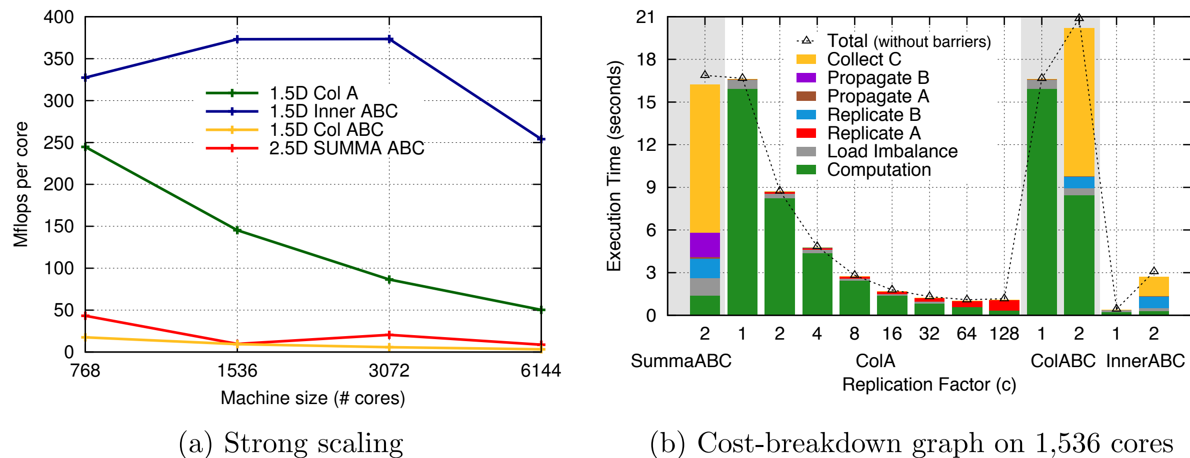
**Mouse gene network (mouse_gene.mtx)** from V. Belcastro is a square, symmetric matrix of size 45,101 $\times$ 45,101 with 28,967,291 nonzeroes (degree $d = 642.28$, 1.424% nonzeroes). Figure 5.10a shows its nonzero structure, plotted with MATLAB's spy function. The structure looks dense because the nonzeroes are scattered all over the matrix and MATLAB's smallest dot size is still too big to show the sparseness. Figure 5.11a and Figure 5.11b show its strong-scaling graph from 384 to 6,144 cores and its cost-breakdown graph on 768 cores. See Figure 5.4a at $\text{nnz}(A)/\text{nnz}(B) = 1.42\%$ for its bandwidth plot. The data for InnerABC at 3,072 and 6,144 cores are collected with one-based CSR, column-major *mkl_dcsrmm* due to the same issue mentioned in Section 5.4. Again, this one-based version is slower than our default zero-based version so these two data points are not comparable to the rest of the points in the figure. We get similar results to past experiments, except this time we see noticeably more load imbalance despite the greedy partitioning. ColA still performs best, followed by InnerABC, ColABC, and SummaABC. The maximum speedup is 5.27$\times$ from ColA over SummaABC on 384 cores.

**Simplicial complexes (shar_te2-b2.mtx)** from V. Welker is a tall matrix with dimensionality 200,200 $\times$ 17,160 with 600,600 nonzeroes (degree $d = 3$, 0.0175% nonzeroes). Figure 5.10b shows its nonzero structure. Figure 5.12a illustrates strong scaling performance from 768 to 6,144 cores. We started from 768 cores because the data were too big to fit in 16 nodes (384 cores). Figure 5.12b shows cost breakdown on 1,536 cores. The corresponding bandwidth plot is shown in Figure 5.4b. We observed mild load imbalance. The computation time for ColA and ColABC is higher than others because their local matrix shapes are tall and skinny. InnerABC has local matrices with better aspect ratio, so it performs best in this scenario. The reduction time for ColABC and SummaABC is also high because the resulting matrix $C$ is fairly large. The message sizes are very skewed: $\text{nnz}(C) \gg \text{nnz}(B) \gg \text{nnz}(A)$. The highest speedup is 38.24$\times$ at 1,536 cores, between InnerABC and SummaABC.



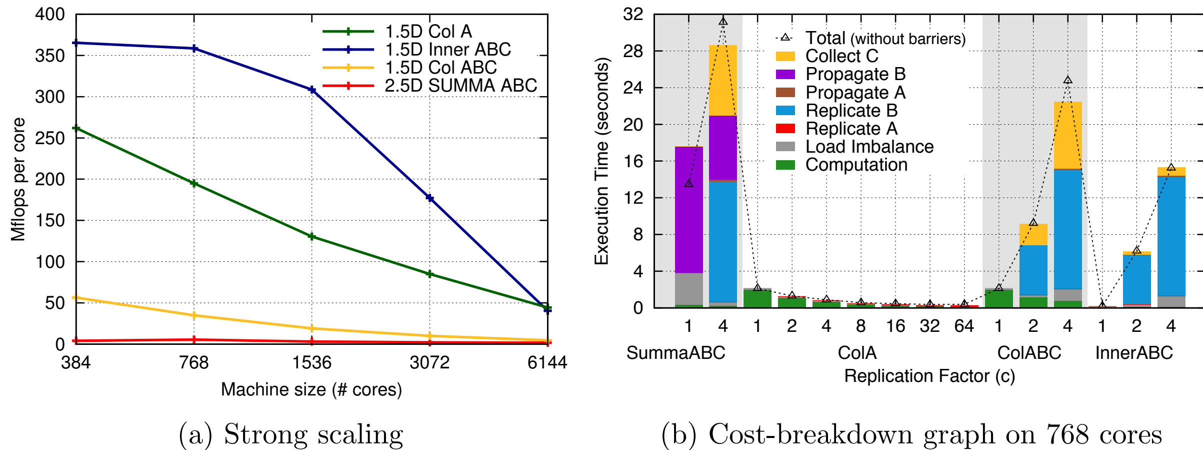

(a) Strong scaling (b) Cost-breakdown graph on 768 cores

Figure 5.11: Strong scaling and cost-breakdown results from multiplying the Mouse gene network matrix (*mouse_gene.mtx*) ($45{,}101 \times 45, 101$, 1.42% nonzeroes) with a dense $45{,}101 \times 45{,}101$ matrix. ColA performs the best again as this configuration is well in its bandwidth-winning region (a vertical line at 1.42% in Figure 5.4a from 32 to 512 MPI processes). The performance drop of InnerABC at $p$=3K and 6K cores is because these points were run with a less efficient MKL routine due to an issue with the default MKL routine we use throughout the chapter.




(a) Strong scaling (b) Cost-breakdown graph on 1,536 cores

Figure 5.12: Strong scaling and cost-breakdown results from multiplying the Simplicial complexes matrix (*shar_te2-b2.mtx*) ($200{,}200 \times 17{,}160$, 0.0175% nonzeroes) with a dense $17{,}160 \times 200{,}200$ matrix. InnerABC wins over ColA because ColA does tall-skinny local matrix multiplications which are significantly slower than InnerABC's wider local matrices.




(a) Strong scaling (b) Cost-breakdown graph on 768 cores

Figure 5.13: Strong scaling and cost-breakdown results from multiplying the Stochastic linear programming problem matrix (*stormg2-215.mtx*) (66,185×172,431, 0.0038% nonzeroes) with a dense 172,431 × 66,185 matrix.

**Stochastic linear programming problem (stormg2-125.mtx)** from C. Meszaros is a short-wide matrix of size 66,185 × 172,431 with 433,256 nonzeroes (degree $d = 6.55$, 0.0038%). Figure 5.10c shows its nonzero structure. Figure 5.13a and Figure 5.13b show its strong scaling graph from 384 to 6,144 cores and its cost breakdown graph on 768 cores. According to the bandwidth plot in Figure 5.4c, any of the 1.5D algorithms could win. Again, InnerABC wins with $c = 1$ because of faster computation time. We also observed load imbalance. The largest speedup is 99.55×, between InnerABC and SummaABC at 1,536 cores.

## 5.5 Consecutive Multiplications

The last section shows that, for a single multiplication, replicating dense matrices $B$ and $C$ usually does not pay off. The situation is different in iterative algorithms with consecutive multiplications where the input matrices $A$ and $B$ in the next iteration come from already replicated matrices. For example, if the next iteration's $A$ comes from some post-processing of the matrix $C = AB$ from the previous iteration and $B$ stays the same, the new $A$ is already replicated during the collection phase of $C$. (Or vice versa if $B$ is the changing matrix instead of $A$.) We only have to pay the costs of replicating $A$ and $B$ once at the beginning, and the costs can be amortized over all iterations. If the savings in propagation costs accumulated throughout all iterations is larger than the one-time replication costs of $A$ and $B$ and the recurring overhead of collecting $C$, then it is worth replicating the dense matrices. This section illustrates various iterative methods that can potentially benefit from using SpDM$^3$.

*Block eigenvalue problems* multiply a sparse $n \times n$ matrix $A$ with a dense $n \times k$ matrix $B$ every iteration. Usually $k < 100$ and sparse-matrix multiple-dense-vector multiplication is used. With the emergence of big data, the problem size becomes larger, and some applications [68, 176] may want up to $k > 1000$ eigenvalues, making this an SpDM$^3$ problem. *Nonnegative matrix factorization* finds a rank-$k$ factorization $WH$ of matrix $A$. Either $W$ or $H$ (or both) can be enforced to be sparse [105, 104], therefore, in each iteration there is a sparse-dense or dense-sparse $n \times k \times n$ multiplication. In *primal-dual interior-point methods* for semidefinite programming, the sparsity of the data matrix $A$ can be exploited when computing the search direction [76]. A matrix multiplication $TAU$, where $T$ and $U$ are dense, square matrices and $A$ is a sparse, square data matrix, is calculated every iteration.

Many algorithms in statistics and machine learning are iterative: A *semi-supervised classification* with graph convolutional networks [106] computes $AX$ every iteration. $A$ is a sparse $n \times n$ matrix and $X$ is a dense $n \times c$ matrix where $c$ is the number of features and is often larger than 1,000. *Attribute inference* [97, 84] multiplies a sparse, square social media graph $M$ with a dense, square posterior probability matrix $P$ every iteration. In *sparse inverse covariance matrix estimation*, the covariance matrix $S$ is square and dense, while the estimate $\Omega$ is square and enforced to be sparse. For example, CLIME-ADMM [174] computes $S\Omega$ every iteration, CONCORD-ISTA [142] computes $\Omega S$ every iteration, etc.

We will use the consecutive sparse-dense square matrix multiplications from CONCORD-ISTA [142] as our case study. The analysis framework developed in this chapter can be applied to specific scenarios, such as those in Chapter 7. Let $X$ denote a matrix of dimension $n \times \rho$, where $n$-rows are independent observations of a $\rho$-dimensional random vector. Such a matrix can represent data from various scientific disciplines including neuroscience, biology, and even social sciences. For example, $X$ may be fMRI scan data collected for $n$ time periods over $\rho$ voxels [52], expressions of $\rho$ genes from $n$ individuals [26], or voting patterns [26]. In many high dimensional datasets, dimensions of matrix $X$ is such that $n \ll \rho$, which we will assume is the case here.

The sample covariance matrix $S = X^T X$ is dense and $\rho \times \rho$. To find $\Omega$, the sparse $\rho \times \rho$ estimate, an SpDM$^3$($W = \Omega S$) is computed every iteration. The replication costs (red and blue bars in Figure 5.5) is only paid once. The propagation costs (brown and purple) and computation costs (green and gray) recur every iteration. CONCORD-ISTA uses an element-wise soft-thresholding operator which depends on the total magnitude of $g_{ij}$ where $G = W + W^T$, so a per-iteration reduction is needed for ColABC and SummaABC. ColA and InnerABC store an entire element $g_{ij}$ on a single layer so they do not need to pay collection costs in each iteration. Suppose there are $s$ iterations. Excluding the transpose, the total bandwidth costs are

$$
\begin{aligned}
W_{\text{ColA}} &= 2\frac{\text{nnz}(A)}{p} c \log c + \text{nnz}(A) s, \\
W_{\text{ColABC}} &= 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \left( \frac{\text{nnz}(A)}{c} + \frac{\text{nnz}(C)}{p} c \log c \right) s,
\end{aligned}
$$

$$W_{\text{InnerABC}} = 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \left(\frac{\text{nnz}(A)}{c}\right) s + \frac{\text{nnz}(C)}{p} c \log c,$$

$$W_{\text{SUMMA}} = 2\frac{\text{nnz}(A) + \text{nnz}(B)}{p} c \log c + \left(\frac{\text{nnz}(A) + \text{nnz}(B)}{\sqrt{pc}} \log \sqrt{\frac{p}{c}} + \frac{\text{nnz}(C)}{p} c \log c\right) s.$$

If $s$ is large enough then both the replication and collection terms of InnerABC with dense matrices can be amortized, making InnerABC more desirable since it does not move any dense matrix in propagation, does not need to collect every iteration, and it has potentially better propagation cost than ColA. Figure 5.14 illustrates the area each algorithm would have the lowest bandwidth cost for 10 and 20 iterations of multiplication. The area where InnerABC is best intuitively increases with the number of iterations. We discuss the implementation of CONCORD-ISTA in detail in Chapter 7.

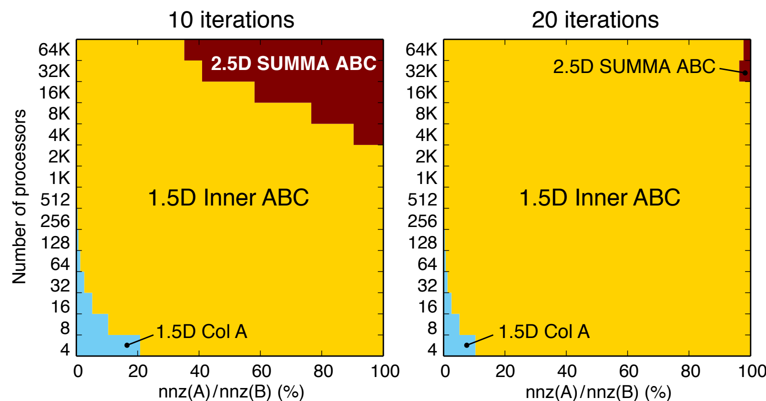



Figure 5.14: Predicting the best algorithms based on optimizing for the lowest overall bandwidth cost after 10 and 20 iterations where each point represents a given machine size and ratio of input matrix sizes. The X-axis is the ratio of nnz$(A)$ versus nnz$(B)$. The Y-axis is the number of processors.

## 5.6 Conclusions

We presented four variations on parallel sparse-dense matrix-matrix multiplication (SpDM$^3$), all based on a traditional $O(n^3)$ algorithm, but using different approaches to replicating data and partitioning work to minimize communication costs. One of these is the 2.5D SUMMA algorithm, and the other three represent new parallelization strategies specific to a setting involving sparse matrices. We derived communication lower bounds for the problem, then presented an analysis of new and existing algorithms, and compared their costs both theoretically and experimentally on over 10 thousand cores. The problem was motivated by

iterative algorithms in machine learning, and both our experiments and cost analysis break the running time into parts to show how the algorithms would compare in such a setting — some parts are one-time costs and others occur at each iteration.

Our analysis shows that no single algorithm is optimal for all settings, but that the choice depends on sparsity, matrix size, available memory, and machine size. We show that when the theoretical analysis shows a difference in cost, it is a good predictor of which algorithm to use. The theory is quite general and uses the number of matrix entries (nonzeroes) of each matrix, independent of whether the matrix is dense or sparse. Thus, while our experiment focuses on the dense-sparse case, the algorithms are relevant to other settings in which one of the two matrices is much larger or denser than another. We give guidelines on how to choose between algorithms in terms of graphs indicating what part of the input parameter space each algorithm would have the lowest bandwidth cost. The four algorithms each have benefits for some cases:

- **SummaABC** (previously known) is best with relatively dense matrices or very large processor counts. Because it moves all matrices during multiplication, it is suboptimal when one is significantly smaller or sparser.

- **ColA** is better with sparser A matrices or smaller scale parallelism. (An analogous algorithm that replicates B would work for the dense-sparse case.)

- **ColABC** and **InnerABC** generally work well in the same range as ColA, but also in the intermediate range between ColA and SummaABC in both matrix density and processor count. They have equivalent theoretical communication costs, but InnerABC is faster in practice, sometimes substantially so.

Since sparse matrices rarely have more than a few percent of nonzeroes, the majority of SpDM$^3$ will be in ColA's area, which means the best algorithm could be ColA for a particular $c \geq 1$, ColABC with $c = 1$, or InnerABC with $c = 1$. Our experimental results matched this trend. We observed up to 100× speedup of the best of our algorithms over SummaABC. Replicating ColABC or InnerABC will likely be more beneficial in consecutive multiplications rather than in single multiplication.

Our models correctly predict the trends of all communication costs and generally predict the faster algorithms and parameter settings, but they do not consider computation cost and are therefore not predictive of total runtime. In practice, MKL library performance varies when matrix shapes are different and in some cases due to different sparsity patterns, with the usual observation that larger matrices and low aspect ratio matrices run at a higher machine efficiency. This is not accounted for in our theory, but the low communication algorithms also tend to have larger local matrices, so it adds to the benefit. This omission in the model does lead to substantial mispredictions of computation time that sometimes are a deciding factor in which algorithm wins, for example, it often is a tie-breaker between ColA, ColABC at $c = 1$, and InnerABC at $c = 1$. ColA and ColABC both have faster computation time when they replicate, but ColABC also replicates the dense matrix and has to pay a

much higher cost to get to the same computation efficiency as ColA. This, combined with the cost of its reduction meant we found it to be inferior to both ColA and InnerABC. Because of this, InnerA (blocked inner product replicating only $A$) might be worth investigating as well. A future analysis should take this unequal local computation efficiency into account.

# Chapter 6

# Generalizing 1.5D Matrix Multiplication

Chapter 5 presented a set of parallelization strategies for matrix multiplication based on the relative size of the matrix and the machine size. But in many real application scenarios, there are variations on simple matrix multiplication, such as handling symmetric matrices or transposed inputs or outputs. In addition, we will want to generalize the 1.5D algorithms from Chapter 5 to allow for a set of distinct replication factors for each matrix to optimize performance. In this chapter, we will develop variations on the algorithms to address these concerns, focusing on the problems that will be necessary for the CONCORD-ISTA case study in Chapter 7.

## 6.1 Mixing Replication Factors in 1.5D Matrix Multiplication

The 1.5D algorithms in Chapter 5 calculate $C = AB$ where $A$ is sparse, $B$ and $C$ are dense, and $\text{nnz}(A) \ll \text{nnz}(B), \text{nnz}(C)$. They rotate only the *cheaper* matrix $A$ and replicate either all matrices equally or just $A$. We showed that replicating all matrices can be more beneficial for our consecutive multiplications case study. However, limiting all matrices to the same replication factor might prevent us from fully utilizing the memory. If $c$ is the largest replication factor that all three matrices can fit in memory, it is likely that we can store $A$ $c_A > c$ times and decrease more communication since $B$ and $C$ are much larger than $A$. Larger $A$'s can save computation time as well since larger/rounder matrix multiplications are more efficient in practice. (See Figure 5.5, 5.6, and 5.8 for examples.) This, along with different system latency and bandwidth characteristics and various $\text{nnz}(A) : \text{nnz}(B) : \text{nnz}(C)$ ratio, motivates us to support different replication factors for $A$, $B$, and $C$.

For completeness, we consider all possible 1D variants and give corresponding 1.5D algorithms. Each variant is a unique combination of the matrix layouts and the matrix being rotated. Table 6.1 lists all possible combinations of layouts (block row or block column) for

| Layout | | | Rotatable matrix | Algorithm name |
|---|---|---|---|---|
| A | B | C | | |
| Row | Row | Row | B | Row |
| Row | Row | Col | - | - |
| Row | Col | Row | B | Inner-B |
| Row | Col | Col | A | Inner-A |
| Col | Row | Row | - | Outer |
| Col | Row | Col | - | |
| Col | Col | Row | - | - |
| Col | Col | Col | A | Col |

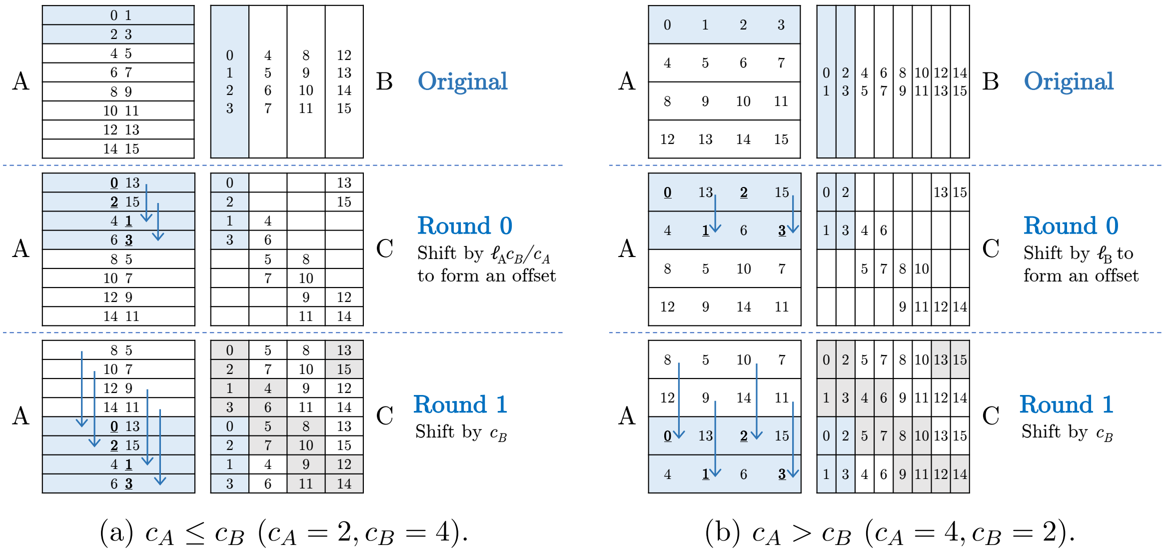
Table 6.1: List of all possible 1D algorithms. Row stands for block row layout. Col stands for block column layout. Dash (-) means the multiplication cannot be completed by rotating any of the matrices. *Outer* algorithm stores $A$ in block column layout and $B$ in block row layout but does not follow the layout for $C$ in the table. It stores the entire matrix $C$.

all three matrices. For each combination, we list the matrices ($A$, $B$, and $C$) that can be rotated to make the full multiplication possible. As the matrix being rotated dictates the layout of one matrix, each layout combination has only one rotatable matrix, none of them being $C$. A dash (-) means the combination is not realizable by rotating any matrix.

There are 5 feasible combinations: Row, Inner-B, Inner-A, Outer, and Col. The word Inner refers to $A$ having block row layout and $B$ having block column layout, forming inner products. The letter $A$ and $B$ after Inner refers to the matrix being rotated, which controls the layout of $C$: Inner-A has $C$ in block column layout and Inner-B has $C$ in block row layout. Outer has $A$ and $B$ in block column and block row layout, forming outer products. In the Outer algorithm, every processor holds an entire copy of $C$, so the layout is neither block row or block column. Our multiple-replication-factor scheme works with Row, Inner-B, Inner-A, and Col, but does not work with Outer. We will discuss the first four variants in details, describe Outer briefly, and analyze communication costs at the end.

The high-level approach remains: (1) we divide processors into teams, (2) each team owns particular parts of $A$, $B$, and $C$, and (3) all team members work together to compute the part of $C$ that they own. Many algorithm properties depend on what matrix is being shifted, so from now on we will call the rotating operand (either $A$ or $B$) $R$ and the fixed operand ($B$ or $A$) $F$. Let $c_R$ and $c_F$ be the replication factors of $R$ and $F$. We introduce two separate processor meshes, $P_R$ and $P_F$. $P_R$ is of size $p/c_R \times c_R$ and partitions $R$ into $p/c_R$ equal parts, with $R(i_R)$ on $P_R(i_R, :)$. Similarly, $P_F$ is of size $p/c_F \times c_F$ and partitions $F$ into $p/c_F$ equal parts, with $F(i_F)$ on $P_F(i_F, :)$. A processor $P(i)$ is both $P_R(i_R, \ell_R)$ and $P_F(i_F, \ell_F)$. The mapping for any processor mesh $P_X$ is $i_X = \lfloor i/c_X \rfloor$ and $\ell_X = i \mod c_X$. The replication factor and layout of $C$ is the same as those of $F$'s. See Figure 6.1 and 6.2 for illustrations.

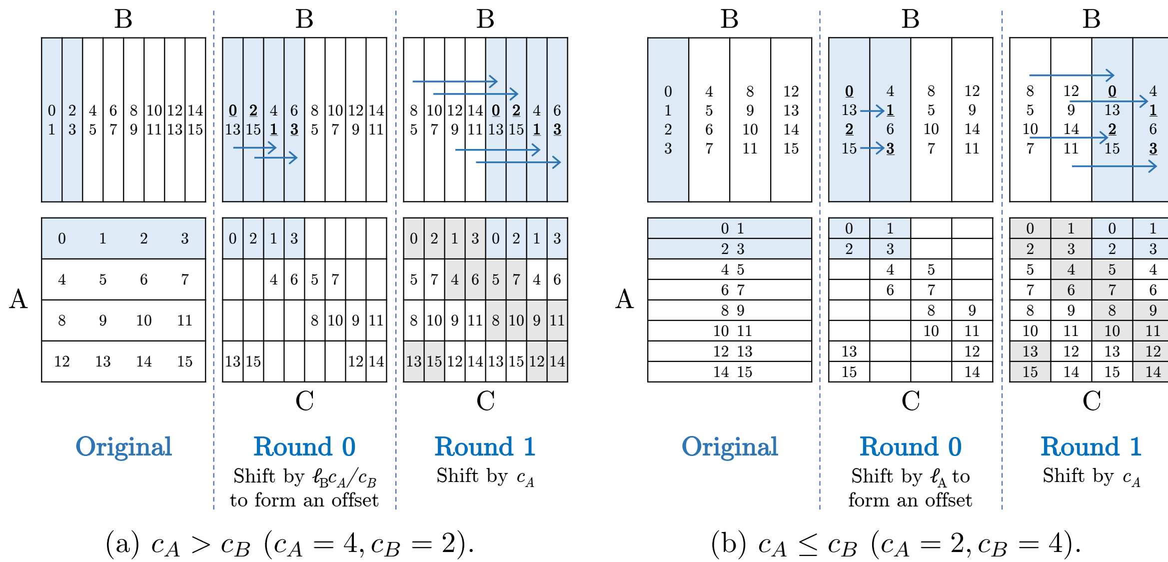
Processors $P_F(i_F, :)$ cooperate on multiplying all $p/c_R$ blocks of the rotating matrix $R$ with $F(i_F)$ that they own. Instead of giving each team member consecutive blocks of $R$ (see Figure 5.3b and 5.3c for examples), we make team members calculate interleaved blocks




(a) $c_A \leq c_B$ $(c_A = 2, c_B = 4)$.

(b) $c_A > c_B$ $(c_A = 4, c_B = 2)$.

Figure 6.1: **Inner-A.** Processors $P_B(i_B, :)$ compute $C(i_B)$ together. The numbers on the matrix parts are the ranks of the processors that they reside on. Here, $p = 16$ and $(c_A, c_B)$ is $(2, 4)$ in (a) and $(4, 2)$ in (b). The first line shows the original layouts of $A$ and $B$. The second line (Round 0) shifts $A$ by $\delta$ to compute the first $c_B$ blocks of $C(i_B)$. The third line (Round 1) shifts $A$ by $c_B$ and computes the rest of $C$.




(a) $c_A > c_B$ $(c_A = 4, c_B = 2)$.

(b) $c_A \leq c_B$ $(c_A = 2, c_B = 4)$.

Figure 6.2: **Inner-B.** Processors $P_A(i_A, :)$ compute $C(i_A)$ together. The numbers on the matrix parts are the ranks of the processors that they reside on. Here, $p = 16$ and $(c_A, c_B)$ is $(4, 2)$ in (a) and $(2, 4)$ in (b). The Original panel shows the original layouts of $A$ and $B$. Round 0 shifts $B$ by $\delta$ to compute the first $c_A$ blocks of $C(i_A, :)$. Round 1 shifts $B$ by $c_A$ and computes the rest of $C$.

**Algorithm 14** 1.5D Multiple-Replication-Factor Matrix Multiplication

**Input:** $P_R$ and $P_F$, 2D processor meshes of sizes $p/c_R \times c_R$ and $p/c_F \times c_F$.
**Input:** $R$, partitioned into $p/c_R$ equal parts and distributed so $R(i_R)$ is in buffer $\tilde{R}$ on $P_R(i_F, :)$.
**Input:** $F$, partitioned into $p/c_F$ equal parts and distributed so $F(i_F)$ is in buffer $\tilde{F}$ on $P_F(i_F, :)$.
**Output:** $C = AB$, partitioned into $p/c_F$ equal parts and distributed so $C(i_F)$ is in buffer $\tilde{C}$ on $P_F(i_F, :)$.

1: **for each** $P_R(i_R, \ell_R) = P_F(i_F, \ell_F)$, in parallel, **do**
2: $\quad \delta \leftarrow \min(\ell_F, \ell_R) \cdot \max(c_F/c_R, 1)$
3: $\quad$ Shift $\tilde{R}$ by $\delta$.
4: $\quad$ **for** $\frac{p}{c_F c_R}$ rounds **do**
5: $\quad\quad$ Multiply $\tilde{R}$ and $\tilde{F}$ and store/accumulate the result in $\tilde{C}$.
6: $\quad\quad$ Shift $\tilde{R}$ by $c_F$.
7: $\quad$ **end for**
8: $\quad$ Sumreduce/allgather $C$ between $P_F(j, :)$.
9: **end for**

instead. This simplifies transposing the resulting matrix $C$, which we will explain in Section 6.2. In the first round, team members are assigned work in a way that covers the first $c_F$ blocks of $R$. In the next round, everyone moves to work on the next block that is $c_F$ blocks away and covers the next $c_F$ blocks of $R$. The process continues until all $p/c_R$ blocks is covered and takes $p/(c_R c_F)$ rounds in total.

The final $C$ is collected from all team members by either a sumreduce (if using 1.5D Row or 1.5D Col) or a gather (if using 1.5D Inner-A or 1.5D Inner-B). We show the pseudocode in Algorithm 14. For completeness, we also give exact computation details of each variant below. Table 6.2 compares the communication costs of all variants.

**1.5D Row.** All matrices are in block row layout. Matrix $B$ is shifted to form $C$ in a block row layout. $P_A(i_A, :)$ has $A^{\{(p/c_A)\times 1\}}(i_A)$ and $C^{\{(p/c_A)\times 1\}}(i_A)$. $P_B(i_B, :)$ has $B^{\{(p/c_B)\times 1\}}(i_B)$. $P_A(i_A, \ell_A) = P_B(i_B, \ell_B)$ computes,

$$C^{\left\{\frac{p}{c_A}\times 1\right\}}(i_A) = \sum_{k=0}^{\frac{p}{c_A c_B}-1} A^{\left\{\frac{p}{c_A}\times\frac{p}{c_B}\right\}}(i_A, i_B + \delta + kc_A) \cdot B^{\left\{\frac{p}{c_B}\times 1\right\}}(i_B + \delta + kc_A),$$

where $\delta = \min(\ell_A, \ell_B)\max(c_A/c_B, 1)$, $k$ is the round number, and $p/(c_A c_B)$ is the total number of rounds.

**1.5D Col.** All matrices are in block column layout. Matrix $A$ is shifted to form $C$ in a block column layout. $P_A(i_A, :)$ has $A^{\{(p/c_A)\times 1\}}(i_A)$. $P_B(i_B, :)$ has $B^{\{(p/c_B)\times 1\}}(i_B)$ and $C^{\{(p/c_B)\times 1\}}(i_B)$. $P_A(i_A, \ell_A) = P_B(i_B, \ell_B)$ computes,

$$C^{\left\{1\times\frac{p}{c_B}\right\}}(i_B) = \sum_{k=0}^{\frac{p}{c_A c_B}-1} A^{\left\{1\times\frac{p}{c_A}\right\}}(i_A + \delta + kc_B) \cdot B^{\left\{\frac{p}{c_A}\times\frac{p}{c_B}\right\}}(i_A + \delta + kc_B, i_B),$$

| Algorithms | #messages = $S$ | | | #words = $W$ | | |
|---|---|---|---|---|---|---|
| | Replication | Propagation | Collection | Replication | Propagation | Collection |
| 1.5D General | $2\log c_A + 2\log c_B$ | $\frac{p}{c_A c_B}$ | | $2\frac{\text{nnz}(A)}{p}c_A\log c_A + 2\frac{\text{nnz}(B)}{p}c_B\log c_B$ | | |
| | | | $\log c_F$ | | $\frac{\text{nnz}(R)}{c_F}$ | $\frac{\text{nnz}(C)}{p}c_F\log c_F$ |
| 1.5D Col<br>1.5D Inner-A | | | $\log c_B$ | | $\frac{\text{nnz}(A)}{c_B}$ | $\frac{\text{nnz}(C)}{p}c_B\log c_B$ |
| 1.5D Inner-B<br>1.5D Row | | | $\log c_A$ | | $\frac{\text{nnz}(B)}{c_A}$ | $\frac{\text{nnz}(C)}{p}c_A\log c_A$ |
| 1D Outer | - | - | $\log p$ | - | - | $\text{nnz}(C)\log p$ |

Table 6.2: Algorithm communication costs. 1.5D General shows the costs that are common to all of the matrix layouts shown in the 4 lines below, regardless of the matrix layouts. Some costs depend on which matrix is being fixed ($F$) and which matrix is being rotated ($R$). $A$ or $B$ after Inner refers to the rotating matrix. 1.5D Col and Inner-A both rotate $A$, so they share the rotating-specific costs. 1.5D Inner-B and 1.5D Row both rotate $B$ and share the rotating-specific costs. 1D Outer is a separate algorithm and does not share the costs with any other algorithms. The algorithms have three phases: replication, propagation, and collection. See the Comparison subsection in Section 5.3 for their definitions.

where $\delta = \min(\ell_B, \ell_A)\max(c_B/c_A, 1)$, $k$ is the round number, and $p/(c_A c_B)$ is the total number of rounds.

**1.5D Inner-A.** $A$ is in block row layout. $B$ is in block column layout. $A$ is shifted to form $C$ in a block column layout. $P_A(i_A, :)$ has $A^{\{(p/c_A)\times 1\}}(i_A)$. $P_B(i_B, :)$ has $B^{\{1\times(p/c_B)\}}(i_B)$ and $C^{\{1\times(p/c_B)\}}(i_B)$. In round $k$, $P_A(i_A, \ell_A) = P_B(i_B, \ell_B)$ computes,

$$C^{\left\{\frac{p}{c_A}\times\frac{p}{c_B}\right\}}(i_A+\delta+kc_B, i_B) = A^{\left\{\frac{p}{c_A}\times 1\right\}}(i_A+\delta+kc_B)\cdot B^{\left\{1\times\frac{p}{c_B}\right\}}(i_A+\delta+kc_B),$$

where $\delta = \min(\ell_B, \ell_A)\max\left(\frac{c_B}{c_A}, 1\right)$, for $0 \le k < p/(c_A c_B)$.

**1.5D Inner-B.** $A$ is in block row layout. $B$ is in block column layout. $B$ is shifted to form $C$ in a block row layout. $P_A(i_A, :)$ has $A^{\{(p/c_A)\times 1\}}(i_A)$ and $C^{\{(p/c_A)\times 1\}}(i_A)$. $P_B(i_B, :)$ has $B^{\{1\times(p/c_B)\}}(i_B)$. In round $k$, $P_A(i_A, \ell_A) = P_B(i_B, \ell_B)$ computes,

$$C^{\left\{\frac{p}{c_A}\times\frac{p}{c_B}\right\}}(i_A, i_B+\delta+kc_A) = A^{\left\{\frac{p}{c_A}\times 1\right\}}(i_B+\delta+kc_A)\cdot B^{\left\{1\times\frac{p}{c_B}\right\}}(i_B+\delta+kc_A),$$

where $\delta = \min(\ell_A, \ell_B)\max\left(\frac{c_A}{c_B}, 1\right)$, for $0 \le k < p/(c_A c_B)$.

**1D Outer.** $A$ is in block column layout, and $B$ is in block row layout. Each processor calculates one outer product and must store the entire $C$, so $C$ cannot be too large here. There is no need to replicate $A$ or $B$. The processor mesh $P$ is 1-dimensional. $P(i)$ has $A^{\{1\times p\}}(i), B^{\{p\times 1\}}(i)$, and $C$. There is only one round, $P(i)$ calculates

$$C = A^{\{1\times p\}}(i)B^{\{p\times 1\}}(i).$$

$P(0)$ then sumreduces the matrix $C$ from all processors to get the final result

$$C = \sum_{k=0}^{p-1} A^{\{1\times p\}}(k)B^{\{p\times 1\}}(k).$$

### Cost analysis

General analysis in Section 5.3 still applies. There are a few new observations with the introduction of two replication factors.

**Replication factor.** The new limit for effective replication factors is $c_A c_B \leq p$, where the algorithm has only one round to complete. When $c_A = p$ and $c_B = 1$, this corresponds to fully replicating $A$ and not replicating $B$ at all. $c_B = 1$ with arbitrary $c_A$ gives us ColA in Chapter 5. $c_A = c_B = \sqrt{P}$ gives us ColABC and InnerABC, also from Chapter 5. Using $c_A = 1$ and $c_B = p$ fully replicates $B$ and does not replicate $A$. The new multiple-replication-factor algorithms allow us a much larger search space.

**Latency.** Latency costs do not depend on the matrix inputs at all. They only depend on $p$, $c_A$, and $c_B$. The choice of the rotating matrix affects latency, but just slightly: only the collection cost changes, and the change is rather small since $c_F$ is in the log term.

**Bandwidth.** Bandwidth costs depend greatly on the sizes and the replication factors of the matrices. The propagation cost depends on the size of the rotating matrix ($\text{nnz}(R)$) and the replication factor of the fixed matrix ($c_F$). If, for layout reasons, we cannot rotate $A$ (to get $\text{nnz}(A)/c_B$ propagation cost), we can still avoid moving too much of $B$ by replicating $A$ as much as possible and get the propagation cost $\text{nnz}(B)/c_A$. The collection cost depends on $c_F$ since $C$ has the same layout as $F$.

## 6.2 Matrix Transpose

This section shows how to transpose a resulting matrix from Inner-A to compute $AB+(AB)^T$. This trivially applies to Inner-B as well. There are two reasons we switched from each team member processing contiguous blocks of $R$ (shift by 1), as shown in ColABC and Inner-ABC in Chapter 5, to them processing every ${c_F}^{\text{th}}$block of $R$ (shift by $c_F$):

- Extending it to support different replication factors is nontrivial when $c_R < c_F$. See Figure 6.1a and 6.2a for examples. Processors 0 and 2 own consecutive blocks of $R$. Since they are on the same layer, they will always hold consecutive blocks of $R$ because shifting moves all blocks on the same layer by the same distance.

- The layout of the resulting matrix $C$ involves all processors in a transpose. Figure 6.3 shows the layout of $C$ from an InnerABC multiplication with $p = 27$ processors and $c = 3$ layers. Each processor has $p/c^2 = 3$ subblocks of $C$ and needs to get their corresponding transposes. Even though everyone only needs to talk to at most $p/c^2$ processors, the overall communication graph spread over all processors.

Shifting by $c_F$ gives us repeated patterns in $C$'s layout and allows us to limit communication within closed, smaller groups, as shown in Figure 6.4: processors in the sets {8, 12, 40, 44} and {10, 14, 26, 30, 42, 46, 58, 62} only need to communicate within their sets to get their transposes. We call these sets of processors *transpose groups*. We will show that this is a property of the algorithm.

Let $\mathcal{L}$ be a layout matrix of $C$. $\mathcal{L}$ has $p/c_A$ rows and $p/c_B$ columns. Each $l_{ij}$ is the rank of the processor that has $C^{\{p/c_A \times p/c_B\}}(i,j)$. We partition $\mathcal{L}$ into $d \times d$ parts where $d = p/(c_A c_B)$ is the number of rounds. First, we show in Lemma 6.2 that each block column of $\mathcal{L}^{\{d\times d\}}$ is just the top block of that column stacked together $d$ times. Lemma 6.1 states that all the ranks that appear in $\mathcal{L}$ repeat every $c_b$ rows.

**Lemma 6.1.** *$l_{mn} = l_{m+kc_B,n}$ for any $0 \le k < d$, $0 \le m < p/c_A$, and $0 \le n < p/c_B$.*

*Proof.* By construction of Inner-A, each processor $P(i) = P_A(i_A, \ell_A) = P_B(i_B, \ell_B)$ calculates $C^{\{p/c_A \times p/c_B\}}(i_A{+}\delta{+}kc_B, i_B)$ where $0 \le k < d$ is the round number. This sets $l_{i_A+\delta+kc_B,i_B} = i$. Since each $0 \le \ell_B < c_B$ has $(i_A{+}\delta) \mod c_B$ mapped uniquely to the range $[0, c_B)$, $i_A{+}\delta{+}kc_B$ has a one-to-one mapping to $[0, p/c_A)$. This means $l_{m,i_B} = l_{m+kc_B,i_B}$ for any $0 \le m < p/c_A$. Because $0 \le i_B < p/c_B$, the proof is complete. □

**Lemma 6.2.** *$\mathcal{L}^{\{d\times d\}}(i,j) = \mathcal{L}^{\{d\times d\}}(0,j)$ for all $0 \le i,j < d$.*

*Proof.*

$$\mathcal{L}^{\{d\times d\}}(0,j) = \begin{bmatrix} l_{0,n} & l_{0,n+1} & \cdots & l_{0,n+c_A-1} \\ l_{1,n} & l_{1,n+1} & \cdots & l_{1,n+c_A-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_{c_B-1,n} & l_{c_B-1,n+1} & \cdots & l_{c_B-1,n+c_A-1} \end{bmatrix},$$

for any $0 \le j < d$ where $n = jc_A$. Applying Lemma 6.1 to all elements of $\mathcal{L}^{\{d\times d\}}(0,j)$, we have,

$$\mathcal{L}^{\{d\times d\}}(0,j) = \begin{bmatrix} l_{0+ic_A,n} & l_{0+ic_A,n+1} & \cdots & l_{0+ic_A,n+c_A-1} \\ l_{1+ic_A,n} & l_{1+ic_A,n+1} & \cdots & l_{1+ic_A,n+c_A-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_{c_B-1+ic_A,n} & l_{c_B-1+ic_A,n+1} & \cdots & l_{c_B-1+ic_A,n+c_A-1} \end{bmatrix} = \mathcal{L}^{\{d\times d\}}(i,j),$$

for any $0 \le i < d$. □

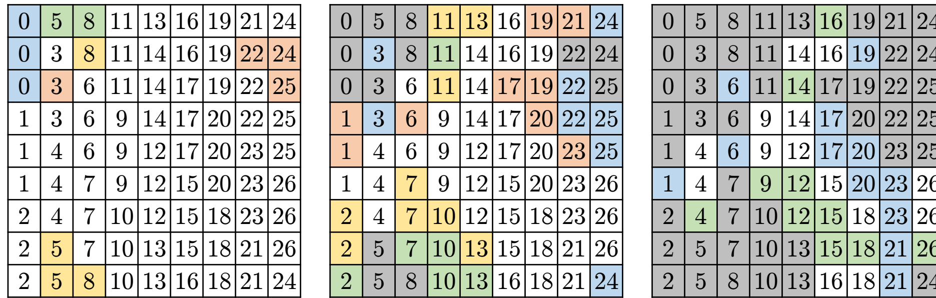
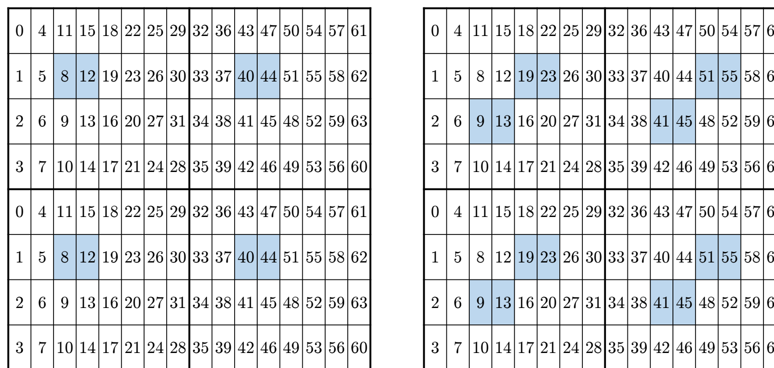


Figure 6.3: The transpose of matrix $C = AB$ has a communication graph that spans all processors if $C$ is computed by Inner-ABC from chapter 5 (shift by 1). All three images show the layout of the resulting matrix $C$ after the multiplication ($p = 27$ processors and $c = 3$ copies). The number in each submatrix of $C$ is the rank of the processor that it resides on. Starting from the left image, processor 0 (blue cells) needs the green cells from processors 5 and 8. The rest of 5 and 8's cells in yellow needs the orange parts from 3, 22, 24, and 25. Middle and right images show subsequent communication in the same color order: blue, green, yellow, and orange, until all processors are involved at the end.



(a) On diagonal. (b) Off diagonal.

Figure 6.4: The transpose of matrix $C = AB$ resulting from Inner-A (shift by $c_B$) can be done within disjointed subsets of processors. Both images show the layout $\mathcal{L}$ of $C$ after Inner-A with $p = 64$, $c_A = 8$, and $c_B = 4$. The layout matrices $\mathcal{L}^{\{2\times2\}}(0,0)$and $\mathcal{L}^{\{2\times2\}}(0,1)$ are of size $4 \times 8$ and have thick border around. As Inner-A shifts $A$ by $c_B$, these patterns repeat every $c_B = 4$ block rows and we can write the overall layout of $C$ as $[\mathcal{L}^{\{2\times2\}}(0,0)\ \mathcal{L}^{\{2\times2\}}(0,1);\ \mathcal{L}^{\{2\times2\}}(0,0)\ \mathcal{L}^{\{2\times2\}}(0,1)]$ where ; indicates the beginning of a new block row. $T$ is a sparse, transpose pattern matrix of size as $4 \times 4$ with nonzeroes only in areas shaded in blue. $T$ serves as a mask and is put on top of each $\mathcal{L}^{\{2\times2\}}(i,j)$. Processors in the shaded area only need to communicate with each other and no other processors outside the mask to complete their transposes. Figures (a) and (b) show two different $T$'s: (a) If $T$ is shaded on the diagonal, the size of communication group is $p/(c_Ac_B)\cdot\max(c_A/c_B, c_B/c_A)$. (b) The size is twice as large when the shaded boxes are off-diagonal.

To see the parts of the matrix each processor would need for its transpose operation, we define a *transpose mask* $T$. $T$ is a sparse, symmetric matrix of size $c_{\min} \times c_{\min}$ where $c_{\min} = \min(c_A, c_B)$. We overlay $T$ over a square submatrix that each $\mathcal{L}^{\{d\times d\}}(i,j)$ span to indicate selection of matrix blocks of $C$. For example, the top left quadrant of Figure 6.4b has $T$of size $4 \times 4$ with only $t_{21}$ and $t_{12}$ set (highlighted in blue), overlaying over $\mathcal{L}^{\{d\times d\}}(0,0)$ and selecting $l_{22}, l_{23}, l_{14}$, and $l_{15}$ (containing ranks 9, 13, 19, and 23). Let $\mathcal{T}$ be a big mask composed of $d \times d$ blocks of $T$, spanning over the entire $\mathcal{L}$. Lemma 6.3 shows that $\mathcal{T}$ is symmetric.

**Lemma 6.3.** *A matrix $M$ constructed from $n \times n$ blocks of a square symmetric matrix $S$,*

$$M = \begin{bmatrix} S & S & \cdots & S \\ S & S & \cdots & S \\ \vdots & \vdots & \ddots & \vdots \\ S & S & \cdots & S \end{bmatrix},$$

*is symmetric.*

*Proof.* We prove by induction on $n$. The base case $n = 1$ is true since $M = S$ which is symmetric. Assuming the lemma is true when $n = k$, we show that it is also true for $n = k+1$: $M^{\{(k+1)\times(k+1)\}}(0:k, 0:k)$ is symmetric, $M^{\{(k+1)\times(k+1)\}}(k,j) = M^{\{(k+1)\times(k+1)\}}(j,k)^T = S$ for all $0 \le j < k+1$, and $M^{\{(k+1)\times(k+1)\}}(k,k) = S$ is symmetric. Therefore, $M$ is symmetric. □

If we can show that, for any processor rank that is selected by $\mathcal{T}$, all of its occurrences in $\mathcal{L}$ are also selected, then $\mathcal{T}$ is a *transpose group*. For example, the blue cells in Figure 6.4b is a *transpose group* because all occurrences of all 8 processor ranks within it are covered.

**Theorem 6.4.** *Inner-A (shift by $c_B$) has*

- *$c_{min}$ transpose groups with $\frac{p}{c_A c_B} \max\left(\frac{c_A}{c_B}, \frac{c_B}{c_A}\right)$ members, and*
- *$\binom{c_{min}}{2}$ transpose groups with $2\frac{p}{c_A c_B} \max\left(\frac{c_A}{c_B}, \frac{c_B}{c_A}\right)$ members,*

*Proof.* Since we apply the same mask $T$ to all blocks of $\mathcal{L}^{\{d\times d\}}$, Lemma 6.1 and 6.2 guarantee that $\mathcal{T}$ covers all occurrences of all ranks selected and that it is a transpose group. Furthermore, the number of processors in the group is just the number of $l_ij$s' that $T$ spans over on $\mathcal{L}^{\{d\times d\}}(0,:)$, since the rest block rows repeat the same pattern.

To count the number of transpose groups the algorithm has, we simply consider how each element $l_{ij}$ in $\mathcal{L}^{\{d\times d\}}(0,0)$ can be selected (because the same pattern $T$ is applied to all other blocks). Each $l_{mn}$ can either be selected with $t_{ii}$ or $t_{ij}$ where $i \neq j$.

- $\mathbf{t_{ii}}$ : $l_{mn}$ lies on the diagonal of $T$. $T$ has just one element set: $t_{ii}$. $T$ is of size $c_{\min} \times c_{\min}$, so there can be $c_{\min}$ different transpose groups like this. Each $t_{ii}$ spans over $\max(c_A/c_B, c_B/c_A)$ ranks. There are $d = p/(c_A c_B)$ different blocks, therefore, the number of processors in the group is $\frac{p}{c_A c_B} \max\left(\frac{c_A}{c_B}, \frac{c_B}{c_A}\right)$.

- $\mathbf{t_{ij}}$ : $l_{mn}$ lies off the diagonal of $T$. Two elements, $t_{ij}$ and $t_{ji}$, have to be set to maintain the symmetry. There are $\binom{c_{\min}}{2}$ ways to choose different pairs of indices $i$ and $j$, and this is the number of transpose groups of this type. $t_ij$ and $t_ji$ each spans over $\max(c_A/c_B, c_B/c_A)$ different ranks. There are $d = p/(c_A c_B)$ different blocks, therefore, the number of processors in the group is $2\frac{p}{c_A c_B}\max\left(\frac{c_A}{c_B}, \frac{c_B}{c_A}\right)$.

□

### Communication Costs

There are at most

$$2\frac{p}{c_A c_B}\max\left(\frac{c_A}{c_B}, \frac{c_B}{c_A}\right) = 2\max\left(\frac{p}{c_A^2}, \frac{p}{c_B^2}\right)$$

processors in a transpose group. Each processor only holds $\mathrm{nnz}(C)/(p/c_B)$ words of $C$, split into $p/(c_A)$ blocks. Each block therefore consists of $\mathrm{nnz}(C)c_A c_B/p^2$ words. To do the transpose, each processor exchanges a block with all other processors in an all-to-all manner. According to Table 2.2, the costs are,

$$\begin{aligned} S_{\text{transpose}} &= O\left(\log\left(2\max\left(\frac{p}{c_A^2}, \frac{p}{c_B^2}\right)\right)\right) \\ &= O\left(\log\max\left(\frac{p}{c_A^2}, \frac{p}{c_B^2}\right)\right), \\ W_{\text{transpose}} &= O\left(\frac{\mathrm{nnz}(C)c_A c_B}{p^2}\max\left(\frac{p}{c_A^2}, \frac{p}{c_B^2}\right)\log\left(\max\left(\frac{p}{c_A^2}, \frac{p}{c_B^2}\right)\right)\right). \end{aligned} \tag{6.1}$$

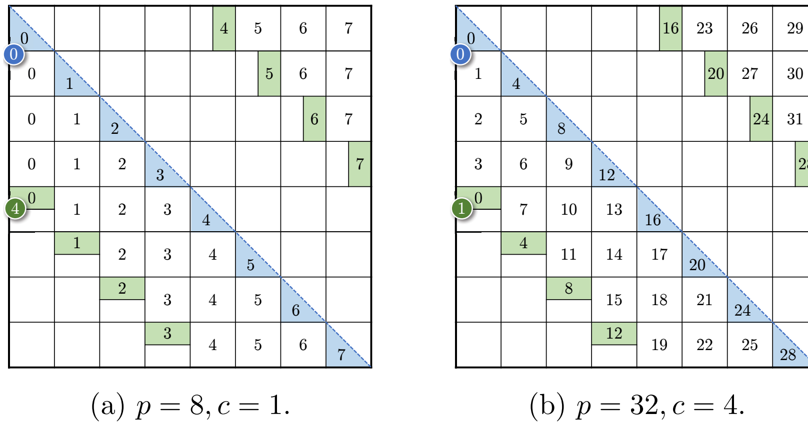
Let $c_{\min} = \min(c_A, c_B)$ and $c_{\max} = \max(c_A, c_B)$, then the costs in Equation (6.1) are simplified to,

$$\begin{aligned} S_{\text{transpose}} &= O\left(\log\left(\frac{p}{c_{\min}^2}\right)\right), \\ W_{\text{transpose}} &= O\left(\frac{\mathrm{nnz}(C)}{p^2}\cdot\frac{c_{\max}}{c_{\min}}\cdot\log\left(\frac{p}{c_{\min}^2}\right)\right). \end{aligned} \tag{6.2}$$

The best replication factors for both bandwidth and latency costs are $c_A = c_B = \sqrt{p}$, regardless of the matrix inputs.

## 6.3 Symmetric Matrix Multiplication

This section focuses on the multiplication of $S = X^T X$ where $X$ is short and wide ($n \times \rho$ where $n \ll \rho$). Since $X$ and $X^T$ have the same number of nonzeroes, there is no benefit in moving just one matrix, but the 1.5D algorithms could still be advantageous over 3D algorithms




(a) $p = 8, c = 1$. (b) $p = 32, c = 4$.

Figure 6.5: The layout of $S$ after running the modified Inner-A for symmetry. The numbers inside each blocks of $S$ are the processor ranks the blocks reside on. The numbers in circles are the round numbers. The blue parts show the $0^{\text{th}}$ round where each team leader only computes half a block. The green parts show the additional round that only team leaders need to do, computing half a block each again.

because of the matrix shapes: partitioning $X^T$ in block row and $X$ in block column layout gives us *rounder* local matrix multiplications which can be done more efficiently. We discuss the algorithm for the same replication factor for both $X$ and $X^T$ because $\text{nnz}(X) = \text{nnz}(X^T)$ and different replication factors would only yield inferior performance in both latency and bandwidth.

The changes we need to support symmetry are straightforward. First, we halve the number of rounds from $p/c^2$ to $(1/2 \cdot p/c^2)$. Second, we make the team leaders compute local symmetric multiplication (only half a block) in the first round. Lastly, we made the team leaders do one additional round, but also computing half a block each. Figure 6.5 illustrates the changes.

The full algorithm is shown in Algorithm 15. The processor mesh $P$ is of size $p/c \times c$. $P(i,:)$ has $A^{\{p/c \times 1\}}(i)$, $B^{\{1 \times p/c\}}$, and $C^{\{1 \times p/c\}}$. In round $k$, a processor $P(i, \ell)$ (except $P(i, 0)$ in round 0) computes,

$$C^{\{p/c \times p/c\}}(i + \ell + kc, i) = A^{\{p/c \times 1\}}(i + \ell + kc) \cdot B^{\{1 \times p/c\}}(i).$$

In round 0, processor $P(i, 0)$ computes,

$$C^{\{2p/c \times p/c\}}(2i + 1, i) = A^{\{2p/c \times 1\}}(2i + 1) \cdot B^{\{1 \times p/c\}}(i).$$

After $1/2 \cdot p/c^2$ rounds, only $P(i, 0)$ compute one extra round. If $0 \leq i < p/(2c)$, $P(i, 0)$ computes,

$$C^{\{2p/c \times p/c\}}(2i + p/c^2, i) = A^{\{2p/c \times 1\}}(2i + p/c^2) \cdot B^{\{1 \times p/c\}}(i).$$

Otherwise, $P(i, 0)$ computes,

$$C^{\{p/c \times 2p/c\}}(i + p/(2c^2), 2i + 1) = A^{\{p/c \times 1\}}(i + p/(2c^2)) \cdot B^{\{1 \times 2p/c\}}(2i + 1).$$

**Algorithm 15** Symmetric 1.5D Inner-A

**Input:** $P$, a 2-dimensional processor mesh of size $p/c \times c$.
**Input:** $A$, distributed so $A^{\{p/c\times 1\}}(i)$ is in buffer $\tilde{A}$ on $P(i,:)$.
**Input:** $B$, distributed so $B^{\{1\times p/c\}}(i)$ is in buffer $\tilde{B}$ on $P(i,:)$.
**Output:** $C = AB$, distributed so $C^{\{1\times p/c\}}(i)$ is in buffer $\tilde{C}$ on $P(i,:)$.

```
1: for each P(i,ℓ), in parallel, do
2:     Shift Ã by ℓ.
3:     for k ∈ {0,1,...,p/(2c^2) − 1} do
4:         if k = 0 and ℓ = 0 then
5:             C̃^{p/c×1}(i) = symmetric_multiply(Ã, B̃).
6:         else
7:             C̃^{p/c×1}(i + ℓ + kc) = ÃB̃.
8:         end if
9:         Shift Ã by c.
10:    end for
11:    if ℓ = 0 and i < p/(2c) then
12:        C̃^{2p/c×1}(2i + p/c) = Ã^{2×1}(0)B̃.
13:    else if ℓ = 0 and i ≥ p/(2c) then
14:        C̃^{p/c×2}(i + p/(2c), 1) = ÃB̃^{1×2}(1).
15:    end if
16:    Allgather C.
17: end for
```

Despite one extra round for the team leaders, all processors still do equal work. Team leaders send one message more than other team members, but this does not matter in asymptotic costs. All flops, messages, and words count (during propagation phase) are halved. The maximum effective $c$ changes to $\sqrt{p/2}$ instead of $\sqrt{p}$.

# Chapter 7

# Sparse Inverse Covariance Matrix Estimation

This chapter presents our case study of consecutive multiplications mentioned in Section 5.5, using our 1.5D multiple-replication-factor matrix multiplication algorithms in Chapter 6. The chapter is organized as follows. Section 7.1 introduces the problem. Section 7.2 describes our parallel algorithm, HP-CONCORD. Section 7.3 gives experimental results and makes comparisons with the leading penalized Gaussian likelihood approach. Section 7.4 presents a detailed empirical study, where we apply HP-CONCORD to high-dimensional functional magnetic resonance (fMRI) data. Section 7.5 gives an expanded set of results for Section 7.4. Section 7.6 discusses further directions and concludes.

## 7.1 Background and Previous Work

Characterizing complex relationships in high-dimensional data is an important research problem in many disciplines including biology, economics, environmental sciences, and neuroscience. Examples of interesting relationships in these applications include associations between genes, financial institutions, temperature measurements, and regions of the brain.

Suppose we want to reconstruct the underlying relationships between variables from samples. Let $X \in \mathbb{R}^{n \times \rho}$ be a data matrix, consisting of $n$ independent observations of a $\rho$-dimensional random vector with mean zero and variance-covariance matrix $\Sigma^\star$. (Statistical and machine learning literature often represents the number of dimensions by $p$. Here, we use $\rho$ to avoid confusion with our $p$ for the number of processors.) It can be shown that the matrix $\Omega^\star = (\Sigma^\star)^{-1}$ encodes the conditional pairwise dependencies between the variables;

This chapter is based on joint work previously published in "Communication-Avoiding Optimization Methods for Massive-Scale Graphical Model Structure Learning"[110].

specifically, the partial correlations are just scaled versions of the elements in $\Omega^\star$, i.e.,

$$r_{ij\cdot V\backslash\{i,j\}} = -\frac{\omega_{ij}}{\sqrt{\omega_{ii}\,\omega_{jj}}}, \quad i,j = 1,\ldots,\rho,$$

where $r_{ij\cdot V\backslash\{i,j\}}$ denotes the partial correlation between the variables $i$ and $j$ given the remaining variables $V\backslash\{i,j\}$, and $\omega_{ij},\ i,j = 1,\ldots,\rho$, denotes the elements of $\Omega^\star$. Under Gaussianity, zero partial correlation implies conditional independence [117, 16].

Recent research into the sparse estimation of $\Omega^\star$ addresses many challenging aspects of estimating $\Omega^\star$ from modern, high-dimensional data. In particular, $\ell_1$-regularized methods induce sparsity, as well as mitigate the instability and rank-deficiency that arise in high-dimensional settings (i.e., when $\rho \gg n$). Sparse estimation of $\Omega^\star$ focusing on model selection is often referred to as *graphical model structure learning*. Broadly, there are two types of statistically motivated methods for estimating $\Omega^\star$: one that uses the Gaussian likelihood function [74, 93, 54], and another that uses some pseudolikelihood-based (i.e., regression-based) formulation [139, 147, 103]. Regularized Gaussian likelihood methods differ only in their regularization terms; however, regression-based regularized pseudolikelihood methods offer more diverse objective functions [158, 119]. In particular, the recently introduced CONCORD [103] and PseudoNet [6] estimators have been shown to have favorable theoretical and practical properties, which we summarize next.

**CONCORD and PseudoNet estimators** minimize a jointly convex $\ell_1$-regularized pseudolikelihood-based objective function, in order to estimate $\Omega^\star$ with $\hat{\Omega}$:

$$\hat{\Omega} = \underset{\Omega\in\mathbb{R}^{\rho\times\rho}}{\operatorname{argmin}} \left\{-\log\det\Omega_D^2 + \operatorname{tr}(\Omega S\Omega) + \lambda_1\|\Omega_X\|_1 + \lambda_2\|\Omega\|_F^2\right\}, \tag{7.1}$$

where $\Omega_D$ and $\Omega_X$ denote matrices containing just the diagonal and off-diagonal elements of $\Omega$, $S = X^TX$ is the sample covariance matrix, $\|\cdot\|_F$ denotes the Frobenius norm, $\|\cdot\|_1$ denotes the elementwise $\ell_1$-norm, and $\lambda_1, \lambda_2 > 0$ are tuning parameters. The $\lambda_2$ term was added in the recent work on PseudoNet [6]; CONCORD's original objective function was a special case, when $\lambda_2 = 0$.

Asymptotically, as $n, \rho \to \infty$, CONCORD was shown to recover the support of the $\Omega$ matrix consistently; PseudoNet improved the consistency proof presented in [103], by providing a two-step method for accurately estimating the diagonal elements of $\Omega$. In simulations, CONCORD was more robust to heavy-tailed data in terms of model selection when compared to the popular $\ell_1$-regularized Gaussian likelihood approach, and PseudoNet improved both parameter estimation and support recovery over CONCORD. PseudoNet also showed practical improvements over CONCORD in a portfolio selection problem. From the standpoint of parallelization and scalability, the two methods are very closely related, so in the interest of brevity, we use CONCORD to refer to both methods in the remainder of this thesis.

**Algorithm 16** CONCORD-ISTA

**Input:** Data matrix $X^{n\times\rho}$, tuning parameter matrix $\Lambda_1^{\rho\times\rho}$.
**Input:** Tuning parameter $\lambda_2$, optimization tolerance $\epsilon > 0$.
**Output:** Estimate $\Omega$.
1: $S \leftarrow X^T X/n$
2: $\Omega_0 \leftarrow I^{\rho\times\rho}$
3: $h_0 \leftarrow -\log\det(\Omega_0)_D + \frac{1}{2}\operatorname{tr}(\Omega_0 S \Omega_0) + \lambda_2\|\Omega_0\|_F^2$
4: **for** $k \leftarrow 0, 1, 2, \ldots$ ▷ Proximal gradient method
5: $\quad G \leftarrow -((\Omega_k)_D)^{-1} + \frac{1}{2}(S\Omega_k + \Omega_k S) + 2\lambda_2\Omega_k$ ▷ Calculating gradient $G$
6: $\quad$ **for** $\tau \leftarrow 1, \frac{1}{2}, \frac{1}{4}, \ldots$ ▷ Backtracking line search
7: $\quad\quad \Omega_{k+1} \leftarrow \mathcal{S}_{\tau\Lambda_1}(\Omega_k - \tau G)$
8: $\quad\quad h_{k+1} \leftarrow -\log\det(\Omega_{k+1})_D + \frac{1}{2}\operatorname{tr}(\Omega_{k+1} S \Omega_{k+1}) + \lambda_2\|\Omega_{k+1}\|_F^2$
9: $\quad\quad q \leftarrow h_k + \operatorname{tr}((\Omega_{k+1} - \Omega_k)^T G) + \frac{1}{2\tau}\|\Omega_{k+1} - \Omega_k\|_F^2$
10: $\quad$ **until** $h_{k+1} \leq q$
11: **until** $\max|\Omega_{k+1} - \Omega_k| < \epsilon$

Computationally, the CONCORD objective function (7.1) can be optimized using a proximal gradient method, CONCORD-ISTA [142], presented in Algorithm 16. The entries of the tuning parameter matrix $(\Lambda_1)_{ij} > 0$, $i, j \in V$, control the amount of $\ell_1$-regularization separately for each element in $\Omega$, while the tuning parameter $\lambda_2 > 0$ controls the amount of $\ell_2$-regularization. We denote the elementwise soft-thresholding operator as $\mathcal{S}_{\Lambda_1}(\Omega) = \operatorname{sign}(\Omega)\max(|\Omega| - \Lambda_1, 0)$, which is the proximal operator for the nonsmooth $\ell_1$-norm part of the CONCORD objective function. As a reminder, CONCORD here refers to two versions of the estimators with and without $\lambda_2$ regularization; similarly, CONCORD-ISTA refers to the algorithm above, with both $\lambda_1$ and $\lambda_2$ regularization terms, possibly with $\lambda_2 = 0$.

In some cases, CONCORD-ISTA can decrease the running time of a coordinate descent-based approach by two orders of magnitude: the authors of [142] show that reconstructing the underlying gene-gene associations in a breast cancer dataset (where $\rho \approx 4{,}000$) using CONCORD-ISTA takes around just 10 minutes. However, CONCORD-ISTA quickly becomes intractable (or at least extremely slow) when analyzing full-sized gene expression data, where $\rho \approx 30{,}000$, because the computational complexity of CONCORD-ISTA can be shown to be $O(d\rho^2)$, where $d$ is the average number of nonzeroes in $\Omega$ on each iteration.

Due to this computational bottleneck, using CONCORD-ISTA on problems with more than a few thousand dimensions is challenging, despite the many desirable theoretical and practical properties of the CONCORD estimators. Furthermore, the running time required to compute estimates across a grid of tuning parameters, often needed in resampling methods such as cross-validation, bootstrap, and stability analysis [138, 126], would be prohibitive. In order to address this scaling challenge, we propose a massively parallel optimization method for graphical model structure learning, which we name HP-CONCORD ("HP" stands for "high performance"). As we show in Section 7.2, HP-CONCORD is able to utilize a

distributed-memory parallel computing system to tackle problems at large scales, taking under 21 minutes to optimize the CONCORD objective function (7.1) for $\rho = 1.28$ million dimensions (corresponding to over 800 billion variables) in some cases.

**Related work.** Other parallel optimization approaches for the sparse recovery of the inverse covariance matrix have been proposed in the literature, including a regularized Gaussian likelihood method called BigQUIC [93], a greedy regularized Gaussian likelihood method [102], and a regularized matrix inverse estimator [174]. In particular, BigQUIC is a highly scalable method for the Gaussian likelihood approach for shared memory systems; we compare it with our method. To our knowledge, however, HP-CONCORD is the only parallel regression-based pseudolikelihood optimization method for the sparse estimation of an inverse covariance matrix.

## 7.2 HP-CONCORD

The most compute-intensive parts of Algorithm 16 are the matrix multiplications $\Omega S$ and $S = X^T X/n$. We consider two algorithmic variations that minimize the total amount of work in different ways, depending on characteristics of the input problem: HP-CONCORD-Cov (**Cov**, for short) and HP-CONCORD-Obs (**Obs**, for short). **Cov** uses $X$ only once to precompute the sample covariance matrix $S$ and uses $S$ throughout (implementing Algorithm 16 as is), whereas **Obs** uses the data/observation matrix $X$ throughout and implicitly recomputes $S$ on the fly (removing line 1 and replacing $S$ with $X^T X$ in Algorithm 16). Algorithms 17 and 18 summarize the most time-consuming operations of **Cov** and **Obs**, respectively. We count the number of floating point operations (flops) of the bottlenecks of Cov and Obs below.

**Algorithm 17** Cov.

1: $S \leftarrow X^T X/n$
2: $W \leftarrow \Omega_0 S$
3: **for** $k \leftarrow 0, 1, 2, \ldots$
4: form $W^T$ to calculate $G$
5: **for** $\tau \leftarrow 1, \frac{1}{2}, \frac{1}{4}, \ldots$
6: $W \leftarrow \Omega_{k+1} S$
7: **until** $h_{k+1} \leq q$
8: **until** $|\Omega_{k+1} - \Omega_k| < \epsilon$

**Algorithm 18** Obs.

1: $Y \leftarrow \Omega_0 X^T$
2: **for** $k \leftarrow 0, 1, 2, \ldots$
3: $Z \leftarrow YX$
4: form $Z^T$ to calculate $G$
5: **for** $\tau \leftarrow 1, \frac{1}{2}, \frac{1}{4}, \ldots$
6: $Y \leftarrow \Omega_{k+1} X^T$
7: **until** $h_{k+1} \leq q$
8: **until** $|\Omega_{k+1} - \Omega_k| < \epsilon$

**Cov** computes $S = X^T X$ once at the beginning (line 1). It computes $W = \Omega S$ once before the loops (line 2) and once per each inner loop (line 6). It transposes $W$ to get $S\Omega$ once per each outer loop ($\Omega$ and $S$ are both symmetric so $(\Omega S)^T = S\Omega$). The term $\text{tr}(\Omega S \Omega)$ in line 3 and 8 of Algorithm 16 is calculated by summing all elements in the result of an element-wise product between $\Omega S$ (stored in $W$) and $\Omega^T$ (same as $\Omega$ since it is symmetric).

$S = X^T X$ is a dense-dense matrix multiplication of size $\rho \times n \times \rho$ which takes $2n\rho^2$ flops. Let $s$ be the number of proximal gradient iterations until convergence, and let $t$ be the average number of backtracking line search iterations per each proximal gradient iteration, and let $d$ be the average number of nonzeroes per row of $\Omega$ throughout all iterations. Then $W = \Omega S$ is a sparse-dense matrix multiplication and takes $2d\rho^2$ flops per iteration on average and is calculated $st + 1$ times. The total number of flops is therefore

$$F_{\text{Cov}} = 2n\rho^2 + 2d\rho^2(st + 1). \tag{7.2}$$

**Obs** never computes the matrix $S$ explicitly, replacing $\Omega S$ with $(\Omega X^T)X$ and $\text{tr}(\Omega S \Omega)$ with $\text{tr}(\Omega X^T X \Omega) = \|\Omega X^T\|_F^2$. It does a sparse-dense matrix-matrix multiplication ($Y = \Omega X^T$) once before the loops (line 1) and once per each inner loop (line 6). It has one dense-dense matrix-matrix multiplication ($Z = YX$ in line 3) and one dense matrix transpose ($Z^T$ in line 4) per each outer iteration. $Y = \Omega X^T$ takes $2dn\rho$ flops. $Z = YX$ takes $2n\rho^2$ flops. The total number of flops is therefore

$$F_{\text{Obs}} = 2n\rho^2 s + 2dn\rho(st + 1). \tag{7.3}$$

The following lemma tells us when Cov is a more efficient option than Obs.

**Lemma 7.1.** *Cov incurs fewer flops than Obs when*

$$\frac{d}{\rho} < \frac{n}{\rho - n} \cdot \frac{1}{t}.$$

*Otherwise, Obs is computationally cheaper than Cov.*

*Proof.* This happens when (7.2) is less than (7.3),

$$\begin{aligned} 2n\rho^2 + 2d\rho^2(st + 1) &< 2n\rho^2 s + 2dn\rho(st + 1) \\ 2d\rho(st + 1)(\rho - n) &< 2n\rho^2(s - 1) \\ d(st + 1)(\rho - n) &< n\rho(s - 1). \end{aligned}$$

We relax the comparison a little by plugging in $st < st + 1$ and $s > s - 1$,

$$\begin{aligned} dst(\rho - n) &< n\rho s \\ \frac{d}{\rho} &< \frac{n}{\rho - n} \cdot \frac{1}{t}. \end{aligned}$$

□

Let $r_{\text{obs}} = n/\rho$ be the ratio of the number of observations to the number of features and let $r_{\text{nnz}} = d\rho/\rho^2 = d/\rho$ be the *average* fraction of nonzeroes of $\Omega$ throughout all iterations, $0 < r_{\text{obs}}, r_{\text{nnz}} \leq 1$, we can reformulate Lemma 7.1 to,

$$r_{\text{nnz}} < \frac{r_{\text{obs}}}{1 - r_{\text{obs}}} \cdot \frac{1}{t}.$$

$r_{\text{obs}}/(1 - r_{\text{obs}})$ is an increasing function of $r_{obs}$. There are scientific problems of interest that favor either Cov or Obs, depending on several factors, including dimensionality and sparsity of the data. The closer $n$ is to $\rho$, the higher $r_{\text{nnz}}$ can be for Cov to still require fewer flops than Obs. For example, assume $t = 10$ (we observed 5-15 inner iterations per one outer iteration in practice) and consider $r_{\text{obs}} = 0.01$, $0.1$, and $0.25$, which results in $r_{\text{nnz}} < 0.001, 0.011$, and $0.033$, respectively. Applications with *average* percentage of nonzeroes (throughout all iterations) less than $0.1\%, 1.1\%$, and $3.3\%$ should benefit from Cov in these cases.

## Distributed operations

Cov and Obs's most expensive operations are its parallel matrix multiplication and global matrix transpose. The challenge is to pick a layout for each matrix, and choose appropriate algorithms that minimize data movement both within and between steps.

The most popular parallel matrix multiplication implementations use a 2D layout [4, 80], treating the processor as a square grid and making each processor responsible for all computations associated with one sub-matrix of the output. 3D [3, 167] algorithms (sometimes called 2.5D) are provably communication-optimal and instead divide the 3D iteration space, essentially making $c$ copies (we will call $c$ the replication factor) of the output matrix and having a rectangular group of processors responsible for a subset of updates to that copy; the copies are summed at the end to produce the final answer. However, as shown in Chapter 5, these are not always the fastest methods in our setting. We describe three special cases based on matrix dimensions and sparsity that arise in HP-CONCORD.

**$\rho \times n \times \rho$ dense-dense ($S = X^T X$ and $Z = YX$).** $X^T$ and $Y$ are tall-skinny and $X$ is short-wide. Partitioning them in a 2D layout (as 2D and 3D algorithms would) would result in tall-skinny and short-wide local matrices, which perform poorly on local memory hierarchies. Instead, we treat the processors as a 1D array and distribute the rows of $X^T$ and $Y$ (a 1D block row layout) and the columns of $X$ (a 1D block column layout).

**$\rho \times \rho \times \rho$ sparse-dense ($W = \Omega S$).** Partitioning all matrices in 1D and shifting just the sparsest matrix around can use much less bandwidth and could outperform the classic 2D and 3D algorithms by up to two orders of magnitude (see Chapter 5). Therefore, we put $\Omega$ in 1D block row and $S$ in 1D block column layout.

**$\rho \times \rho \times n$ sparse-dense ($Y = \Omega X^T$).** All layouts of $Y, \Omega$, and $X^T$ are already chosen from the two multiplications earlier. They all have 1D layout.

Figure 7.1 shows how all distributed operations are connected together. We use the 1.5D algorithms in Section 6.1 and do the distributed transpose according to Section 6.2. Both algorithms replicate $X$ and $\Omega$ $c_X$ and $c_\Omega$ times, respectively. **Cov** computes $S = X^T X$ by shifting $X^T$ (using 1.5D Inner-A). In each iteration, it computes $W = \Omega S$ by shifting $\Omega$ (also

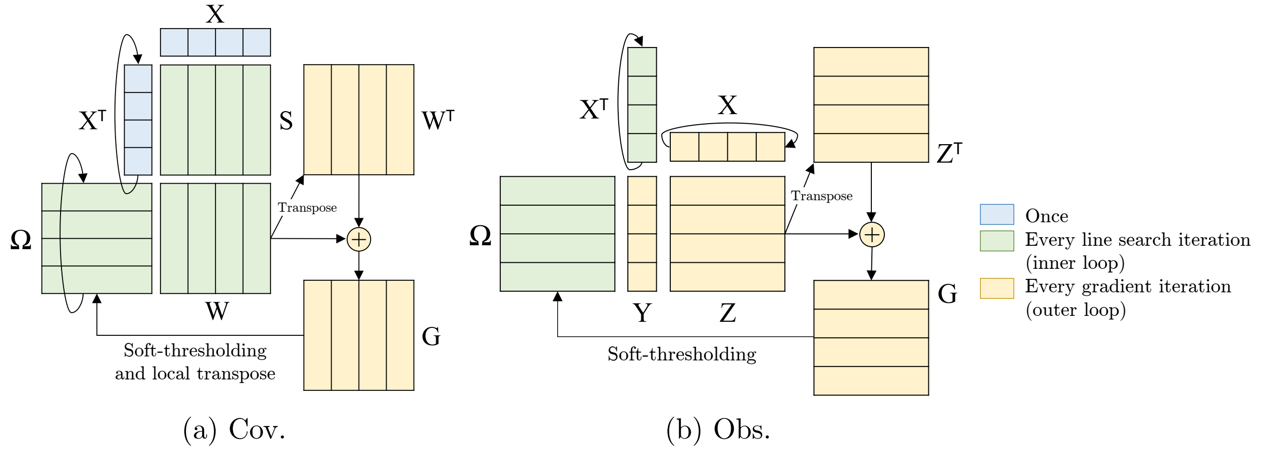



Figure 7.1: Distributed operations of two HP-CONCORD variants: **Cov** uses $X$ only once to precompute the *cov*ariance matrix $S$ and uses $S$ throughout. **Obs** uses the *obs*ervation matrix $X$ every iteration and does not explicitly form $S$. For each multiplication, we draw the first operand on the left and the second operand on the top of the resulting matrix. A loopy arrow shows the matrix being rotated in each multiplication.

using 1.5D Inner-A), globally transposes $W$, computes $G$ from $W$ and $W^T$, soft-thresholds (sparsifies) $G$ to get a new $\Omega$ and converts $\Omega$ back to 1D block row layout by doing a local matrix transpose as it is symmetric. For every iteration, **Obs** computes $Y = \Omega X^T$ by shifting $X^T$ (using 1.5D Row), computes $Z = YX$ by shifting $X$ (using 1.5D Inner-B), globally transposes $Z$ to get $Z^T$ in the same layout, computes $G$ from $Z$ and $Z^T$, and then soft-thresholds $G$ to get the new $\Omega$.

## Total communication costs

**Cov.** According to the costs of 1.5D Inner-A in Table 6.2, $S = X^T X$ takes $p/c_X^2$ messages and $n\rho/c_X$ words. $\Omega S$ takes $p/(c_\Omega c_X)$ messages and $d\rho/c_X$ words. $\Omega S$ is calculated $st$ times. Let $c_{\min} = \min(c_\Omega, c_X)$ and $c_{\max} = \max(c_\Omega, c_X)$. The costs of the distributed transpose can be found in Equation 6.2, replacing $c_A$ with $c_\Omega$ and $c_B$ with $c_X$. The total communication costs of Cov are

$$S_{\text{Cov}} = \frac{p}{c_X^2} + st\frac{p}{c_X c_\Omega} + \log_2\left(\frac{p}{c_{\min}}\right), \tag{7.4}$$

$$W_{\text{Cov}} = \frac{n\rho}{c_X} + st\frac{d\rho}{c_X} + \frac{\rho^2}{p} \cdot \frac{c_{\max}}{c_{\min}} \cdot \log_2\left(\frac{p}{c_{\min}}\right). \tag{7.5}$$

**Obs.** By Table 6.2, the costs of 1.5D Row and 1.5D Inner-B are similar. $Y = \Omega X^T$ and $Z = YX$ both take $p/(c_\Omega c_X)$ messages and $n\rho/c_\Omega$ words. $Y = \Omega X^T$ is computed $st$ times. $Z = YX$ is calculated $s$ times. The distributed transpose costs are the same as Cov's. The

total communication costs of Obs are

$$S_{\text{Obs}} = s(t+1)\frac{p}{c_\Omega c_X} + \log_2\left(\frac{p}{c_{\min}}\right), \tag{7.6}$$

$$W_{\text{Obs}} = s(t+1)\frac{n\rho}{c_\Omega} + \frac{\rho^2}{p}\cdot\frac{c_{\max}}{c_{\min}}\cdot\log_2\left(\frac{p}{c_{\min}}\right). \tag{7.7}$$

## Space complexity

Asymptotically, both algorithms take at least $O(\rho^2)$ storage space: $O(c_X\rho^2)$ for Cov and $O(c_\Omega\rho^2)$ for Obs. We plan to reduce their space requirement by applying blocking with some recomputation in the future. As of now, we simply scale up to more nodes when $\rho$ increases since the computation complexity grows faster than the space. As for the exact memory usage, both variants take the most space when computing $G$. **Cov** needs $S$, $\Omega$, $W$, and $W^T$ in memory. ($G$ can be stored in place of $W$.) **Obs** needs $\Omega$, $X$, $X^T$, $Y$, $Z$, and $Z^T$ in memory. ($G$ can be stored in place of $Z$.) Therefore, their memory requirements are,

$$M_{\text{Cov}} = c_\Omega d\rho + 3c_X\rho^2 \tag{7.8}$$

$$M_{\text{Obs}} = 2c_X n\rho + c_\Omega(d\rho + n\rho + 2\rho^2) \tag{7.9}$$

## Total running time

Plugging in $F_{\text{Cov}}$, $S_{\text{Cov}}$, and $W_{\text{Cov}}$ from Equations 7.2, 7.4, and 7.5 into Equation 2.1, we get the total running time of Cov,

$$\begin{aligned} T_{\text{Cov}} = & \left[\frac{2n\rho^2 + 2d\rho^2(st+1)}{p}\right]\gamma + \\ & \left[\frac{p}{c_X^2} + st\frac{p}{c_X c_\Omega} + \log_2\left(\frac{p}{c_{\min}}\right)\right]\alpha + \\ & \left[\frac{n\rho}{c_X} + st\frac{d\rho}{c_X} + \frac{\rho^2}{p}\cdot\frac{c_{\max}}{c_{\min}}\cdot\log_2\left(\frac{p}{c_{\min}}\right)\right]\beta. \end{aligned} \tag{7.10}$$

Similarly, $F_{\text{Obs}}$, $S_{\text{Obs}}$, and $W_{\text{Obs}}$ from Equations 7.3, 7.6, and 7.7 give us the total running time of Obs,

$$\begin{aligned} T_{\text{Obs}} = & \left[\frac{2n\rho^2 s + 2dn\rho(st+1)}{p}\right]\gamma + \\ & \left[s(t+1)\frac{p}{c_\Omega c_X} + \log_2\left(\frac{p}{c_{\min}}\right)\right]\alpha + \\ & \left[s(t+1)\frac{n\rho}{c_\Omega} + \frac{\rho^2}{p}\cdot\frac{c_{\max}}{c_{\min}}\cdot\log_2\left(\frac{p}{c_{\min}}\right)\right]\beta. \end{aligned} \tag{7.11}$$

Whether $T_{\text{Cov}}$ or $T_{\text{Obs}}$ is better depends on the problem characteristics ($n, p, d, s,$ and $t$), the hardware parameters ($\alpha$ and $\beta$), and the replication factors ($c_\Omega$ and $c_X$, subject to

$c_\Omega c_X \leq P$ and $M_{\text{Cov}}, M_{\text{Obs}} \leq M_{\text{machine}}$, where $M_{\text{machine}}$ is the amount of memory available to one processor).

## 7.3 Experimental Results

Our experiments are designed to (1) determine when to use Cov vs. Obs in practice, (2) illustrate that replication helps avoid communication and increases scalability, and (3) compare HP-CONCORD with BigQUIC, another estimator of the inverse covariance matrix with a C++/OpenMP shared memory implementation. HP-CONCORD is implemented in C++ with OpenMP and MPI, and therefore run on multiple nodes of a cluster. We call threaded MKL for local matrix multiplications. Our test platforms are Edison at National Energy Research Scientific Computing Center (NERSC) and Eos at Oak Ridge Leadership Computing Facility (OLCF). Edison is a 5,586-node Cray XC30 machine with two 12-core Intel Xeon E5-2695 processors at 2.4GHz and 64GB DDR3 RAM each node. Eos is a 736-node Cray XC30 machine with 16-core Intel Xeon E5-2670 at 2.6GHz per node. All running times reported are benchmarked on Edison with 2 MPI processes per node and 12 threads per process, unless noted otherwise. Eos is used to search for penalty parameters and collect the brain results in Section 7.4.

### Cov vs. Obs

**Synthetic datasets.** We generated two synthetic datasets: chain graphs (degree 2) and random graphs (degree 60), fixing $\rho = 40$k features and varying $n$. We searched for the tuning parameters that would give $\Omega$ with degrees close to the solutions (2 and 60). Figure 7.2 shows time to convergence in seconds. The times at $n = 100$ and 200 are higher than larger $n$'s because they took more iterations to converge (see Table 7.1). Cov's running times are

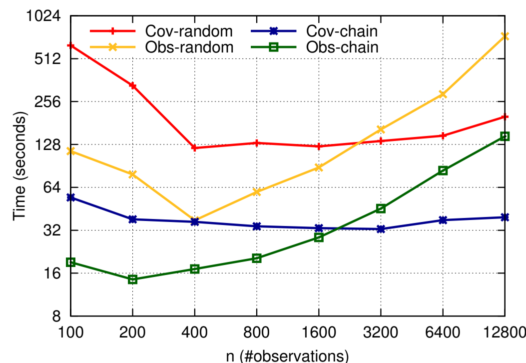



Figure 7.2: Cov vs. Obs on synthetic chain and random graphs ($\rho = 40{,}000$ on 16 nodes = 32 MPI processes = 384 cores). The times at $n = 100$ and 200 are higher than larger $n$'s because they took more iterations to converge.

mostly flat across all $n$'s for both graphs because $F_{\text{Cov}}$'s most dominant term, $2d\rho^2(st+1)$, does not depend on $n$. $F_{\text{Obs}}$ has $n$ in both terms so Obs's running time increases linearly with $n$.

Table 7.1 presents the details necessary to predict the crossover point (where Cov is faster than Obs): the number of proximal gradient (outer loop) iterations ($s$), the total number of backtracking line search (inner loop) iterations, the average number of line search iterations per one gradient iteration ($t$), the average percent nonzeroes throughout all iterations ($d/\rho \times 100\%$), and the predicted % nonzeroes threshold ($n/(\rho - n)/t \times 100\%$) according to Lemma 7.1, which predicts Cov does fewer flops than Obs when $d/\rho \times 100\%$ is less than this threshold, and vice versa otherwise. Our predictions say Cov should be faster than Obs in all cases for chain graphs, and when $n \geq 200$ for random graphs. The actual crossover points happen much later than in theory, between $n$ = 1,600 and 3,200 for both chain and random graphs, because most of Cov's flops are from sparse-dense matrix multiplication while a significant chunk of Obs's flops comes from dense-dense matrix multiplication, which is much more efficient ($\gamma_{\text{dense-dense}} \ll \gamma_{\text{sparse-dense}}$). Table 7.1 also presents the flops rate for all local matrix multiplications in Cov and Obs, i.e., not including distributed communication

| Graph | Counter | $n$ (observations) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 100 | 200 | 400 | 800 | 1,600 | 3,200 | 6,400 | 12,800 |
| Chain | Gradient iterations: $s$ | 28 | 21 | 20 | 18 | 17 | 16 | 16 | 15 |
| | Line search iterations | 71 | 46 | 44 | 40 | 38 | 36 | 36 | 34 |
| | Average line/grad: $t$ | 2.54 | 2.19 | 2.20 | 2.22 | 2.24 | 2.25 | 2.25 | 2.27 |
| | Average % nonzeroes: $d/\rho \times 100\%$ | 0.0101 | 0.0081 | 0.0075 | 0.0075 | 0.0075 | 0.0075 | 0.0075 | 0.0075 |
| | % nonzeroes threshold prediction | 0.0988 | 0.2294 | 0.4591 | 0.9184 | 1.8640 | 3.8647 | 8.4656 | 20.7612 |
| | Cov: $S = X^TX$ Gflops/core | 15.02 | 16.36 | 4.54 | 7.81 | 7.90 | 13.91 | 14.55 | 15.48 |
| | Cov: $W = \Omega S$ Gflops/core | 0.71 | 0.64 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 |
| | Cov: Average Gflops/core | 0.93 | 1.41 | 1.74 | 3.07 | 4.39 | 8.10 | 10.51 | 12.95 |
| | Obs: $Y = \Omega X^T$ Gflops/core | 0.35 | 0.33 | 0.36 | 0.33 | 0.11 | 0.08 | 0.08 | 0.08 |
| | Obs: $Z = YX$ Gflops/core | 15.51 | 16.71 | 13.69 | 16.33 | 16.90 | 17.00 | 16.86 | 16.10 |
| | Obs: Average Gflops/core | 15.33 | 16.56 | 13.61 | 16.19 | 16.48 | 16.38 | 16.26 | 15.54 |
| Random | Gradient iterations: $s$ | 155 | 102 | 47 | 58 | 58 | 66 | 61 | 76 |
| | Line search iterations | 519 | 269 | 88 | 91 | 77 | 84 | 94 | 131 |
| | Average line/grad: $t$ | 3.35 | 2.64 | 1.87 | 1.57 | 1.33 | 1.27 | 1.54 | 1.72 |
| | Average % nonzeroes: $d/\rho \times 100\%$ | 0.1824 | 0.1706 | 0.1694 | 0.1656 | 0.1668 | 0.1669 | 0.1730 | 0.1781 |
| | % nonzeroes threshold prediction | 0.0748 | 0.1905 | 0.5395 | 1.3007 | 3.1385 | 6.8323 | 12.3607 | 27.3013 |
| | Cov: $S = X^TX$ Gflops/core | 15.00 | 16.33 | 4.99 | 5.78 | 7.69 | 10.76 | 15.19 | 15.74 |
| | Cov: $W = \Omega S$ Gflops/core | 1.11 | 1.12 | 1.13 | 1.13 | 1.13 | 1.12 | 1.11 | 1.12 |
| | Cov: Average Gflops/core | 1.12 | 1.13 | 1.18 | 1.25 | 1.41 | 1.65 | 2.04 | 2.41 |
| | Obs: $Y = \Omega X^T$ Gflops/core | 1.47 | 1.74 | 1.91 | 1.36 | 1.10 | 0.91 | 0.83 | 0.80 |
| | Obs: $Z = YX$ Gflops/core | 16.12 | 16.74 | 15.35 | 16.15 | 16.84 | 16.98 | 17.03 | 16.16 |
| | Obs: Average Gflops/core | 15.20 | 16.11 | 15.02 | 15.70 | 16.32 | 16.36 | 16.18 | 15.26 |

Table 7.1: Details for Figure 7.2. From the top: number of proximal gradient iterations (outer loop), total number of backtracking line search iterations (inner loop), average number of line search iterations per one gradient iteration ($t$ in Lemma 7.1), average % nonzeroes of $\Omega$ throughout all iterations, % nonzeroes threshold prediction according to Lemma 7.1 (Cov wins if the average % nonzeroes is less than this number, Obs wins otherwise), and flop rates (in Gflops) for various local matrix multiplications in Cov and Obs.

times. Cov has one dense-dense matrix multiplication at the beginning which was able to hit flop rates close to the theoretical peak (19.2 Gflops/core), but it does $st$ sparse-dense matrix multiplications, which achieved relatively much lower flop rates, causing its overall flop rates to be low. Obs does $st$ smaller sparse-dense matrix multiplications ($\rho \times \rho \times n$, as opposed to Cov's $\rho \times \rho \times \rho$) which attained low flop rates, but the more flops-heavy multiplication is the dense-dense multiplication with great flop rates, so its overall flop rates stay relatively close to the machine peak.

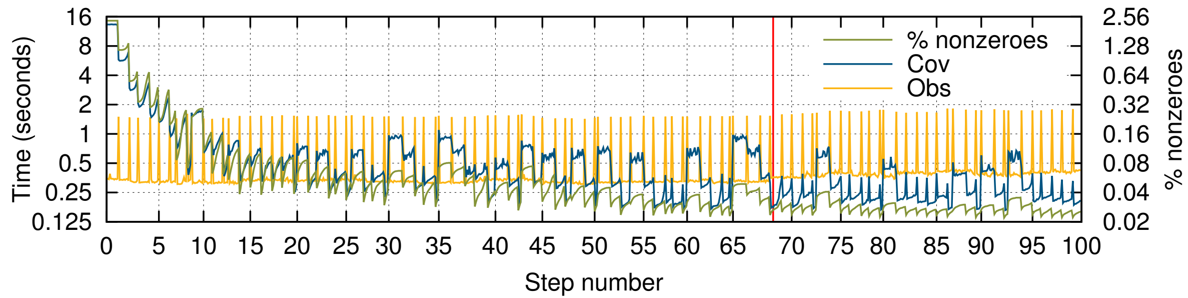
**Real dataset.** We ran Covs and Obs on real 3-dimensional resting state fMRI (functional Magnetic Resonance Imaging) data of the human brain for 100 proximal gradient iterations on 8,192 cores with 1 MPI process per core. We use a time series data with 1,200 timepoints (observations) and 110k brain voxels (features) from The Human Connectome Project [163]. See more details on the brain fMRI data in Section 7.4, where we use HP-CONCORD to segment another brain dataset (group-average data) from the same project.

Figure 7.3 compares the time Cov and Obs spent in each backtracking line search. Each tick on the x-axis indicates the beginning of each proximal gradient iteration. The distances between ticks are not uniform because each gradient iteration performs a different number of line search iterations. Figure 7.3a shows all 100 iterations. Figures 7.3b-7.3d *zoom in* to show more details.

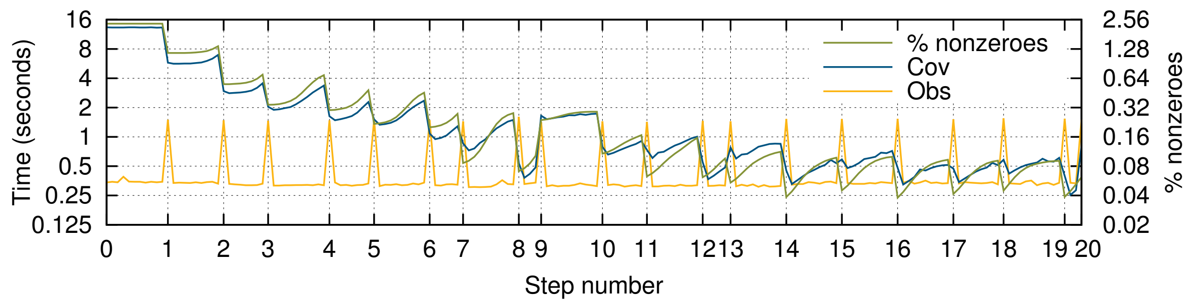
The total running times are the areas under each line: 663.16 seconds for Cov and 431.35 seconds for Obs. Obs' running time stays invariant throughout most iterations because its computational complexity does not depend on $\Omega$'s sparsity. Its yellow spikes at the beginning of each gradient iteration are due to the multiplications $Z = YX$. Cov's running time matches closely with the % nonzeroes of $\Omega$ as expected. As $\Omega$ gets sparser, Cov's time for each line search iteration starts getting cheaper than Obs around iteration 50. This suggests a hybrid version where we start off with Obs and switch to Cov mid-way when $\Omega$ gets sufficiently sparse.

## Replication effects

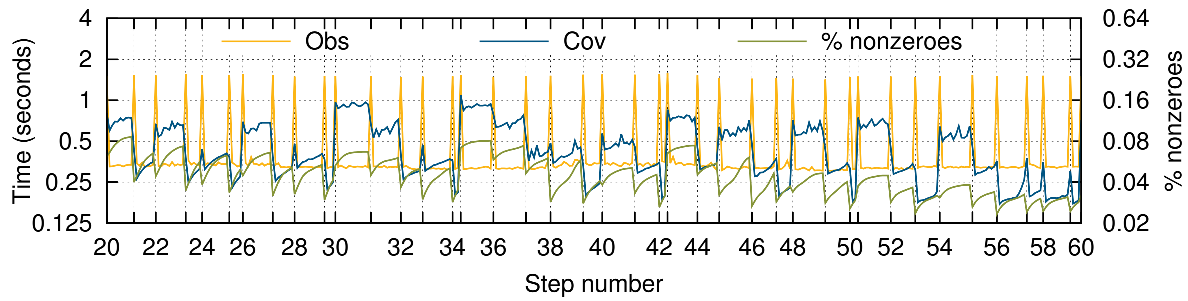
Replication can improve running time drastically. We observed up to a 10.18× speedup over the non-replicating version in our synthetic experiments. To illustrate, we run all possible replication configurations on a chain graph with $n = 100$, $p = 40{,}000$ on 256 nodes (512 MPI processes, 6,144 cores). Figure 7.4a shows the running times of Obs. (There is no $c_\Omega = 512$ because of the memory limit since the dense $\rho \times \rho$ matrices are also replicated $c_\Omega$ times.) Enabling various replication factors allows our algorithm to cover many new approaches, in addition to the common ones. At $(c_\Omega, c_X) = (1, 1)$, our algorithm degenerates to the non-communication-avoiding version, which partitions everything to $P$ equal parts. It took the longest running time, as expected. The notation $(1, 512)$ means every MPI process has the whole $X$ in memory, does all multiplications locally, and only communicates when replicating $X$ and during the transpose. The best replication factor is at $(16, 8)$, 5× faster than $(1, 1)$.



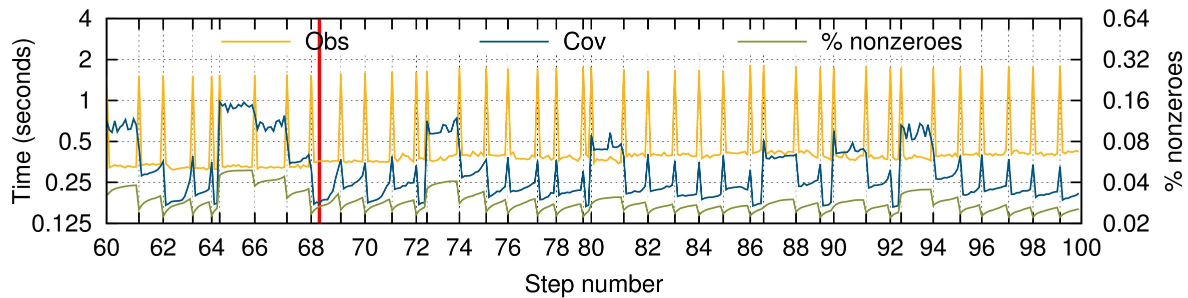

(a) Overall (895 inner iterations).




(b) Gradient steps 0-19 (175 inner iterations).




(c) Gradient steps 20-59 (358 inner iterations).



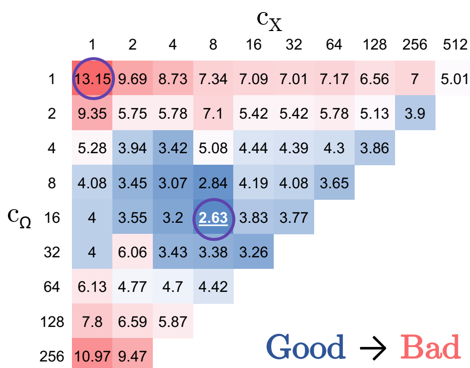

(d) Gradient steps 60-99 (362 inner iterations).

Figure 7.3: The time Cov and Obs took in each line search iteration (inner iteration). The ticks on the x-axis show the gradient step number (outer loop index). Subfigure (a) shows all steps from 0-99. Subfigures (b), (c), and (d) feature the step range in more details. Yellow spikes at the beginning of gradient steps are due to $YX$. Cov's running time varies closely with the sparsity while Obs' is invariant as expected. The red line shows an example point where switching from Obs to Cov would be beneficial.

Figure 7.4b shows cost breakdowns for selected replication factor pairs $(c_\Omega, c_X) = (1,1)$, $(4,1)$, $(16,1)$, and $(16,8)$. Blue is input read time and is the same across all replication pairs. $\Omega X^T$ has three major costs: shift (in red), local matrix multiplication (in dark green), and allreduce (in orange) which sums all copies of $Y$ within the same team. $YX$ has two main costs: shift (in pink) and local matrix multiplication (in green). The transpose operation consists of processors doing a distributed all-to-all operation (within the communication groups defined in Section 6.2), shown in yellow, and local transpose, shown in teal.

Both shift costs decrease with the increasing replication factors. Computation times decrease as well because of better cache reuse. The transpose cost is more complicated and does not show a clear trend, as the latency and bandwidth decreases logarithmically with $c_{\min} = \min(c_\Omega, c_X)$ but the bandwidth also increases when $c_\Omega : c_X$ ratio is not close to 1 (see Equation (6.2)). However, if we compare the transpose cost of $(16, 8)$ to $(1, 1)$, we can still see a clear decrease, since 16 is close to 8, and $8 > 1$. The allreduce cost of $\Omega X^T$ increases with $c_\Omega$ but not $c_X$ because $c_\Omega$ controls the matrix size to send, and it is also the number of processors that perform the allreduce together. Khaki shows the allreduce time for statistics-collecting purposes, e.g., reporting % nonzeroes every line search iteration, etc. – it is not related to actual computation. This allreduce is done across teams (within layers), for example, summing the total number of nonzeroes of all matrix parts from all teams. As $c$ increases, there are fewer teams, so the cost decreases.

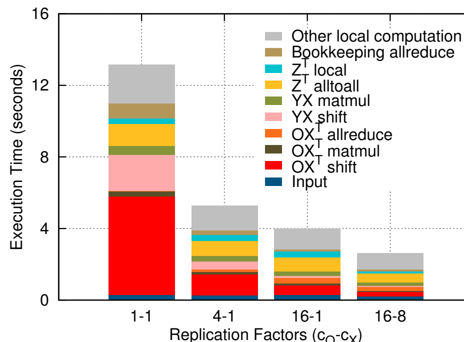
Lastly, grey color shows other local computation time. It is relatively large because of two reasons: (1) The number of passes a local matrix is read through has not been optimized. We wrote this as a matrix library, and some operations that can be done together are done separately for composability and readability. For example, the computation of $G$ in line 5 of Algorithm 16 computes the three summing terms separately, first $G = 0.5(Z + Z^T)$, then $G -= ((\Omega_D))^{-1}$, and $G += 2\lambda_2\Omega$. $G$ is read and written multiple times. We can reduce the



| $c_\Omega$ \ $c_X$ | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 13.15 | 9.69 | 8.73 | 7.34 | 7.09 | 7.01 | 7.17 | 6.56 | 7 | 5.01 |
| 2 | 9.35 | 5.75 | 5.78 | 7.1 | 5.42 | 5.42 | 5.78 | 5.13 | 3.9 | |
| 4 | 5.28 | 3.94 | 3.42 | 5.08 | 4.44 | 4.39 | 4.3 | 3.86 | | |
| 8 | 4.08 | 3.45 | 3.07 | 2.84 | 4.19 | 4.08 | 3.65 | | | |
| 16 | 4 | 3.55 | 3.2 | 2.63 | 3.83 | 3.77 | | | | |
| 32 | 4 | 6.06 | 3.43 | 3.38 | 3.26 | | | | | |
| 64 | 6.13 | 4.77 | 4.7 | 4.42 | | | | | | |
| 128 | 7.8 | 6.59 | 5.87 | | | | | | | |
| 256 | 10.97 | 9.47 | | | | | | | | |



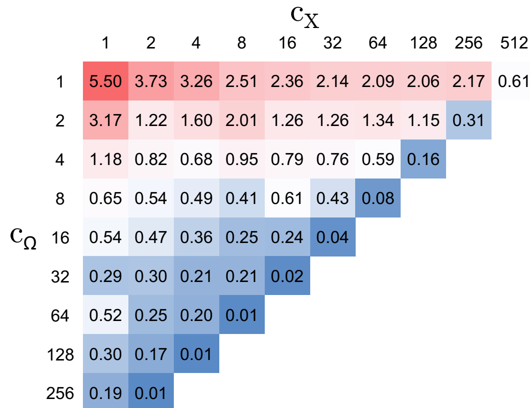
(a) Overall running time in seconds



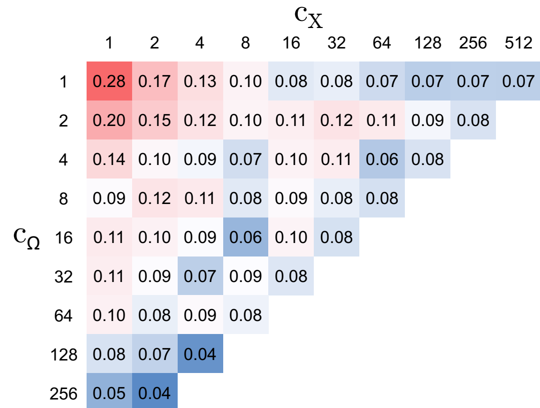


(b) Cost breakdown

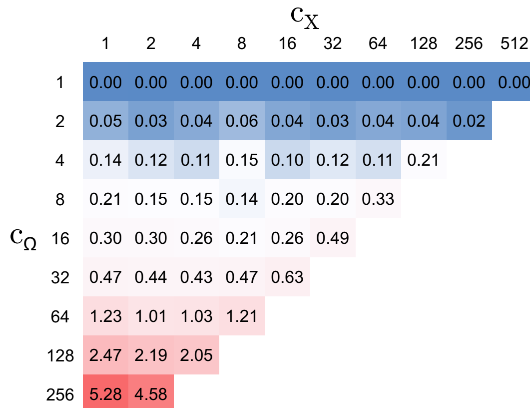
Figure 7.4: Obs' running times in seconds with various replication factors for the chain graph on 256 nodes with $n = 100$ and $p = 40{,}000$. At $(c_\Omega, c_X) = (16, 8)$, the algorithm achieves a factor of five speedup over the non-communication-avoiding result at $(1, 1)$.



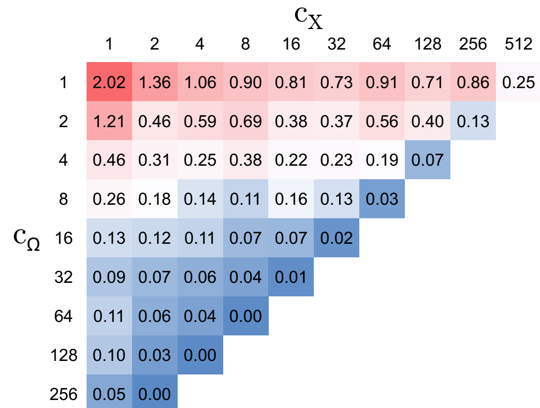

(a) $\Omega X^T$: shift.



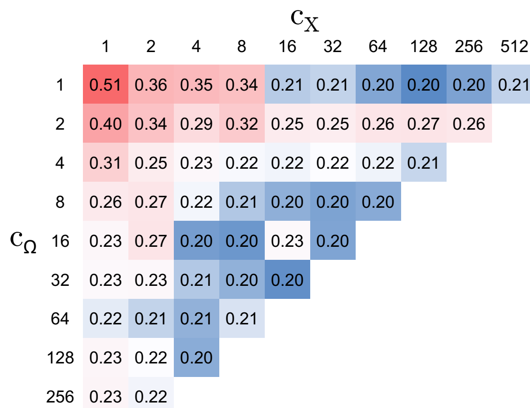

(b) $\Omega X^T$: local matrix multiplication.



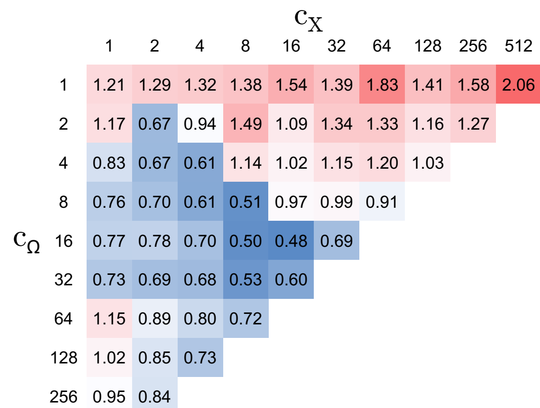

(c) $\Omega X^T$: allreduce.




(d) $YX$: shift.




(e) $YX$: local matrix multiplication.




(f) $Z^T$: all-to-all communication.

Figure 7.5: Some of Obs' time breakdowns in seconds for Figure 7.4.

time by trading-off modularity and writing a kernel that does everything in one go. Another example is the Frobenius norm computation. It is implemented as a separate function and it requires a full pass over the matrix. It can easily be computed with other computations. (2) Some parts are still not efficiently threaded.

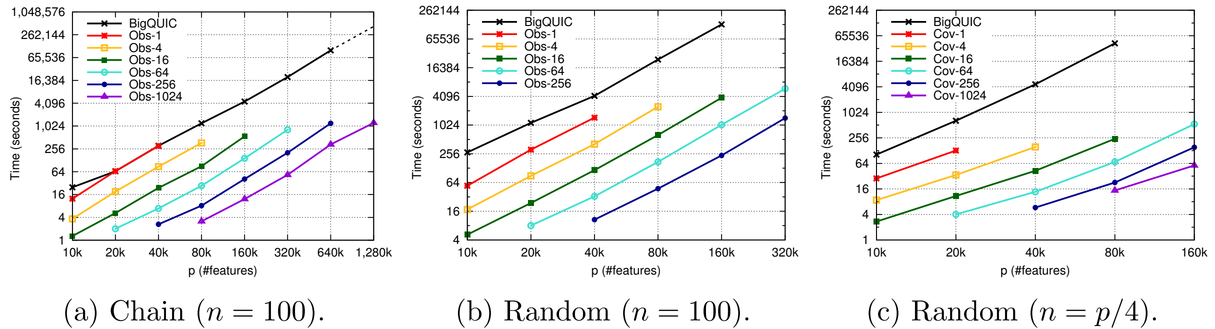
Even at the best replication factor pair $(16, 8)$, communication still comprises a large fraction of the total running time, especially the transpose costs. Further optimization includes communication overlapping.

Figure 7.5 compares each cost breakdowns between all replication factor pairs. Figures 7.5a and 7.5d confirm that the shift time decreases with either $c$'s. The $\Omega X^T$ local matrix multiplication time in Figure 7.5b depends on matrix shapes and sparsity, but also shows a general trend that larger local matrix sizes achieve better efficiency. The $YX$ local matrix multiplication cost in Figure 7.5e favors $c_\Omega : c_X$ ratio close to 1, in addition to large local matrix sizes. This is logical since $Y$ and $X$ has the same number of nonzeroes and similar shapes. The allreduce cost in Figure 7.5c only changes with $c_\Omega$ but not with $c_X$ because it only operates on $Y$ (where $Y = \Omega X^T$) and $Y$ is replicated $c_\Omega$ times. Finally, the transpose cost is best when $c_\Omega = c_X$, as shown in Equation 6.2.

## Comparison with BigQUIC

BigQUIC [93] is a second-order method with a C++/OpenMP shared memory implementation. It is the only other ICM (Inverse Covariance Matrix) estimation method we are aware of that can handle more than a few tens of thousands of variables. We generated three synthetic datasets and chose the tuning parameters so that BigQUIC and HP-CONCORD recover the same number of edges (nonzeroes). We also ran HP-CONCORD on multiple nodes to test its scalability. At each number of nodes, we tried several different replication factors and picked the best running time.

**Chain graphs.** We fixed $n = 100$ and varied $\rho$ from 10,000 to 1,280,000 to reproduce BigQUIC's experiment on chain graphs. As $d/\rho$ is not too much smaller than $n$, we used HP-CONCORD with Obs. Figure 7.6a and Table 7.2 show the total running time and number of iterations to convergence. The black line represents BigQUIC's time on one node. The colored lines are the times of Obs on 1, 4, 16, 64, 256, and 1,024 nodes. The last point at 1,280,000 for BigQUIC was interpolated because it did not finish in 96 hours, which is the maximum running time allowed on Edison. BigQUIC has the computational complexity $O((\rho + |B|)d\rho T T_{\text{outer}})$ where $|B|$ is the number of boundary nodes, $T$ is the number of steps, and $T_{\text{outer}}$ is the number of conjugate gradient iterations. It is a second-order method, so it took many fewer steps to converge than ours which is a first-order method. Even though the computational complexity and convergence rate of BigQUIC and CONCORD are vastly different, HP-CONCORD matched the running time of BigQUIC, when both ran on one node. HP-CONCORD also demonstrated good scalability, allowing the user to choose the running time they want for their problem size, e.g., at 80,000 features, they can get the results in under 4 seconds with 1,024 nodes.




(a) Chain ($n = 100$). (b) Random ($n = 100$). (c) Random ($n = p/4$).

Figure 7.6: Running times of BigQUIC on 1 node (24 cores) and HP-CONCORD on 1 to 1,024 nodes (24 to 24,576 cores).

| Graph | Method | $\rho$ (features) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 10K | 20K | 40K | 80K | 160K | 320K | 640K | 1,280K |
| Chain | BigQUIC | 6 | 5 | 6 | 6 | 5 | 5 | 5 | - |
| | HP-CONCORD | 25 | 33 | 37 | 36 | 43 | 51 | 69 | 57 |
| Random ($n = 100$) | BigQUIC | 6 | 6 | 5 | 6 | 5 | - | - | - |
| | HP-CONCORD | 114 | 144 | 155 | 203 | 270 | 330 | - | - |
| Random ($n = \rho/4$) | BigQUIC | 5 | 5 | 5 | - | - | - | - | - |
| | %PPV | 99.48 | 99.71 | 99.78 | 99.81 | - | - | - | - |
| | %FDR | 0.52 | 0.29 | 0.22 | 0.19 | - | - | - | - |
| | HP-CONCORD | 16 | 17 | 17 | 21 | 35 | - | - | - |
| | %PPV | 99.75 | 99.92 | 99.94 | 99.94 | 99.20 | - | - | - |
| | %FDR | 0.25 | 0.08 | 0.06 | 0.06 | 0.80 | - | - | - |

Table 7.2: Numbers of iterations BigQUIC and HP-CONCORD took to converge in the chain and random graphs experiments. The tuning parameters $(\Lambda_1)_{ij}$ were chosen to give about 17% of the true number of nonzeros, relative to an underlying $\Omega^\star$. "%PPV" and "%FDR" indicate the positive predictive values and false discovery rates, respectively, relative to $\Omega^\star$.

**Random graphs.** We generated random graphs with degree 60. Figure 7.6b fixes $n = 100$ and varies $\rho$ from 10,000 to 320,000. We also use Obs because $d/\rho$ here is even greater than in the chain graph case. Obs was 4 times faster than BigQUIC even on one node. Obs has even better scalability here than in the chain graphs since $\Omega$ is less sparse and the algorithm has more flops to compute. Figure 7.6c uses $n = \rho/4$ for the same set of $\rho$. We use Cov here because $n$ is large and see a similar trend as Figure 7.6b. Although the two values of $n$ use different implementation approaches (Obs vs Cov), both dominate BigQUIC on a single node and scale well to multiple nodes using our communication-avoiding approach.

## 7.4 Massive-scale Structure Learning from fMRI Data

In this section, we reconstruct the functional connectivity structure of the human brain from high-dimensional blood oxygenation level dependent (BOLD) signal measurements in functional magnetic resonance imaging (fMRI) data. In practice, a commonly used measure of functional connectivity (at fine resolution) is marginal or Pearson correlation. However, pairwise marginal correlations may be driven by indirect relationships through a third variable [67], and partial correlations are more suitable for modeling direct associations [132].

Using HP-CONCORD, we estimate a sparse partial correlation matrix as a measure of functional connectivity for the whole brain. In this context, an element in the partial correlation matrix represents the residual correlation between a pair of points after regressing out the remaining variables. Estimating partial correlations at the native resolution of modern fMRI scanner is computationally challenging [93]. With HP-CONCORD, a sparse partial correlation matrix for the whole brain can be reconstructed in controllable time using a parallel computing system.

Subsequently, the inferred partial correlation graph is used to induce a functional connectivity based parcellation [67]. To parcellate the brain into regions, we pass the functional dependency structure output from HP-CONCORD to graph clustering algorithms. We compare the resulting parcellations (clusterings) to a state-of-the-art clustering from the neuroscience literature by Glasser et al.[82]. Our preliminary and entirely data-driven parcellations are able to capture some of the important features presented in Glasser's parcellations [82] that combines multimodal imaging data using significant domain knowledge.

### Data

We use a 91,282×91,282 extensively processed, group-average sample correlation matrix from the Human Connectome Project [163]. The first, second, and last ∼30K features correspond to the left hemisphere, the right hemisphere, and sub-cortical regions, respectively. The size of the dataset is about 60 gigabytes. The matrix was generated in the following way (c.f. Figure 2 of [163]). First, 1,200 subjects were put into a state-of-the-art fMRI machine and measurements were taken without stimulating the subjects every 0.7 seconds for an hour, at 2 millimeter × 2 millimeter × 2 millimeter cubes/voxels spread evenly throughout the cerebral cortex. Next, as fMRI data is typically very noisy, a significant amount of post-processing was done to denoise the data, ultimately leading to a data matrix $X$ with dimensions $n \approx 6,171,400,\ p = 91,282$. To further reduce the level of noise, the columns of the data matrix were then averaged over the 1,200 subjects, leading to a data matrix with dimensions $n \approx 5,142,\ p = 91,282$, from which the sample covariance matrix was finally computed.

## Approach

We (1) generate a partial correlation graph using HP-CONCORD, and then (2) apply a graph-based clustering algorithm to the partial correlation graph arising from the sparsity pattern of the HP-CONCORD estimate. For (1), we consider all combinations of the tuning parameters $(\lambda_1, \lambda_2) \in L_1 \times L_2$, where

$$L_1 = \{0.48, 0.5, 0.52, 0.54, 0.57, 0.59, 0.61, 0.64, 0.67, 0.69, 0.72\},$$
$$L_2 = \{0.10, 0.13, 0.16, 0.2, 0.25, 0.31, 0.39, 0.49\}.$$

Tuning parameters outside these ranges yielded either trivially sparse or dense estimates. For each graph generation, we used scalar values of $\lambda_1$ and $\lambda_2$ for simplicity. Qualitative comparison with thresholded sample correlation matrix shows the advantage of using partial correlation approach.

Subsequently, for (2), we consider two graph clusterings: the well-known Louvain method [31], and a relatively new clustering method from the persistent homology literature [66] that leverages the degree matrix associated with the partial correlation graph. We summarize the method in the next paragraph. Additionally, because the clusterings from [82] treat the left and right hemispheres of the brain separately, we also run and evaluate our clustering algorithms on the subgraphs associated with only the left and right hemispheres.

**Persistent Homology.** We map the degree of a vertex in the inverse covariance graph, described by matrix $\Omega^{p\times p}$, onto the surface of a brain. We thus obtain a function $f : S \to \mathbb{Z}$, where $S$ is the triangulation of the cortical surface. We apply the watershed algorithm [51] to $f$ by sweeping the vertices from the highest value to the lowest. We start a new label if the vertex has no labeled neighbors in $S$. If it does, we propagate the label with the maximum starting value.

The resulting parcellation is usually too fine: every local maximum of $f$ produces a new label. We use the theory of persistent homology [66] to coarsen the parcellation. During the sweep, we build the dual graph $G$ of the labels. When we start a new label $l$ at a vertex $u$, we add $l$ to the graph and assign to it the value $f(u)$. When two labels, $l_1$ and $l_2$, that fall in different components of $G$, meet at a vertex $v$ during the sweep, we add an edge $(l_1, l_2)$ to $G$. We find the maximum values, $a_1$ and $a_2$, assigned to any vertex in the components of $l_1$ and $l_2$ in $G$, and assign to the new edge the value $\min\{a_1, a_2\} - f(v)$. (It's not difficult to verify that this is exactly the persistence of the vertex $v$.)

Once we construct the dual graph $G$, given a simplification threshold $\varepsilon$, we treat the connected components of the subgraph of $G$ induced by the edges with values at most $\varepsilon$ as the new parcels. As we increase $\varepsilon$, the parcels merge, and the parcellation gets coarser.

## Evaluation

Our main points of comparison are the state-of-the-art clusterings, for the left and right hemispheres, from Glasser et al.[82], presented in Figure 7.7. However, we also consider a

simple baseline, given by discarding 99, 99.1, ..., 99.8, 99.9, 99.91, ..., 99.98, 99.99% of the sample covariance matrix entries, i.e., keep entries with the largest magnitudes (c.f. [134]) in order to generate marginal correlation graphs. This baseline lets us probe the comparative advantage of using marginal vs. partial correlations. To quantitatively compare clusterings, we consider a variation of the standard Jaccard score,

$$\mathrm{Sim}(\mathcal{C}_1, \mathcal{C}_2) = \frac{1}{\max(k, \ell)} \cdot \sum_{(i,j)\in\mathcal{E}} W_{ij}, \tag{7.12}$$

where $\mathcal{C}_1 = \{A_1, \ldots, A_k\}$ and $\mathcal{C}_2 = \{B_1, \ldots, B_\ell\}$ are two clusterings, $\mathcal{E} \subseteq \{1, \ldots, k\} \times \{1, \ldots, \ell\}$ is a maximum weighted edge covering in a weighted bipartite graph, where the vertices on a side of the graph correspond to the clusters in a clustering, and the edge weights $W_{ij},\ i = 1, \ldots, m,\ j = 1, \ldots, \ell$, are the usual Jaccard scores given by $\frac{|A_i \cap B_j|}{|A_i \cup B_j|}$. The use of the edge covering here resolves various complications that arise when comparing clusterings of different sizes; the $\frac{1}{\max(k,\ell)}$ term in (7.12) can be thought of as a normalizing constant Finally, to compute the edge covering, we use the algorithm of Azad et al. [15].

## Results

Table 7.3 shows examples of dependency structures recovered by HP-CONCORD and thresholding the sample correlation matrix. Top two sparsity patterns are the most prominent partial correlations recovered by HP-CONCORD corresponding to two different sets of two penalty parameter choices, $\lambda_1$ and $\lambda_2$. The bottom sparsity pattern is from thresholding the sample correlation matrix. Striking features of the sparsity patterns of partial correlation matrices are (1) the pronounced block-diagonal structure, and (2) spatial locality of the most prominent relationships. (3) Furthermore, the differences between the partial correlation matrix and thresholded marginal correlation matrix may seem subtle due to the extreme size of the matrix; but, the seemingly subtle differences clearly result in significant difference in the downstream analysis. We discuss these three aspects in more detail below. We emphasize that the features leading to the following observations arise naturally, without being hard-coded into our method and without imposing any assumptions about the underlying functional connectivity structure.

(1) In Table 7.3, HP-CONCORD estimates show near perfect block-diagonal structures, where the blocks turn out to correspond to the left and right hemispheres. The recovered block diagonal structure indicates that much of the variation at any given point can be explained by neighbors in the same hemisphere. (2) Furthermore, the sparsity patterns of the blocks themselves turn out to correspond to the spatially closest voxels (Figure 7.8 in Section 7.5), which is consistent with the belief in neuroscience [145]. When taken together, these two observations suggest that locally contiguous regions of the brain are functionally more closely associated as compared to their respective symmetric areas in the mirroring hemisphere. However, thresholded sample correlation matrix shows significant off-block-diagonal structures which are likely to be indirect associations [132].

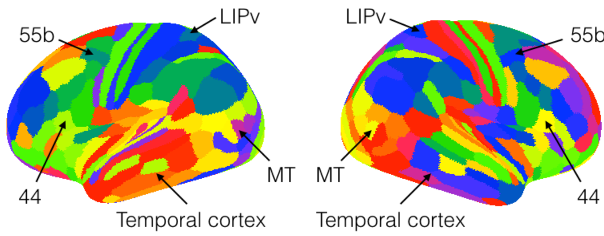



Figure 7.7: Left and right brain hemisphere clusterings from Glasser et al. [82], generated by applying a multi-class, shallow neural network to the same data we use [163], but use a significant amount of domain knowledge in order to post-process the results by hand. The colors have no significance, except to demarcate the different clusters.

| | Whole brain's sparsity pattern | Left hemisphere's parcellation | Right hemisphere's parcellation |
|---|---|---|---|
| HP-CONCORD + Persistent Homology | $\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$ % of best score = 100 | $\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$ % of best score = 100 | $\lambda_1 = 0.5,\ \lambda_2 = 0.39,\ \epsilon = 3,$ % of best score = 100 |
| HP-CONCORD + Louvain | $\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$ % of best score = 75.03 | $\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$ % of best score = 75.03 | $\lambda_1 = 0.5425,\ \lambda_2 = 0.39,\ k = 0,$ % of best score = 73.45 |
| Thresholding sample covariance matrix + Louvain | $t = 99.9,\ k = 4$ % of best score = 32.24 | $t = 99.9,\ k = 4,$ % of best score = 32.24 | $t = 99.9,\ k = 3,$ % of best score = 32.45 |

Table 7.3: The best clusterings relative to Glasser's[82], according to our modified Jaccard score. The leftmost column presents the sparsity patterns of the whole brain (black indicates a nonzero entry) of the estimate yielding the best clustering for the left hemisphere. The score under each figure is the percentage of the best Jaccard score it attains (higher is better); since the persistent homology clusterings perform the best, these percentages are just 100. The actual Jaccard scores, as well as a significantly expanded set of results, can be found in Section 7.5. Also indicated are the tuning parameter values yielding the clusterings (i.e., $\lambda_1, \lambda_2$ for HP-CONCORD; $\varepsilon \geq 0,\ k \in \mathbb{Z}_+$ controlling the number of clusters for the persistent homology and Louvain methods, respectively; and $t$ denoting the percentage of discarded sample covariance matrix entries). The colors in the various plots have no special meaning.

(3) Often in practice, inferred dependency structures/functional connectivities are used for downstream analysis. We will illustrate the significant difference between the estimated sparse partial correlation and thresholded sample correlation matrices by using them as inputs to a functional connectivity based parcellation clustering procedure. Although the sparsity patterns of the HP-CONCORD and sample covariance matrix estimates appear vaguely similar, the subtle differences between them drive the significant differences in the resulting clusterings.

The top and middle rows of Table 7.3 present the best clusterings generated by HP-CONCORD followed by the persistent homology and Louvain methods, respectively, when compared to Glasser's clusterings [82] presented in Figure 7.7, according to the modified Jaccard score; the bottom row presents the best clusterings generated by thresholding the sample covariance matrix at various levels. The middle and right columns present the results for the left and right hemispheres, respectively. We see that the persistent homology clusterings perform the best, in terms of Jaccard score, across both hemispheres.

Qualitatively, we see that the persistent homology clusterings are able to identify several clusters of interest to the neuroscience community (c.f. Figure 3 in [82]); this is certainly encouraging, since we do not expect perfect recovery of all the clusters in Figure 7.7, as the latter clusters rely on a significant amount of domain knowledge.[1] Some examples: the persistent homology clusterings seem to pick out area 55b, involved in hearing; the lateral intraparietal cortex (LIPv), involved in eye movement; and much of the variation in the temporal cortex, involved in processing information from the senses. On the other hand, the Louvain method and the clusterings generated by the sample covariance matrix seem to miss these clusters, as they appear overly smooth. Along these lines, all the methods seem to miss Brodmann's area 44, involved in hearing and speaking, and the middle temporal visual area (MT), involved in seeing moving objects.

## 7.5 Expanded Set of Results

This section provides supplemental details on the experiments in Section 7.4. Readers might want to skip this section and go directly to the conclusions in Section 7.6.

**The sparsity patterns of the HP-CONCORD estimates.** The sparsity patterns of the diagonal blocks of the HP-CONCORD estimates correspond to the spatially closest voxels on the left and right hemispheres. In Figure 7.8, we present the sparsity pattern of the HP-CONCORD estimate attaining the best clustering, for the left hemisphere, in the middle column of Table 7.3. In Figure 7.8, we also present the sparsity pattern of a matrix we constructed, where the $(i, j)^{\text{th}}$entry of the matrix is the great-circle distance between the voxels $i$ and $j$, after retaining only 0.1% of closest voxels. Both sparsity patterns are visually similar, suggesting that the best HP-CONCORD estimate has recovered some of the spatial

[1] Recently, a preprint of work that compares a large set of parcellations became available [9]. As a future work, a comparison of our approach to those in this paper would shed more insight into our parcellation.

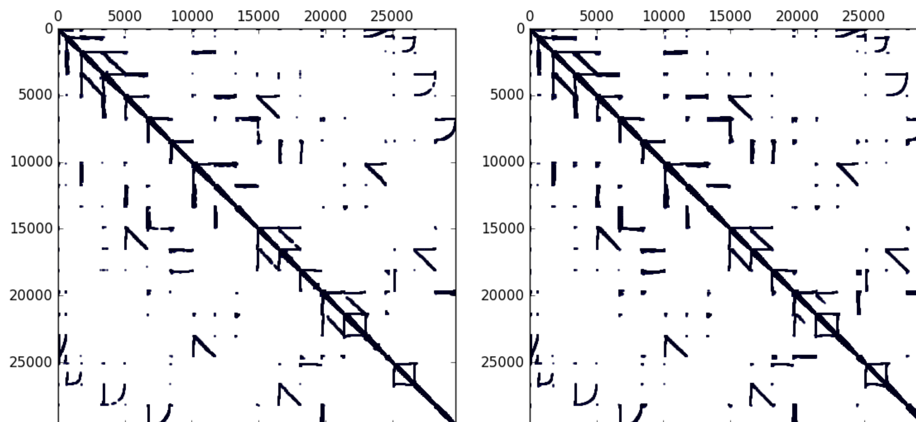


Figure 7.8: Left: the sparsity pattern of the HP-CONCORD estimate attaining the best clustering in the left column of Table 7.3. Right: the sparsity pattern of a $(91,282 \times 91,282)$-dimensional (symmetric) matrix we constructed, whose $(i, j)$th entry is the great-circle distance between the voxels $i$ and $j$, where we have retained just the 0.1% of closest voxels. In both plots, black indicates a nonzero entry and only the 29,696 coordinates belonging to the left hemisphere are shown.

signal in the data, without being "told" to do so. Inspecting the right hemisphere conveys the same message.

**Full experimental results.** This subsection provides an expanded set of figures and tables from the experiments. First, we present the sparsity patterns of the estimates from HP-CONCORD and their degrees of connectivity mapping directly on the brain. Tables 7.5, 7.6, and 7.7 show the sparsity patterns for the whole brain, the left hemisphere, and the right hemisphere, respectively. Tables 7.8 and 7.9 show the connectivity mapping on the left and right hemisphere.

We show our clustering results with (1) various penalty values of HP-CONCORD: $\lambda1$ and $\lambda2$, (2) two parts of the brain: left and right hemisphere, (3) two clustering methods, taking in the partial correlation graph from HP-CONCORD as inputs: persistent homology and Louvain methods, and (4) two clustering coarseness levels: more clusters and fewer clusters. Table 7.4 summarizes them in one table we have. We also present several clustering results from directly thresholding the sample covariance matrix in Tables 7.18 and 7.19.

| Method \ Brain part | | Left hemisphere | | Right hemisphere | |
|---|---|---|---|---|---|
| | | Clusterings | Jaccard scores | Clusterings | Jaccard scores |
| Persistent homology | Fewer clusters ($\epsilon = 3$) | Table 7.10 | Table 7.20 | Table 7.11 | Table 7.21 |
| | More clusters ($\epsilon = 0$) | Table 7.12 | Table 7.22 | Table 7.13 | Table 7.23 |
| Louvain method | Fewer clusters ($k = 0$) | Table 7.14 | Table 7.24 | Table 7.15 | Table 7.25 |
| | More clusters ($k_{\max}$) | Table 7.16 | Table 7.26 | Table 7.17 | Table 7.27 |

Table 7.4: Summarizing tables corresponding to various clustering experiments.

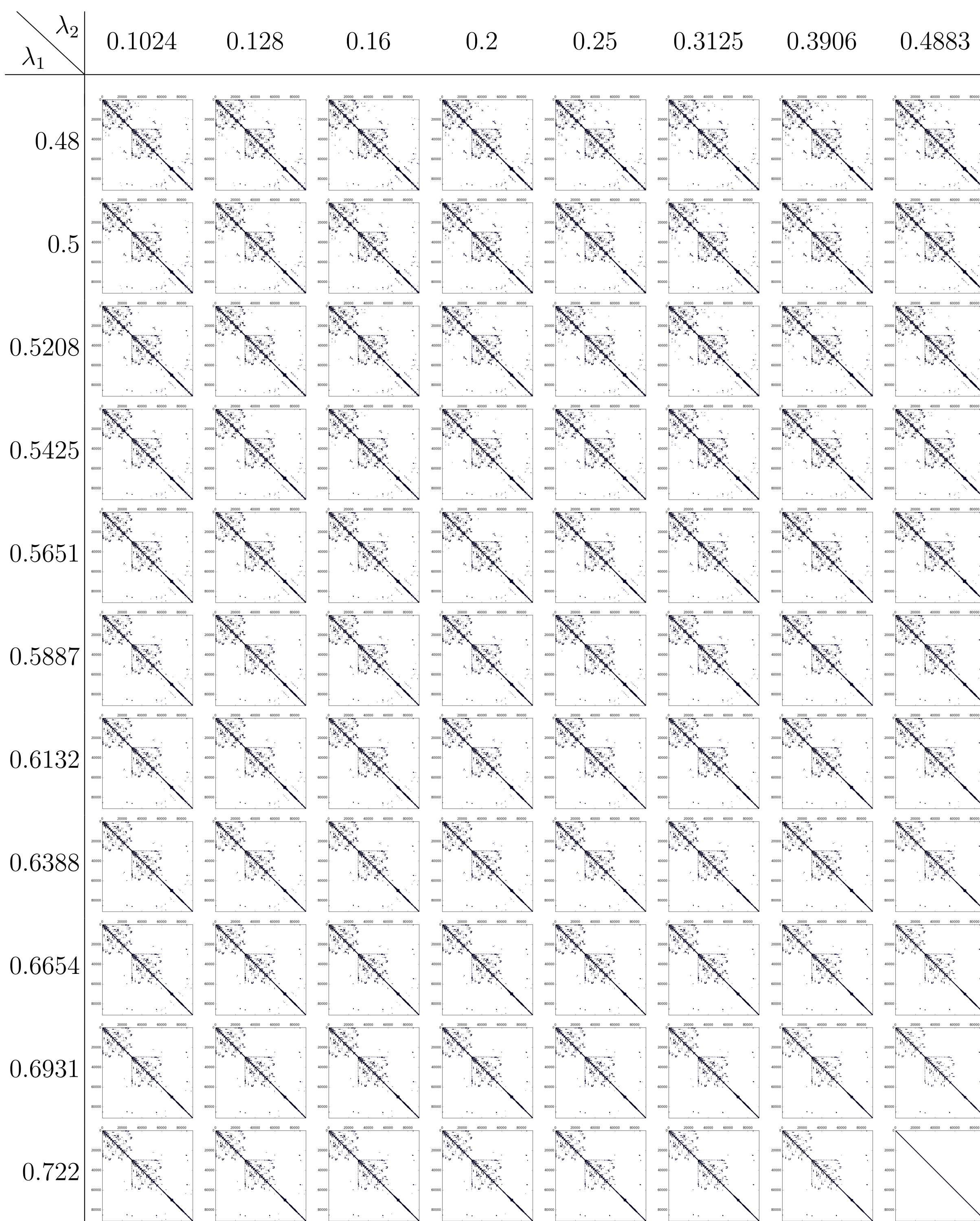



Table 7.5: Sparsity patterns of the whole brain.

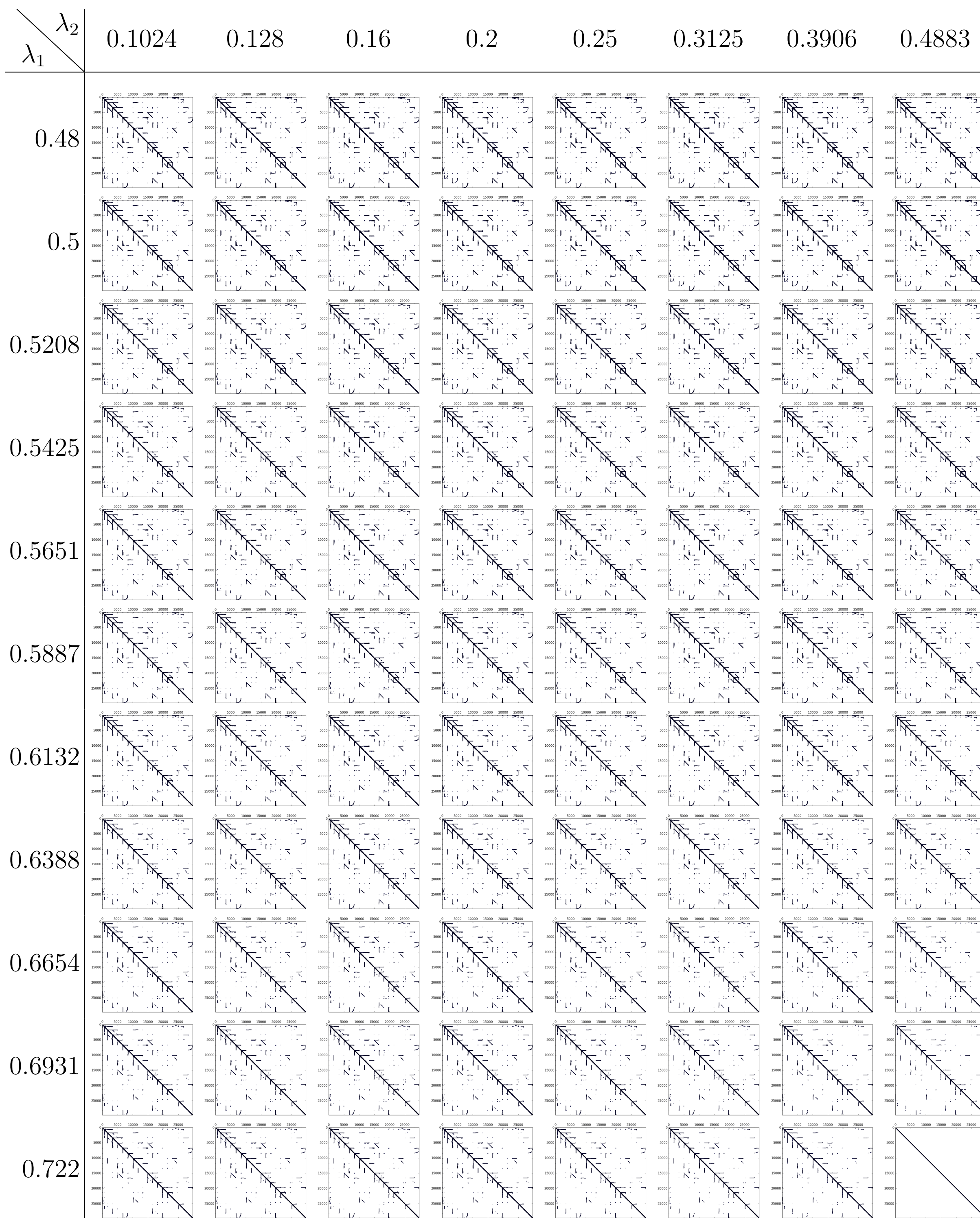



Table 7.6: Sparsity patterns of the left hemisphere.

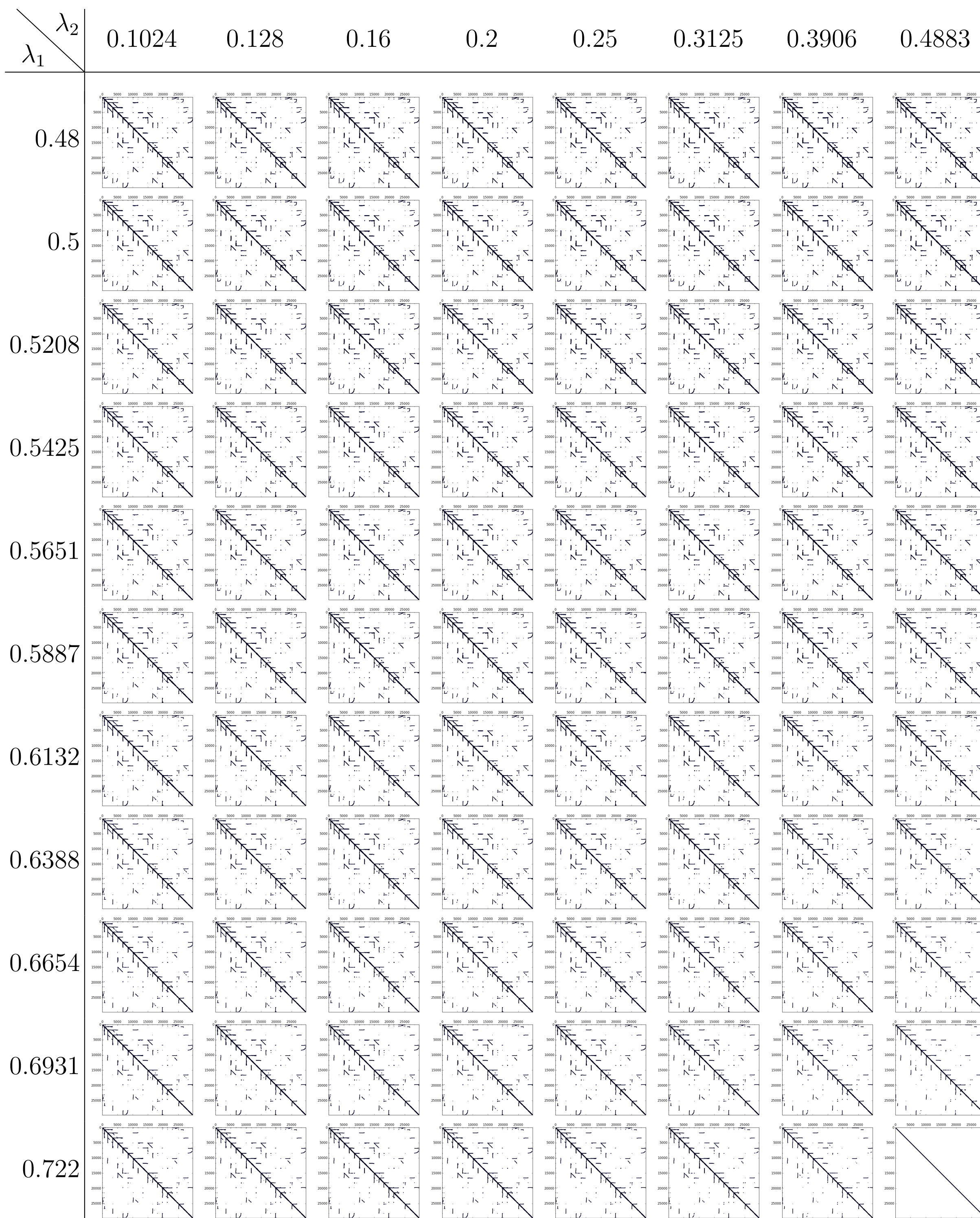



Table 7.7: Sparsity patterns of the right hemisphere.

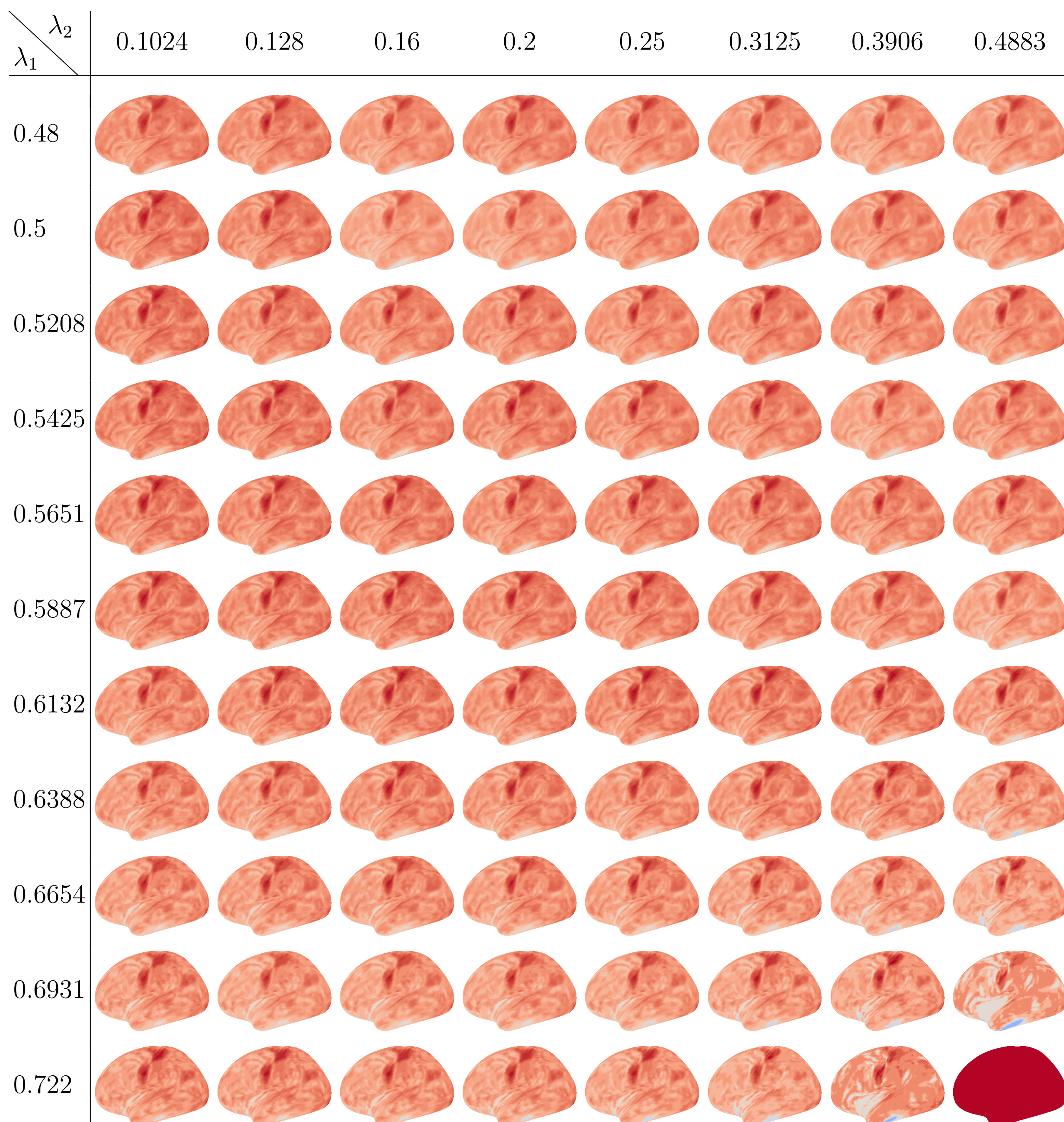



Table 7.8: Degree matrices of the left hemisphere. The point at $(0.722, 0.4883)$ is a degenerated clustering that puts each voxel into its own cluster.

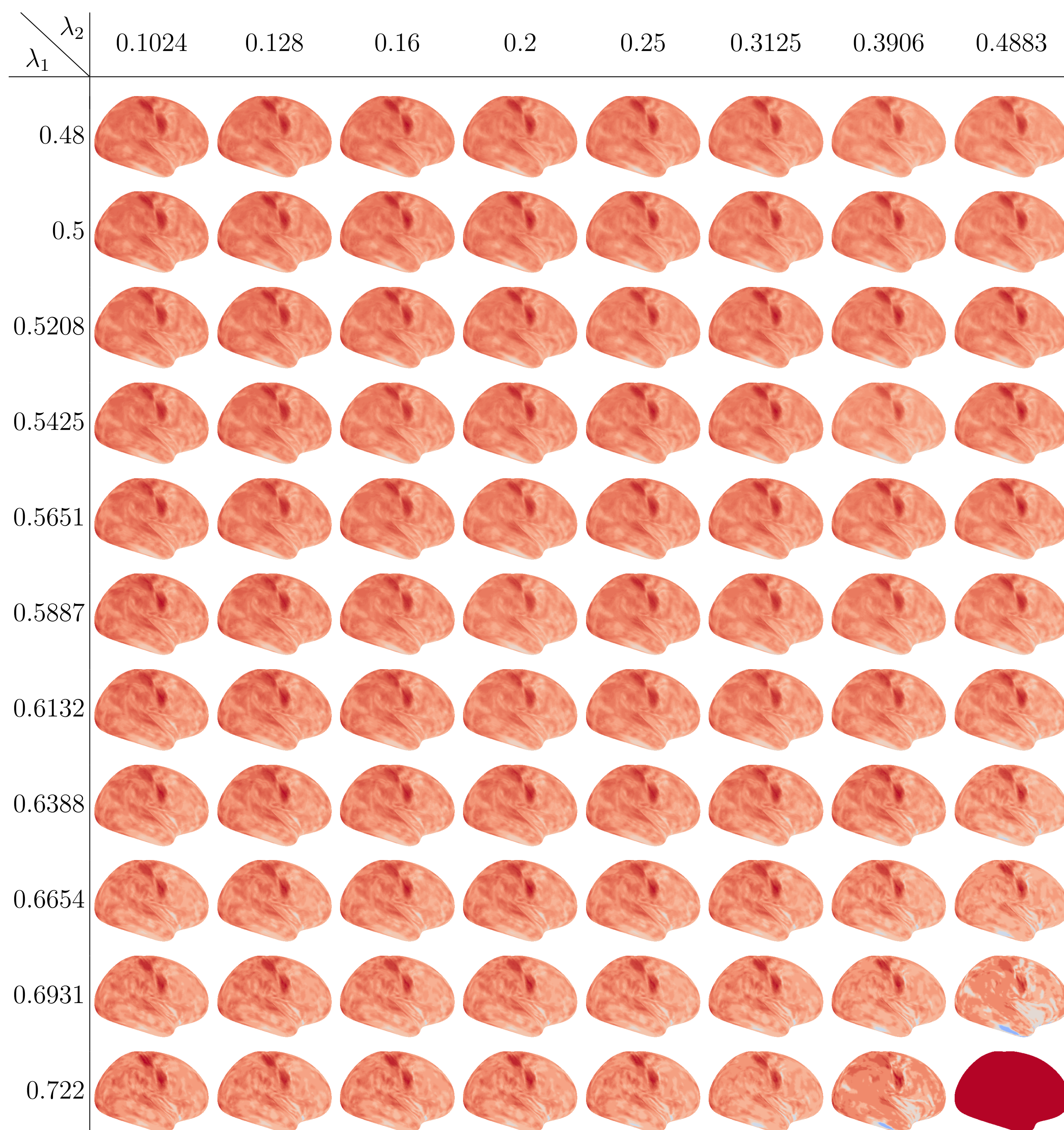



Table 7.9: Degree matrices of the right hemisphere. The point at $(0.722, 0.4883)$ is a degenerated clustering that puts each voxel into its own cluster.

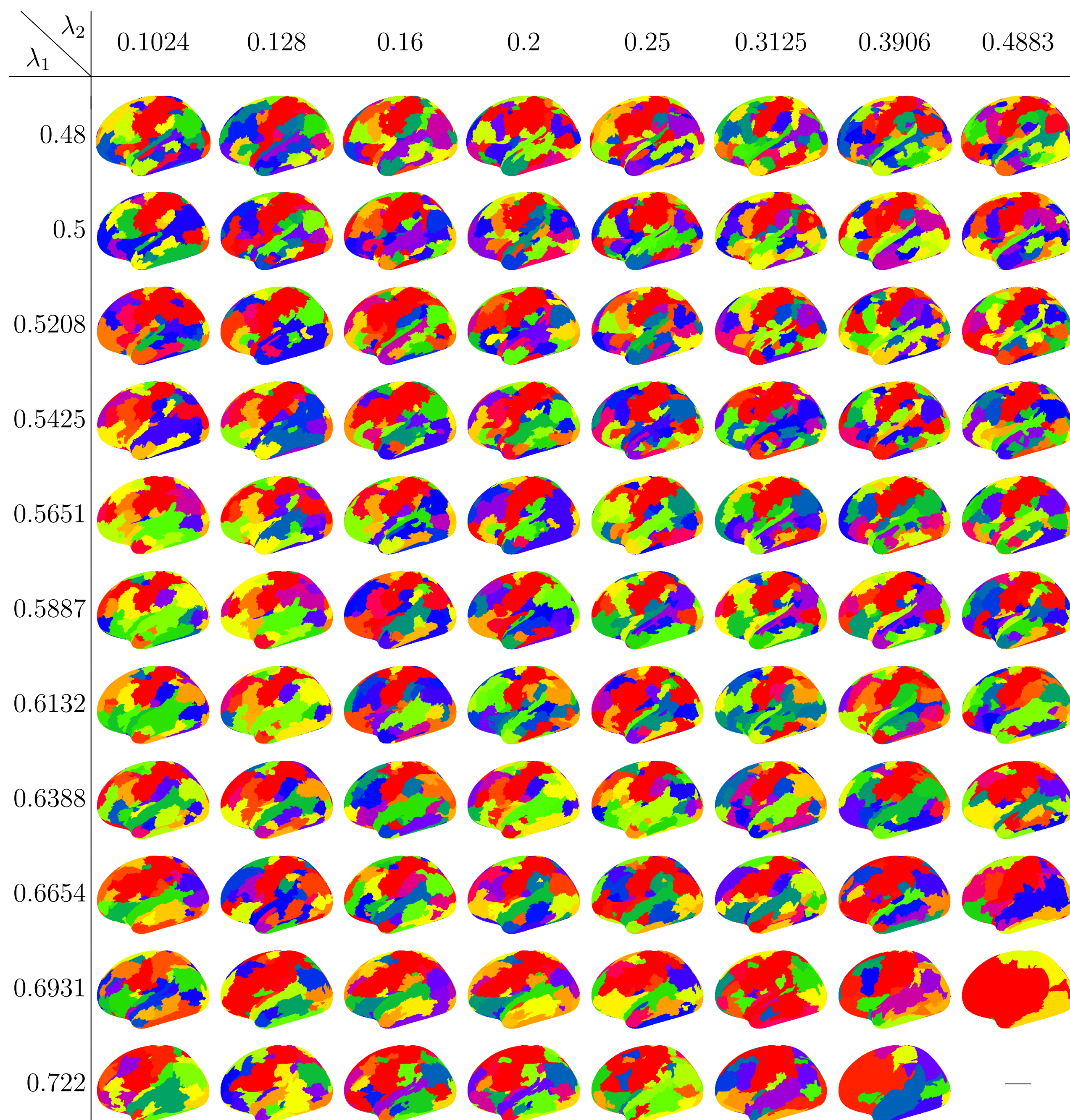



Table 7.10: *Left* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method with $\varepsilon = 3$ (corresponding to *fewer* clusters), as described in Section 7.4. Table 7.20 presents the modified Jaccard scores (7.12) for these clusterings. "—" indicates a degenerated clustering that puts each voxel into its own cluster.

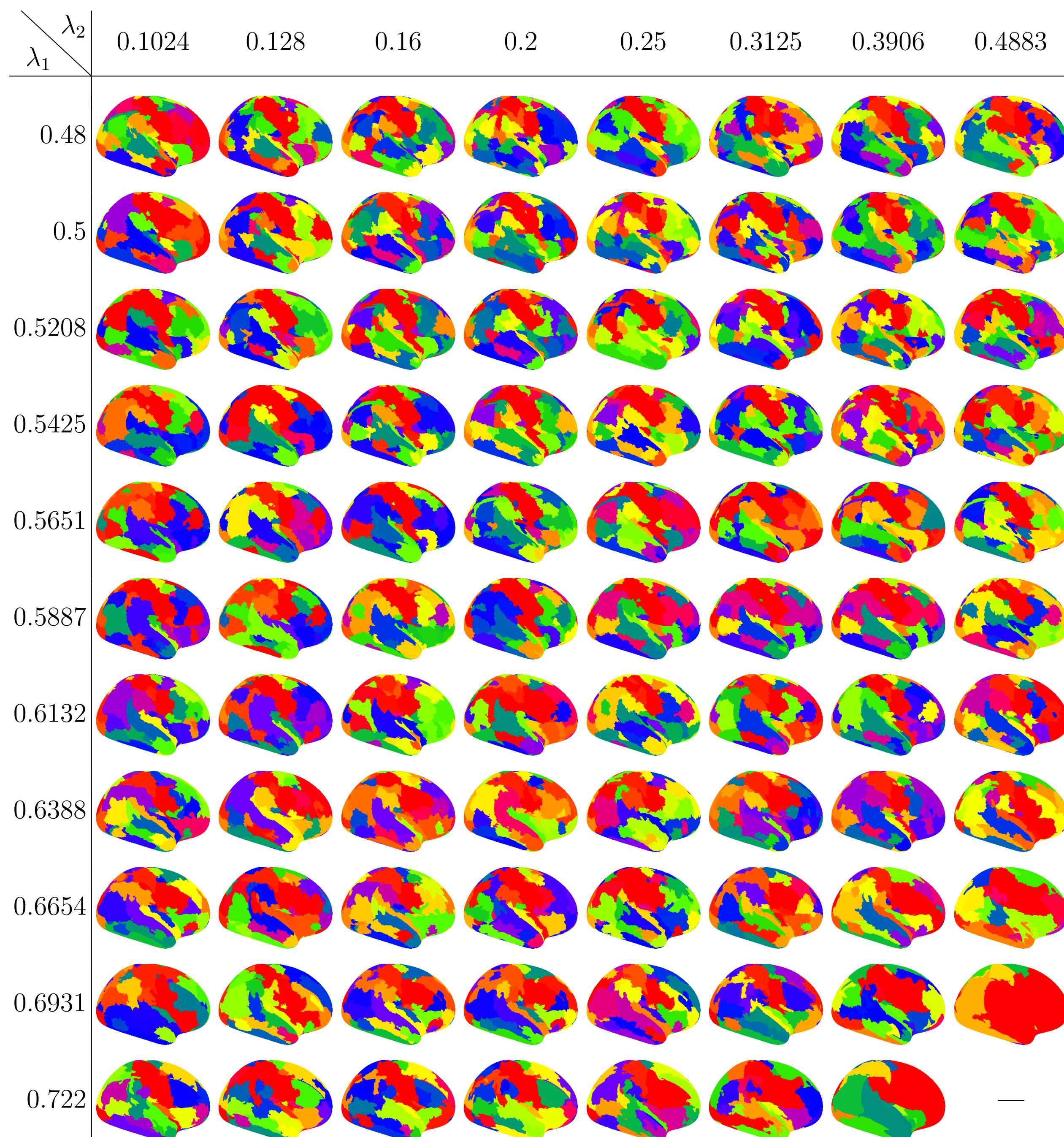



Table 7.11: *Right* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method with $\varepsilon = 3$ (corresponding to *fewer* clusters), as described in Section 7.4. Table 7.21 presents the modified Jaccard scores (7.12) for these clusterings. "—" indicates a degenerated clustering that puts each voxel into its own cluster.

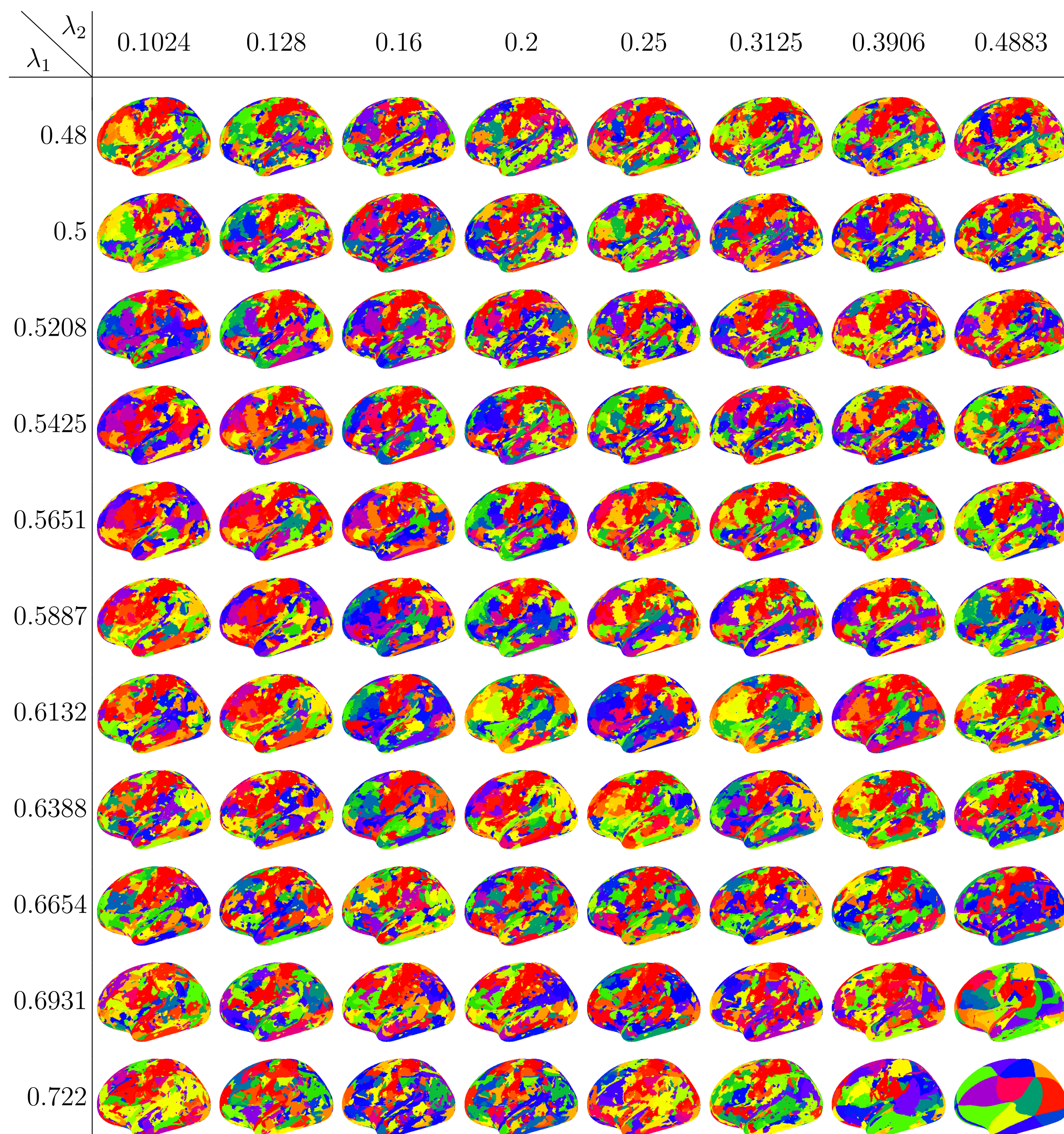



Table 7.12: *Left* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method with $\varepsilon = 0$ (corresponding to *more* clusters), as described in Section 7.4. Table 7.22 presents the modified Jaccard scores (7.12) for these clusterings.

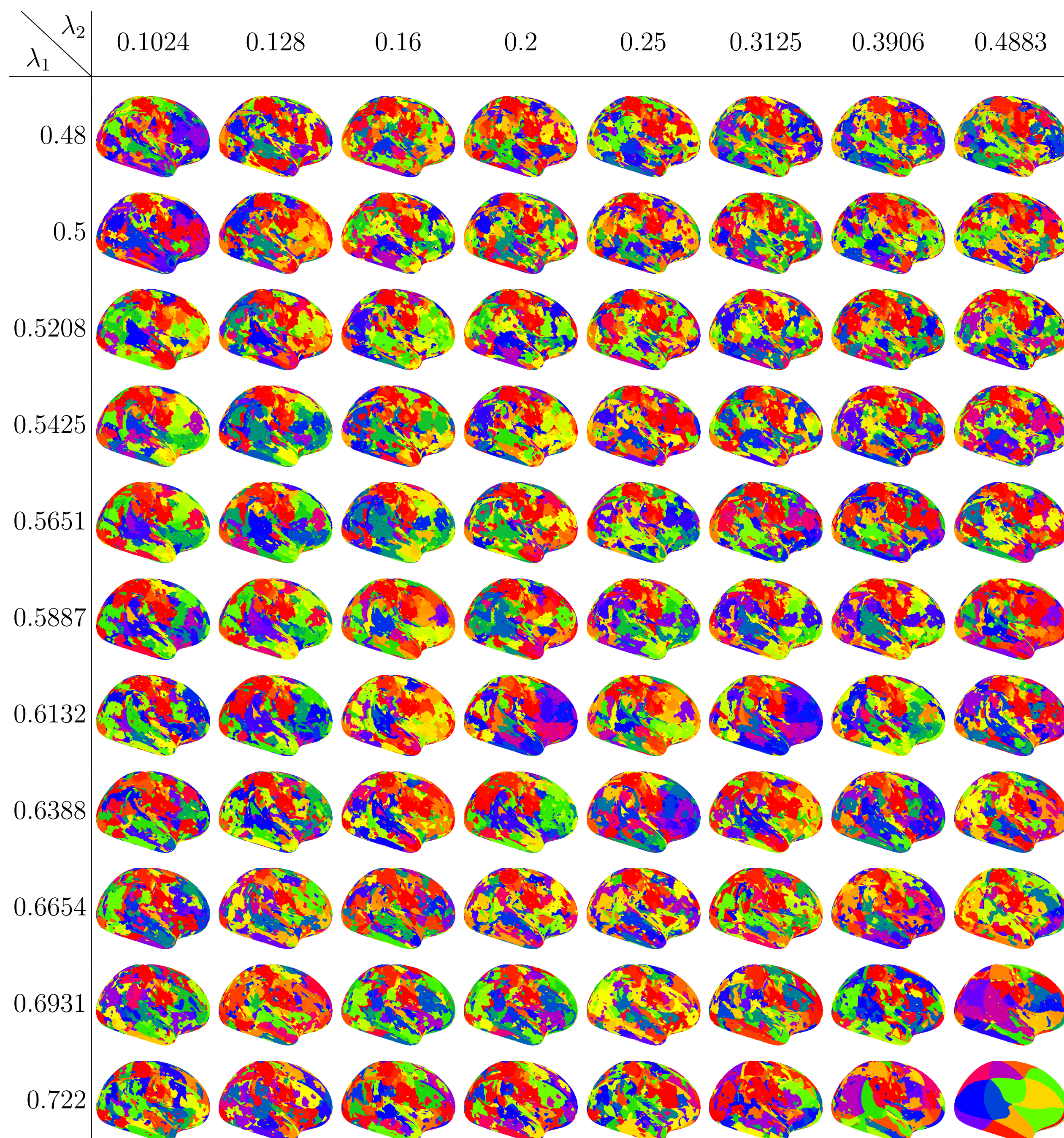



Table 7.13: *Right* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method with $\varepsilon = 0$ (corresponding to *more* clusters), as described in Section 7.4. Table 7.23 presents the modified Jaccard scores (7.12) for these clusterings.

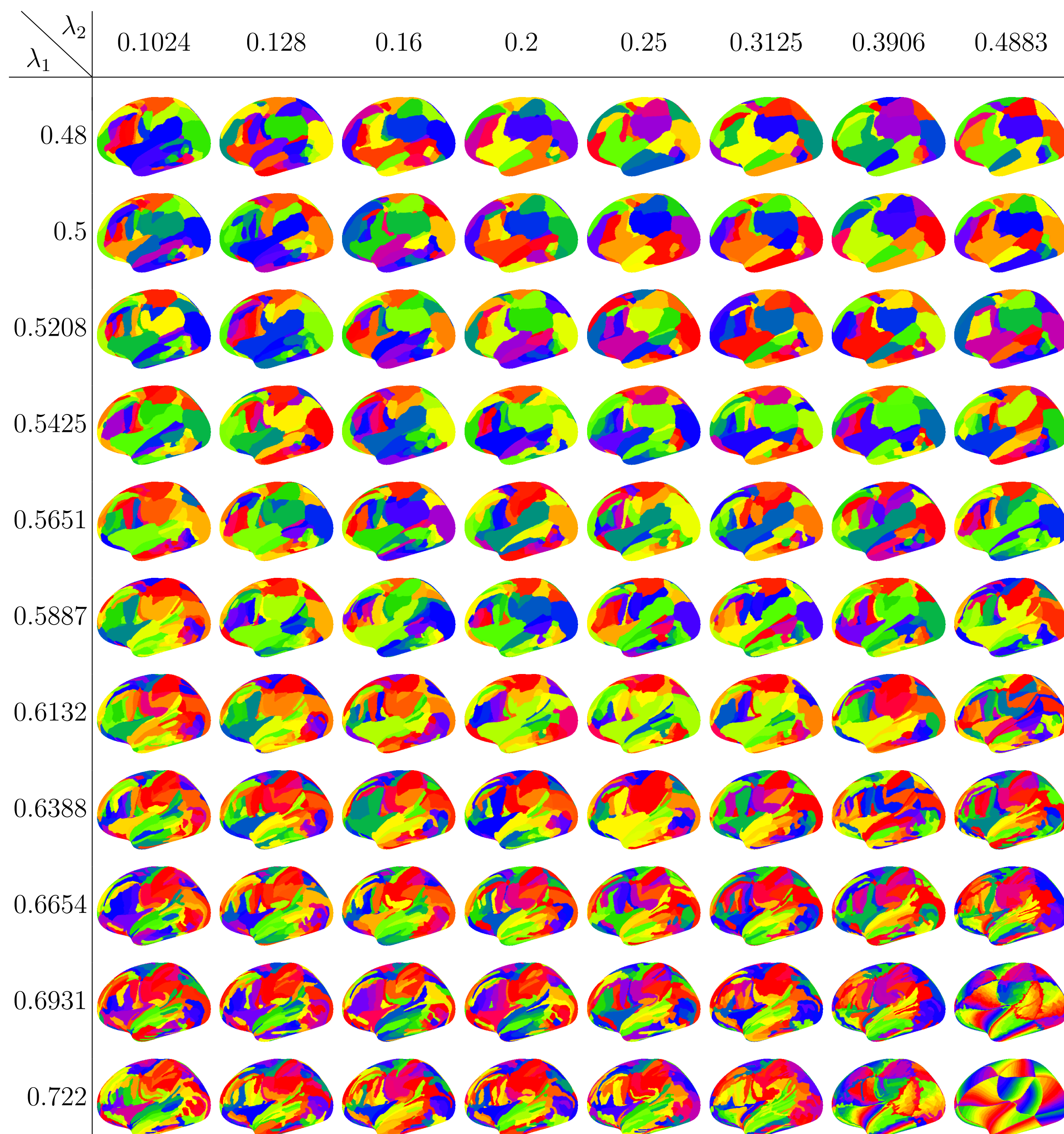



Table 7.14: *Left* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *Louvain* method with $k = 0$ (corresponding to *fewer* clusters), as described in Section 7.4. Table 7.24 presents the modified Jaccard scores (7.12) for these clusterings.

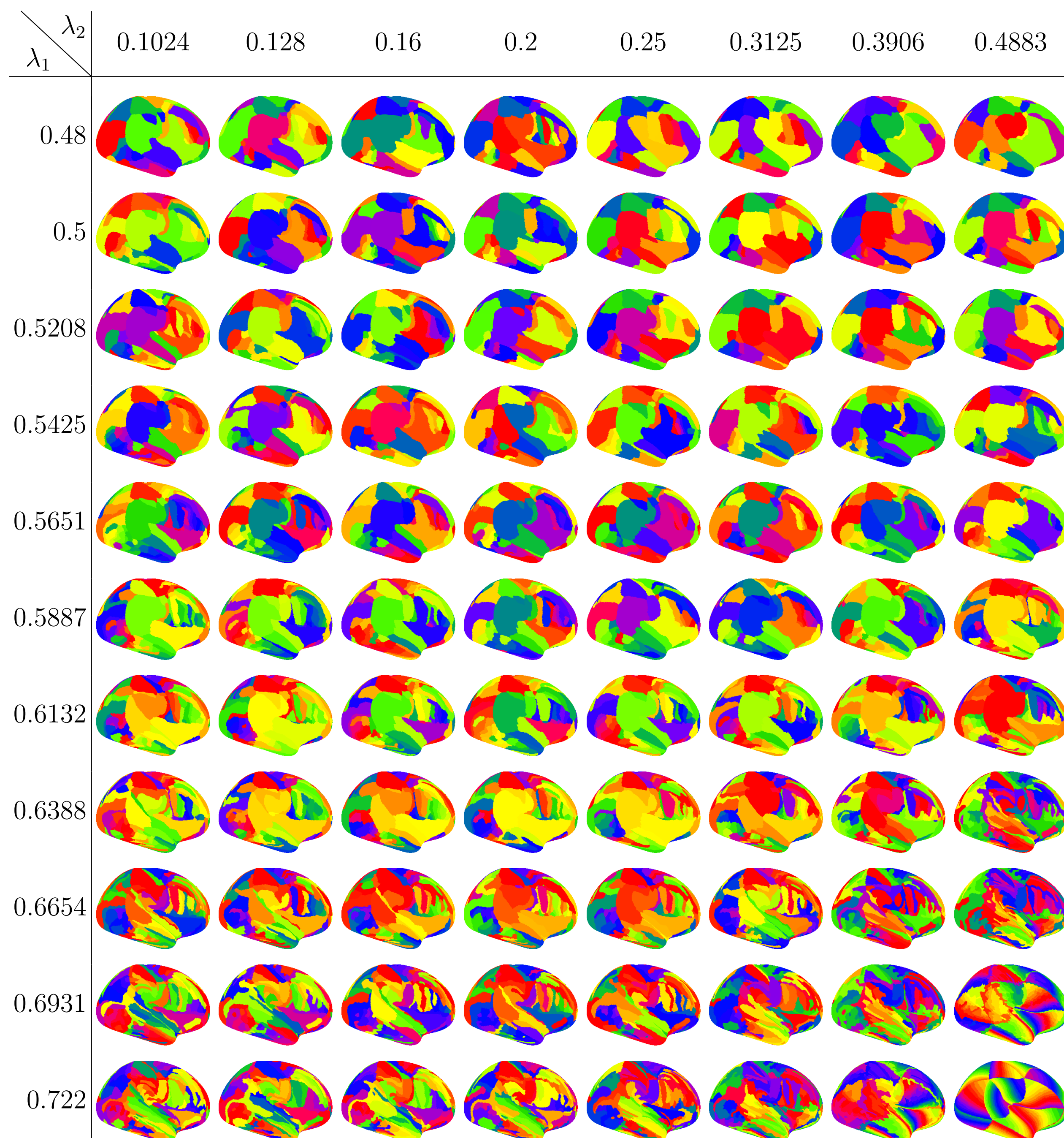



Table 7.15: *Right* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *Louvain* method with $k = 0$ (corresponding to *fewer* clusters), as described in Section 7.4. Table 7.25 presents the modified Jaccard scores (7.12) for these clusterings.

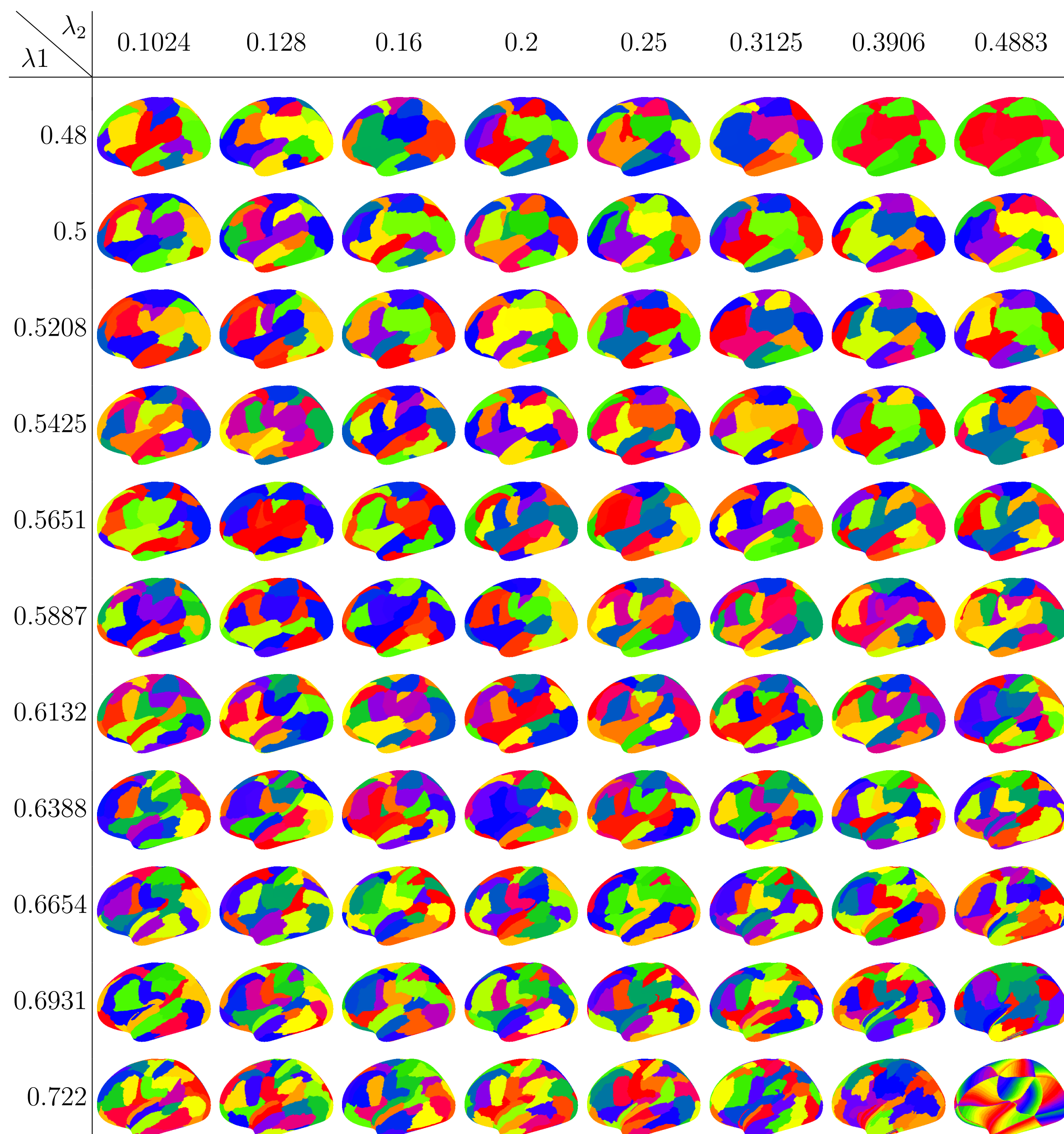



Table 7.16: *Left* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *Louvain* method with the largest parameter value $k$ considered by Louvain (corresponding to *more* clusters), as described in Section 7.4. Table 7.26 presents the modified Jaccard scores (7.12) for these clusterings.

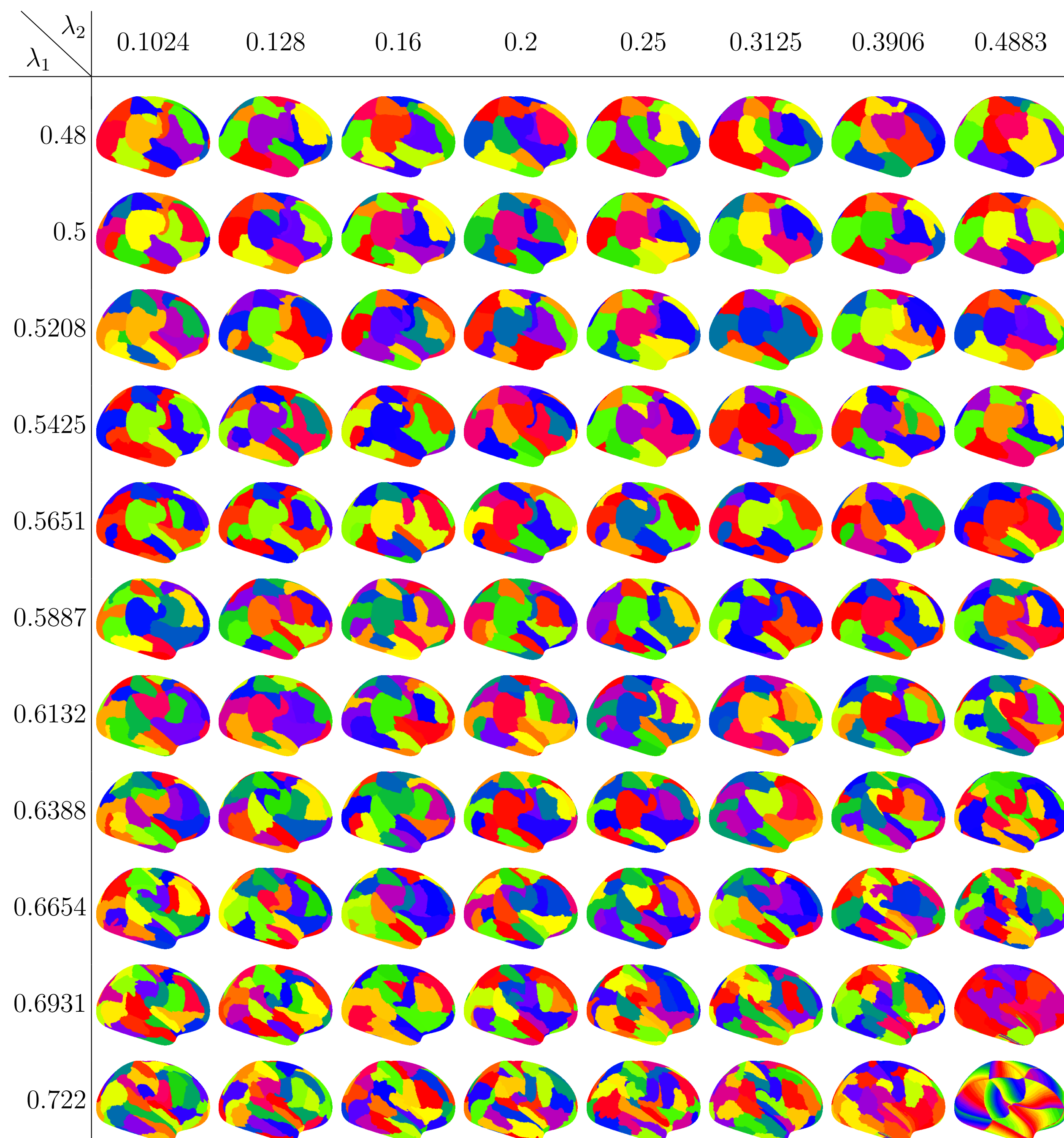



Table 7.17: *Right* hemisphere clusterings, generated by HP-CONCORD using various $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *Louvain* method with the largest parameter value $k$ considered by Louvain (corresponding to *more* clusters), as described in Section 7.4. Table 7.27 presents the modified Jaccard scores (7.12) for these clusterings.

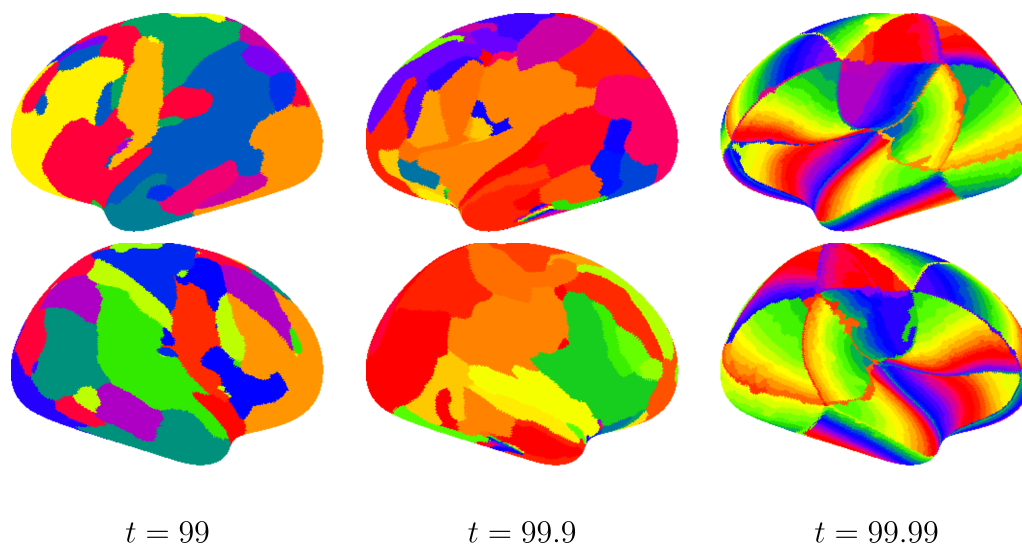



Table 7.18: *Left* and *right* hemisphere clusterings (in the top and bottom row, respectively), generated by thresholding the sample covariance matrix at various levels $t$ followed by the *Louvain* method, at the tuning parameter value $k = 0$ (corresponding to *fewer* clusters), as described in Section 7.4.

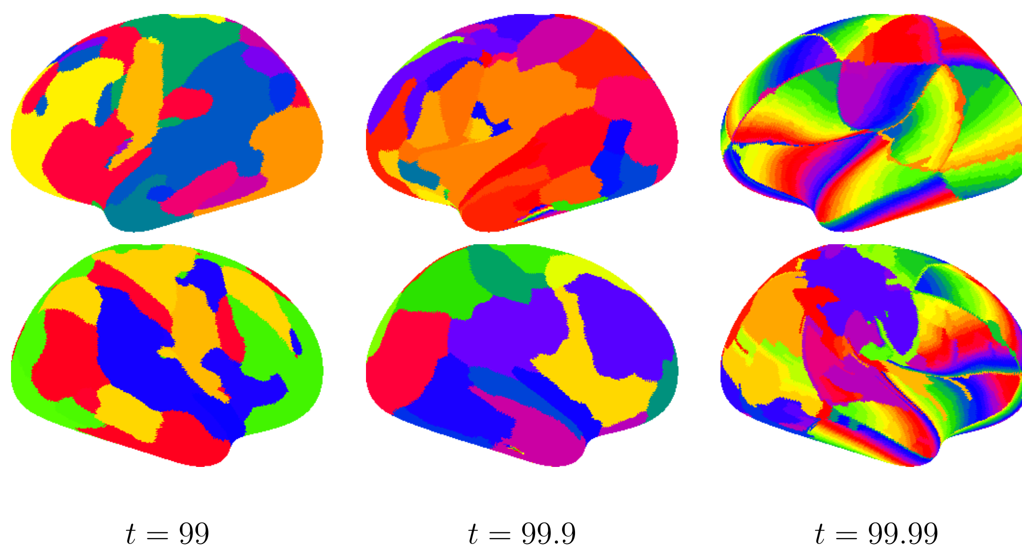



Table 7.19: *Left* and *right* hemisphere clusterings (in the top and bottom row, respectively), generated by thresholding the sample covariance matrix at various levels $t$, followed by the *Louvain* method at the largest tuning parameter value $k$ considered by Louvain (corresponding to *more* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.2043 | 0.2199 | 0.2242 | 0.2326 | 0.2277 | 0.2422 | 0.2447 | 0.23 |
| 0.5 | 0.2112 | 0.224 | 0.2315 | 0.2329 | 0.2343 | 0.2283 | 0.2197 | 0.2185 |
| 0.5208 | 0.1964 | 0.1895 | 0.2264 | 0.2385 | 0.2317 | 0.2282 | 0.2348 | 0.2358 |
| 0.5425 | 0.1905 | 0.1972 | 0.1951 | 0.2181 | 0.2268 | 0.2295 | 0.2289 | 0.2255 |
| 0.5651 | 0.1833 | 0.197 | 0.1973 | 0.1981 | 0.2125 | 0.2268 | 0.2242 | 0.2213 |
| 0.5887 | 0.1838 | 0.1845 | 0.1992 | 0.2067 | 0.1953 | 0.2057 | 0.2078 | 0.2155 |
| 0.6132 | 0.1702 | 0.1752 | 0.198 | 0.1995 | 0.2121 | 0.2014 | 0.2036 | 0.1891 |
| 0.6388 | 0.1698 | 0.1693 | 0.1864 | 0.1837 | 0.1859 | 0.191 | 0.1831 | 0.1785 |
| 0.6654 | 0.1538 | 0.1854 | 0.1759 | 0.1701 | 0.1748 | 0.1844 | 0.1805 | 0.1467 |
| 0.6931 | 0.1652 | 0.1689 | 0.1664 | 0.1686 | 0.1722 | 0.162 | 0.1472 | 0.0516 |
| 0.722 | 0.1382 | 0.1536 | 0.1556 | 0.1536 | 0.1442 | 0.1394 | 0.0758 | — |

Table 7.20: The Jaccard scores (7.12) for the clusterings of the *left* hemisphere in Table 7.10, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method at the tuning parameter $\varepsilon = 3$ (corresponding to *fewer* clusters), as described in Section 7.4. "—" indicates a degenerate clustering that puts each voxel into its own cluster.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.2258 | 0.2315 | 0.2461 | 0.2279 | 0.2451 | 0.2436 | 0.2311 | 0.2431 |
| 0.5 | 0.2036 | 0.2245 | 0.2328 | 0.2326 | 0.2427 | 0.2314 | 0.2654 | 0.2528 |
| 0.5208 | 0.2255 | 0.2166 | 0.2317 | 0.2311 | 0.2427 | 0.2399 | 0.2381 | 0.2417 |
| 0.5425 | 0.21 | 0.2172 | 0.232 | 0.2355 | 0.2279 | 0.2299 | 0.245 | 0.2349 |
| 0.5651 | 0.2233 | 0.2182 | 0.2236 | 0.2341 | 0.2367 | 0.231 | 0.2286 | 0.2413 |
| 0.5887 | 0.2055 | 0.2187 | 0.2179 | 0.2369 | 0.2261 | 0.2321 | 0.2279 | 0.2067 |
| 0.6132 | 0.1843 | 0.2002 | 0.2245 | 0.2224 | 0.2113 | 0.219 | 0.2256 | 0.21 |
| 0.6388 | 0.1817 | 0.1843 | 0.2024 | 0.204 | 0.2154 | 0.2161 | 0.1981 | 0.1826 |
| 0.6654 | 0.1786 | 0.1678 | 0.1824 | 0.1891 | 0.1952 | 0.1749 | 0.1851 | 0.1273 |
| 0.6931 | 0.1652 | 0.1714 | 0.1686 | 0.1736 | 0.1714 | 0.1702 | 0.1284 | 0.061 |
| 0.722 | 0.1372 | 0.1562 | 0.162 | 0.1563 | 0.1364 | 0.1264 | 0.0875 | — |

Table 7.21: The Jaccard scores (7.12) for the clusterings of the *right* hemisphere in Table 7.11, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method at the tuning parameter $\varepsilon = 3$ (corresponding to *fewer* clusters), as described in Section 7.4. "—" indicates a degenerate clustering that puts each voxel into its own cluster.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.0507 | 0.051 | 0.0527 | 0.0532 | 0.0511 | 0.0503 | 0.051 | 0.0518 |
| 0.5 | 0.053 | 0.0518 | 0.052 | 0.0519 | 0.0526 | 0.0531 | 0.0517 | 0.0516 |
| 0.5208 | 0.0517 | 0.0519 | 0.0527 | 0.0517 | 0.0524 | 0.0526 | 0.0522 | 0.0536 |
| 0.5425 | 0.0522 | 0.0519 | 0.0509 | 0.0516 | 0.0516 | 0.0514 | 0.0532 | 0.0533 |
| 0.5651 | 0.0514 | 0.0524 | 0.0512 | 0.0528 | 0.0529 | 0.0518 | 0.0518 | 0.0533 |
| 0.5887 | 0.0498 | 0.0524 | 0.0534 | 0.0521 | 0.0522 | 0.0521 | 0.0526 | 0.0532 |
| 0.6132 | 0.0504 | 0.0501 | 0.0531 | 0.052 | 0.0523 | 0.0529 | 0.0523 | 0.0505 |
| 0.6388 | 0.053 | 0.052 | 0.0494 | 0.0502 | 0.0517 | 0.0502 | 0.0516 | 0.0543 |
| 0.6654 | 0.0526 | 0.0529 | 0.0536 | 0.0533 | 0.0537 | 0.0506 | 0.0535 | 0.0558 |
| 0.6931 | 0.0529 | 0.054 | 0.0518 | 0.052 | 0.0532 | 0.0543 | 0.0566 | 0.0815 |
| 0.722 | 0.0549 | 0.0528 | 0.0525 | 0.0534 | 0.056 | 0.0561 | 0.0718 | 0.0884 |

Table 7.22: The Jaccard scores (7.12) for the clusterings of the *left* hemisphere in Table 7.12, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method at the tuning parameter $\varepsilon = 0$ (corresponding to *more* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.0532 | 0.0506 | 0.0519 | 0.0516 | 0.0522 | 0.0521 | 0.0516 | 0.0508 |
| 0.5 | 0.0538 | 0.0536 | 0.0513 | 0.0512 | 0.0515 | 0.0525 | 0.0524 | 0.0506 |
| 0.5208 | 0.052 | 0.0519 | 0.0521 | 0.0509 | 0.0527 | 0.0517 | 0.0522 | 0.0509 |
| 0.5425 | 0.0505 | 0.0523 | 0.0528 | 0.0532 | 0.0511 | 0.0529 | 0.0526 | 0.0522 |
| 0.5651 | 0.0523 | 0.0501 | 0.0513 | 0.0513 | 0.053 | 0.0521 | 0.0512 | 0.0528 |
| 0.5887 | 0.0516 | 0.0528 | 0.0504 | 0.0515 | 0.0518 | 0.0515 | 0.0523 | 0.0511 |
| 0.6132 | 0.0505 | 0.0517 | 0.0525 | 0.0534 | 0.0511 | 0.0516 | 0.0543 | 0.0534 |
| 0.6388 | 0.0514 | 0.0543 | 0.0516 | 0.0522 | 0.0519 | 0.0533 | 0.0532 | 0.0544 |
| 0.6654 | 0.0544 | 0.0528 | 0.0514 | 0.0518 | 0.0525 | 0.0529 | 0.0565 | 0.061 |
| 0.6931 | 0.0527 | 0.0555 | 0.0525 | 0.0528 | 0.055 | 0.0529 | 0.0597 | 0.0845 |
| 0.722 | 0.0535 | 0.0524 | 0.0535 | 0.0527 | 0.0553 | 0.0566 | 0.0698 | 0.0918 |

Table 7.23: The Jaccard scores (7.12) for the clusterings of the *right* hemisphere in Table 7.13, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method at the tuning parameter $\varepsilon = 0$ (corresponding to *more* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda 2_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.1069 | 0.1101 | 0.0956 | 0.1042 | 0.1015 | 0.0958 | 0.0901 | 0.0905 |
| 0.5 | 0.1123 | 0.1076 | 0.1065 | 0.102 | 0.1089 | 0.1053 | 0.1007 | 0.107 |
| 0.5208 | 0.1097 | 0.111 | 0.1092 | 0.1096 | 0.1065 | 0.0982 | 0.1001 | 0.105 |
| 0.5425 | 0.1313 | 0.1123 | 0.1148 | 0.1085 | 0.1166 | 0.1143 | 0.1065 | 0.117 |
| 0.5651 | 0.1258 | 0.1216 | 0.1134 | 0.1167 | 0.1164 | 0.1097 | 0.1151 | 0.12 |
| 0.5887 | 0.129 | 0.1228 | 0.1233 | 0.1091 | 0.1203 | 0.1205 | 0.1238 | 0.1188 |
| 0.6132 | 0.1337 | 0.1294 | 0.1298 | 0.1289 | 0.1185 | 0.1285 | 0.1231 | 0.1455 |
| 0.6388 | 0.1477 | 0.1368 | 0.1363 | 0.1296 | 0.131 | 0.1344 | 0.1473 | 0.1517 |
| 0.6654 | 0.1486 | 0.1486 | 0.1458 | 0.1405 | 0.1488 | 0.1534 | 0.1486 | 0.1583 |
| 0.6931 | 0.1469 | 0.1453 | 0.1512 | 0.1483 | 0.146 | 0.1627 | 0.1706 | 0.0273 |
| 0.722 | 0.1581 | 0.1608 | 0.1557 | 0.1608 | 0.1661 | 0.1779 | 0.0461 | 0.0061 |

Table 7.24: The Jaccard scores (7.12) for the clusterings of the *left* hemisphere in Table 7.14, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method *Louvain* method at the tuning parameter value $k = 0$ (corresponding to *fewer* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.1105 | 0.1014 | 0.1034 | 0.1042 | 0.0988 | 0.0976 | 0.0976 | 0.0935 |
| 0.5 | 0.1084 | 0.1064 | 0.1011 | 0.1105 | 0.1003 | 0.1012 | 0.0992 | 0.1022 |
| 0.5208 | 0.1263 | 0.1059 | 0.1195 | 0.1056 | 0.0995 | 0.1059 | 0.1 | 0.1051 |
| 0.5425 | 0.122 | 0.1167 | 0.1113 | 0.1111 | 0.0997 | 0.1085 | 0.1125 | 0.0997 |
| 0.5651 | 0.1219 | 0.1212 | 0.1144 | 0.1022 | 0.1044 | 0.1109 | 0.1029 | 0.1189 |
| 0.5887 | 0.1218 | 0.1205 | 0.1184 | 0.1219 | 0.1159 | 0.1202 | 0.1183 | 0.135 |
| 0.6132 | 0.132 | 0.1259 | 0.1339 | 0.1265 | 0.1269 | 0.124 | 0.1294 | 0.1361 |
| 0.6388 | 0.1362 | 0.1364 | 0.1289 | 0.1286 | 0.1318 | 0.1279 | 0.1357 | 0.158 |
| 0.6654 | 0.1483 | 0.1451 | 0.1428 | 0.142 | 0.1438 | 0.1498 | 0.1626 | 0.1675 |
| 0.6931 | 0.1518 | 0.1552 | 0.1451 | 0.1473 | 0.1552 | 0.1671 | 0.1736 | 0.027 |
| 0.722 | 0.1648 | 0.1725 | 0.1556 | 0.1607 | 0.1643 | 0.1758 | 0.0482 | 0.0061 |

Table 7.25: The Jaccard scores (7.12) for the clusterings of the *right* hemisphere in Table 7.15, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method *Louvain* method at the tuning parameter value $k = 0$ (corresponding to *fewer* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.1678 | 0.1666 | 0.154 | 0.14 | 0.1297 | 0.135 | 0.1311 | 0.1284 |
| 0.5 | 0.1632 | 0.1778 | 0.1595 | 0.1515 | 0.1537 | 0.1454 | 0.128 | 0.1422 |
| 0.5208 | 0.1578 | 0.1719 | 0.1589 | 0.1663 | 0.1604 | 0.1572 | 0.145 | 0.1609 |
| 0.5425 | 0.1538 | 0.1572 | 0.166 | 0.1542 | 0.1702 | 0.1689 | 0.1684 | 0.1569 |
| 0.5651 | 0.1503 | 0.1602 | 0.1541 | 0.1473 | 0.1561 | 0.1587 | 0.1502 | 0.155 |
| 0.5887 | 0.1526 | 0.158 | 0.1537 | 0.1622 | 0.1564 | 0.1547 | 0.149 | 0.1338 |
| 0.6132 | 0.1438 | 0.1425 | 0.154 | 0.1487 | 0.151 | 0.1489 | 0.1327 | 0.1191 |
| 0.6388 | 0.1414 | 0.1453 | 0.134 | 0.1431 | 0.1393 | 0.1357 | 0.1238 | 0.0967 |
| 0.6654 | 0.1252 | 0.1263 | 0.1403 | 0.1301 | 0.1279 | 0.1196 | 0.0987 | 0.0653 |
| 0.6931 | 0.1137 | 0.1159 | 0.1161 | 0.1163 | 0.109 | 0.0937 | 0.0701 | 0.0216 |
| 0.722 | 0.1008 | 0.1005 | 0.1015 | 0.0961 | 0.0891 | 0.0679 | 0.0298 | 0.0061 |

Table 7.26: The Jaccard scores (7.12) for the clusterings of the *left* hemisphere in Table 7.16, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method *Louvain* method at the largest tuning parameter value $k$ considered by Louvain (corresponding to *more* clusters), as described in Section 7.4.

| $\lambda_1$ \ $\lambda_2$ | 0.1024 | 0.128 | 0.16 | 0.2 | 0.25 | 0.3125 | 0.3906 | 0.4883 |
|---|---|---|---|---|---|---|---|---|
| 0.48 | 0.1719 | 0.1697 | 0.1633 | 0.1763 | 0.1499 | 0.1475 | 0.1442 | 0.1365 |
| 0.5 | 0.1689 | 0.1675 | 0.167 | 0.17 | 0.1661 | 0.1587 | 0.1411 | 0.1729 |
| 0.5208 | 0.1651 | 0.1581 | 0.1808 | 0.1694 | 0.1655 | 0.1528 | 0.153 | 0.1512 |
| 0.5425 | 0.1697 | 0.1556 | 0.1634 | 0.1637 | 0.1591 | 0.1581 | 0.1857 | 0.1598 |
| 0.5651 | 0.1651 | 0.1663 | 0.1509 | 0.1554 | 0.1567 | 0.1542 | 0.151 | 0.1416 |
| 0.5887 | 0.1492 | 0.1602 | 0.1635 | 0.1541 | 0.1512 | 0.1586 | 0.1506 | 0.1536 |
| 0.6132 | 0.1474 | 0.1596 | 0.1586 | 0.1593 | 0.1649 | 0.1548 | 0.1436 | 0.1168 |
| 0.6388 | 0.1321 | 0.1337 | 0.1495 | 0.1502 | 0.1458 | 0.1272 | 0.1188 | 0.0938 |
| 0.6654 | 0.119 | 0.1203 | 0.1233 | 0.1221 | 0.1185 | 0.1125 | 0.0973 | 0.0635 |
| 0.6931 | 0.112 | 0.1136 | 0.1128 | 0.111 | 0.107 | 0.0932 | 0.0694 | 0.0213 |
| 0.722 | 0.0943 | 0.098 | 0.0994 | 0.0937 | 0.0832 | 0.0672 | 0.0299 | 0.0061 |

Table 7.27: The Jaccard scores (7.12) for the clusterings of the *right* hemisphere in Table 7.17, generated by HP-CONCORD using different $(\lambda_1, \lambda_2)$ penalty parameters, followed by the *persistent homology* method *Louvain* method at the largest tuning parameter value $k$ considered by Louvain (corresponding to *more* clusters), as described in Section 7.4.

## 7.6 Conclusions

This chapter presents HP-CONCORD, a highly scalable inverse covariance matrix estimation method. Based on regularized pseudolikelihood approach, our parallel algorithm leverages distributed memory systems to recover the underlying graph structure at unprecedented dimensionalities. HP-CONCORD exhibits good parallel scaling to thousand of nodes, taking advantage of the aggregate memory across the system to analyze previously intractable data sets. Implementation of our approach is freely available as source code at `https://bitbucket.org/penpornk/spdm3-hpconcord` and as a compiled global user module on Edison system at National Energy Research Scientific Computing Center (`http://www.nersc.gov/`).

HP-CONCORD bridges a computational scalability gap between statistically sound CONCORD methods and practical usability for some of the largest modern day datasets. HP-CONCORD is a statistically grounded and extremely scalable unsupervised learning method that is able to sift through close to a trillion pairwise relationships to find the most prominent ones.

We presented multiple version of HP-CONCORD that are optimized for different sparsity and dimensionality settings, using communication-avoiding linear algebra routines to give both versions good parallel efficiency. We analyzed the performance of HP-CONCORD on synthetic data sets, comparing it to the dominant method for large problems, BigQUIC. While the two methods have similar performance on a single compute node with a chain graph of dependences, HP-CONCORD outperforms BigQUIC for a randomized graph on a single node for the problem sizes we tested. By using multiple compute nodes, HP-CONCORD decreases the running time by orders of magnitude.

Unlike the BigQUIC method, HP-CONCORD does not make any assumptions about the nature of underlying dependency structure. This makes HP-CONCORD more general, but not as fast as they could be for those special cases. For example, BigQUIC benefits from preselecting the active set and graph clustering to divide up computation into separate chunks. Scalability and speed-up of BigQUIC benefits from approximate block structure and any deviation from the assumption is corrected by the method. These improvements would also be beneficial for HP-CONCORD and a good future research direction.

In addition to the synthetic data, we presented a case study using HP-CONCORD to recover the underlying connectivity from a resting state fMRI dataset leveraging many processors. From the HP-CONCORD output, a clustering step takes the graphical model structure and generates a data-driven functional connectivity parcellation. This analysis shows good agreement with previous analysis done with domain knowledge, and will lead to additional work in collaboration with domain experts on the implications to our understanding of functional connectivity of the brain.

# Chapter 8

# Conclusions

This thesis explored communication avoidance in algorithms with sparse all-to-all interactions, some of which are the first applications of the general communication lower bound theorem [47, 107, 48] outside of linear algebra that include both lower bounds and experimental analysis. We proved that replication techniques reduce communication costs asymptotically, and those reductions scale with the replication factor. We showed that replication improves the total running time, even with sparsity in the interaction patterns, although the best measured performance does not always come from maximum replication. Efficient and scalable implementations of these all-to-all problems require new parallel algorithms, restructuring and reformulating the computation to effectively avoid communication. Merely optimizing naïve algorithms post-implementation is not sufficient. For example, communication overlapping gives at best 2× speedup and cannot help much when the communication time far exceeds the computation time, which is often the case when we scale to a large number of cores. Therefore, the communication complexity should be a first class consideration when designing algorithms, just as computation complexity is. It is also important to consider the full algorithmic context and not just optimizing application kernels separately.

Chapters 3 and 4 derived communication lower bounds and communication-optimal algorithms for the 2-way and $k$-way interaction N-body problems, without and with cutoff. These results also showed that structural sparsities such as cutoff and symmetry sparsity could be rearranged to use dense computation to achieve communication optimality. Chapters 5 and 6 gave communication lower bounds for the sparse-dense matrix multiplication problem and new communication-efficient 1.5D matrix multiplication algorithms based on different matrix layouts. Chapter 7 applied the 1.5D matrix multiplication algorithms to implement a massively parallel sparse inverse covariance matrix estimator. Our implementation of the 1.5D multiple-replication-factor matrix multiplications and HP-CONCORD are open source and available online at `https://bitbucket.org/penpornk/spdm3-hpconcord`. Table 8.1 summarizes our lower bound and empirical speedup contributions. Even though we focused on distributed-memory supercomputers, our algorithms can be adapted to cloud computer settings as well because our performance model still applies and we only need to know the available memory size.

| Computation | Lower Bounds | Communication Optimality | Largest speedup reported |
|---|---|---|---|
| All-pairs 2-body [65] | Tight | Optimal | 11.8× |
| 2-body with cutoff [65] | Tight | Optimal | 2× |
| All-unique-triplets 3-body [109] | Tight | Optimal | 42× |
| All-unique-tuples $k$-body with cutoff [109] | Tight | Optimal | N/A |
| Sparse-dense matrix-matrix multiplication [111] | Not tight | N/A | 100× |
| Sparse inverse covariance matrix estimation [110] | N/A | N/A | 10.18× |

Table 8.1: Communication lower bounds, optimality, and empirical speedups of the *communication-avoiding* algorithms presented in this thesis. All algorithms listed as communication-optimal are optimal for all memory sizes. Speedups reported are compared to non-replicating algorithms.

A common parallelization pitfall is to simply partition all data to $p$ equal parts since processors will often have unused memory to spare, i.e., given a problem size $n$ and $p$ processors, the total memory per processor is often more than $n/p$. The key to communication optimality for these all-to-all algorithms is to utilize all available memory. The challenge is to figure out how much of the memory should be allocated to what parts of the inputs and outputs, which is application-specific. For the N-body problem, there is only one input array and the output size is asymptotically the same as input; we showed that the optimal solution is to store $O(M)$ particles where $M$ is the size of the memory available to a processor. For the sparse-dense matrix-matrix multiplication problem $C = A \times B$, there are two inputs ($A$ and $B$) and one output ($C$), all of which can have asymptotically different sizes. We have not found a communication-optimal algorithm nor tight communication lower bounds for the problem, but we know from the lower bounds that the algorithm would store $c_A \operatorname{nnz}(A)/p$ words of $A$, $c_B \operatorname{nnz}(B)/p$ words of $B$, and $c_C \operatorname{nnz}(C)/p$ words of $C$, where $c_A \operatorname{nnz}(A)/p + c_B \operatorname{nnz}(B)/p + c_C \operatorname{nnz}(C)/p = O(M)$ for some constants $c_A$, $c_B$, and $c_C$. The situation is more complicated for the sparse inverse covariance matrix estimation problem because there are more variables. There is ongoing research to find communication lower bounds and optimal loop tiling factors (leading to the amount and exact location of each input and output to be stored) for loop nests with array subscripts that are affine functions of loop indices [47, 107, 48].

Algorithms with multiple dependent steps, such as time-stepping algorithms and iterative algorithms, benefit significantly from communication avoidance since they often require aggressive scaling. In the N-body problem, each timestep depends on the previous timestep, and each simulation usually needs thousands of timesteps to discover useful information. To get the whole simulation done in a reasonable amount of time, each timestep needs to be very fast. Another example is the sparse inverse covariance matrix estimation in Chapter 7. The algorithm can take hundreds of iterations to converge, so to get a short time to convergence, each iteration needs to be done quickly. For example, in Figure 7.6a, Obs' time to convergence for $\rho = 80$K features and $n = 100$ observations in the chain graph case was 3.1455 seconds on 256 nodes. There were 36 gradient iterations and a total of 93 line search steps (across all gradient iterations). This translates to 36 distributed matrix

transposes, 36 dense-dense $80K \times 100 \times 80K$ matrix multiplications, and 93 sparse-dense $80K \times 80K \times 100$ matrix multiplications. This implies these 165 distributed operations must take $3.15/165 \times 1000 = 19$ milliseconds on average to run across all 6,144 cores.

## 8.1 Future Work

The work in this thesis leads to several open questions and opportunities for additional research, ranging from improving our performance model based on what we observed in experiments, to general matrix multiplication lower bounds and algorithms, to applying SpDM$^3$and HP-CONCORD to other applications.

**Computation-latency-bandwidth tradeoffs.** Previous work shows tradeoffs between computation, latency, and bandwidth costs for certain applications such as dense triangular solver [168], dense LU factorization [168], Krylov subspace methods [40], and block coordinate descent methods [63], most of which trade computation and bandwidth with latency costs. Chapter 5 shows that there is a latency-bandwidth tradeoff in the matrix multiplication problem as well, when the operand sizes are vastly different. Chapter 7 has numerous tradeoffs to explore including redundant computations. Instead of minimizing latency and bandwidth individually and assuming the computation workload is always the same, we would focus on minimizing the total running time as a whole instead, taking $\gamma, \alpha$, and $\beta$ as inputs.

**Variable flop rates.** As shown in Chapter 5, the local matrix multiplication efficiency varies significantly with the shapes of the input matrices, to the point that sometimes it is the deciding factor between two algorithm variants. To handle this, our performance model should have $\gamma$ be a function instead of a constant. The local matrix multiplication might need some tuning as well to make $\gamma$ predictable. $\alpha$ and $\beta$ are also varying in practice, but $\alpha$ is hard to statically model since it depends on the current network congestion, and we assume that our message size is large enough to attain the best bandwidth cost and that $\beta$ stays roughly constant.

**General matrix multiplication.** To the best of our knowledge, there are no communication lower bounds for general matrix multiplications where operands can have different shapes and sparsities. Ballard et al. [25] gave lower bounds for matrices with uniformly-distributed nonzereos (Erdős-Rényi) but assumed that both operands are square and have the same number of nonzeroes per row. We are interested in deriving the communication lower bounds and communication-optimal algorithms for the general multiplication of Erdős-Rényi matrices. Adapting the recursive partitioning multiplication from Demmel et al. [60] to also account for sparsities when splitting dimensions is worth exploring.

**Tuning HP-CONCORD.** HP-CONCORD's performance could still be nontrivially improved: We proposed the symmetric 1.5D multiplication algorithm in Section 6.3 but have not taken advantage of the symmetry in the HP-CONCORD implementation. We plan to use this to compute $S = X^T X$ in Cov to save half the floating-point operations. We could merge multiple local operations to minimize the number of passes through the data (or apply blocking). Overlapping communication and computation would also help with the rest of the communication we cannot avoid. We could also use the roofline model [177] on each kernel for fine-tuning.

**Other applications.** There are many applications with distributed sparse-dense matrix-matrix multiplication (SpDM$^3$) as one of their bottlenecks, many of them are iterative algorithms. They would benefit from the 1.5D matrix multiplication algorithms. HP-CONCORD can also be used to analyze many more datasets other than the human brain fMRI data.

# Bibliography

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. "The data locality of work stealing". In: *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2000, pp. 1–12.

[2] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems". In: *Parallel Computing* 59.Supplement C (2016). Theory and Practice of Irregular Applications, pp. 71–96. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2016.10.001. URL: http://www.sciencedirect.com/science/article/pii/S0167819116301041.

[3] R. C. Agarwal et al. "A Three-dimensional Approach to Parallel Matrix Multiplication". In: *IBM J. Res. Dev.* 39.5 (Sept. 1995), pp. 575–582. ISSN: 0018-8646.

[4] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. "Communication complexity of PRAMs". In: *Theoretical Computer Science* 71.1 (1990), pp. 3–28. ISSN: 0304-3975.

[5] Hasan Metin Aktulga et al. "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations". In: *IPDPS*. IEEE. 2014, pp. 1213–1222.

[6] Alnur Ali et al. "Generalized Pseudolikelihood Methods for Inverse Covariance Estimation". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, 20–22 Apr 2017, pp. 280–288. URL: http://proceedings.mlr.press/v54/ali17a.html.

[7] F.E. Allen and J. Cocke. *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971. URL: https://books.google.com/books?id=oeXaZwEACAAJ.

[8] Roland Arnold et al. "SIMAP – the database of all-against-all protein sequence similarities and annotations with new interfaces and increased coverage". In: *Nucleic acids research* 42.D1 (2013), pp. D279–D284.

[9] Salim Arslan et al. "Human brain mapping: A systematic comparison of parcellation methods for the human cerebral cortex". In: *NeuroImage* (Apr. 2017). DOI: 10.1016/j.neuroimage.2017.04.014.

[10] Krste Asanovic et al. *The landscape of parallel computing research: A view from berkeley.* Tech. rep. 2006.

[11] Steve Ashby et al. "The opportunities and challenges of exascale computing". In: *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee* (2010), pp. 1–77.

[12] Atlassian. *Bitbucket: the Git solution for professional teams.* URL: `https://bitbucket.org/`.

[13] BM Axilrod and Ei Teller. "Interaction of the van der Waals type between three atoms". In: *Journal of Chemical Physics* 11 (1943), pp. 299–300.

[14] Ariful Azad et al. "Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication". In: *arXiv preprint arXiv:1510.00844* (Oct. 2015).

[15] Ariful Azad et al. "Identifying Rare Cell Populations in Comparative Flow Cytometry." In: *WABI*. Springer. 2010, pp. 162–175.

[16] Kunihiro Baba, Ritei Shibata, and Masaaki Sibuya. "Partial Correlation and Conditional Correlation as Measures of Conditional Independence". In: *Australian & New Zealand Journal of Statistics* 46.4 (Dec. 2004), pp. 657–664. DOI: `10.1111/j.1467-842x.2004.00360.x`.

[17] J Bachan et al. "UPC++ Specification v1. 0, Draft 4". In: (2017).

[18] Michael Bader and Alexander Heinecke. "Cache oblivious dense and sparse matrix multiplication based on Peano curves". In: *Proceedings of the PARA*. Vol. 8. 2008.

[19] Grey Ballard, James Demmel, and Nicholas Knight. "Communication avoiding successive band reduction". In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 35–44.

[20] Grey Ballard et al. "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23 (2014), pp. 1–155.

[21] Grey Ballard et al. "Communication-optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. Pittsburgh, Pennsylvania, USA: ACM, 2012, pp. 193–204. ISBN: 978-1-4503-1213-4. DOI: `10.1145/2312005.2312044`. URL: `http://doi.acm.org/10.1145/2312005.2312044`.

[22] Grey Ballard et al. "Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication". In: *ACM Trans. Parallel Comput.* 3.3 (Dec. 2016), 18:1–18:34. ISSN: 2329-4949. DOI: `10.1145/3015144`. URL: `http://doi.acm.org/10.1145/3015144`.

[23] Grey Ballard et al. "Minimizing communication in numerical linear algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901.

[24] Grey Ballard et al. "Reconstructing Householder vectors from tall-skinny QR". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 1159–1170.

[25] G. Ballard et al. “Communication optimal parallel multiplication of sparse random matrices”. In: *SPAA*. ACM, 2013, pp. 222–231.

[26] Onureena Banerjee, Laurent El Ghaoui, and Alexandre d’Aspremont. “Model Selection Through Sparse Maximum Likelihood Estimation for Multivariate Gaussian or Binary Data”. In: *J. Mach. Learn. Res.* 9 (June 2008). ISSN: 1532-4435.

[27] Prithviraj Banerjee et al. “The PARADIGM compiler for distributed-memory multicomputers”. In: *Computer* 28.10 (1995), pp. 37–47.

[28] Josh Barnes and Piet Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: (1986).

[29] Jarle Berntsen. “Communication efficient matrix multiplication on hypercubes”. In: *Parallel Computing* 12.3 (1989), pp. 335–342. ISSN: 0167-8191.

[30] Abhinav Bhatele et al. *NAMD: A portable and highly scalable program for biomolecular simulations*. Tech. rep. 2009.

[31] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[32] Robert D Blumofe et al. “Cilk: An efficient multithreaded runtime system”. In: *Journal of parallel and distributed computing* 37.1 (1996), pp. 55–69.

[33] Kevin J. Bowers, Ron O. Dror, and David E. Shaw. “The midpoint method for parallelization of particle simulations”. In: *The Journal of Chemical Physics* 124.18, 184109 (2006), p. 184109. DOI: 10.1063/1.2191489.

[34] Kevin J. Bowers et al. “Scalable algorithms for molecular dynamics simulations on commodity clusters”. In: (2006).

[35] Aydin Buluc and Kamesh Madduri. “Graph Partitioning for Scalable Distributed Graph Computations”. In: *Graph Partitioning and Graph Clustering* 588 (2013), p. 83.

[36] David Callahan, Steve Carr, and Ken Kennedy. “Improving register allocation for subscripted variables”. In: *ACM Sigplan Notices* 25.6 (1990), pp. 53–65.

[37] David Callahan and Ken Kennedy. “Compiling programs for distributed-memory multiprocessors”. In: *The Journal of Supercomputing* 2.2 (1988), pp. 151–169.

[38] Lynn E Cannon. *A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM.* Tech. rep. MONTANA STATE UNIV BOZEMAN ENGINEERING RESEARCH LABS, 1969.

[39] William W Carlson et al. *Introduction to UPC and language specification*. Tech. rep. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[40] Erin Claire Carson. *Communication-avoiding Krylov subspace methods in theory and practice*. University of California, Berkeley, 2015.

[41] Charles W Carter Jr et al. "Four-body potentials reveal protein-specific correlations to stability changes caused by hydrophobic core mutations". In: *Journal of Molecular Biology* 311.4 (2001), pp. 625–638.

[42] Umit V Catalyurek and Cevdet Aykanat. "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication". In: *IEEE Transactions on parallel and distributed systems* 10.7 (1999), pp. 673–693.

[43] Bradford L Chamberlain, David Callahan, and Hans P Zima. "Parallel programmability and the chapel language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[44] Bradford L. Chamberlain, E. Christopher Lewis, and Lawrence Snyder. "Problem Space Promotion and Its Evaluation As a Technique for Efficient Parallel Computation". In: *Proceedings of the 13th International Conference on Supercomputing*. ICS '99. Rhodes, Greece: ACM, 1999, pp. 311–318. ISBN: 1-58113-164-X. DOI: 10.1145/305138.305208. URL: http://doi.acm.org/10.1145/305138.305208.

[45] Ernie Chan et al. "Collective communication: theory, practice, and experience". In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783.

[46] Philippe Charles et al. "X10: an object-oriented approach to non-uniform cluster computing". In: *Acm Sigplan Notices*. Vol. 40. 10. ACM. 2005, pp. 519–538.

[47] Michael Christ et al. *Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays - Part 1*. Tech. rep. UCB/EECS-2013-61. EECS Department, University of California, Berkeley, May 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-61.html.

[48] M. Christ et al. "On Holder-Brascamp-Lieb inequalities for torsion-free discrete Abelian groups". In: *ArXiv e-prints* (Oct. 2015). arXiv: 1510.04190 [math.CA].

[49] Stephanie Coleman and Kathryn S McKinley. "Tile size selection using cache organization and data layout". In: *ACM SIGPLAN Notices*. Vol. 30. 6. ACM. 1995, pp. 279–290.

[50] CF Cornwell and LT Wille. "Parallel molecular dynamics simulations for short-ranged many-body potentials". In: *Computer physics communications* 128.1 (2000), pp. 477–491.

[51] M. Couprie and G. Bertrand. "Topological gray-scale watershed transform". In: *Proc. of SPIE Vision Geometry V*. Vol. 3168. 1997, 136–146.

[52] Ivor Cribben, Tor Wager, and Martin Lindquist. "Detecting functional connectivity change points for single-subject fMRI data". In: *Frontiers in Computational Neuroscience* 7.143 (2013).

[53] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[54] Onkar Dalal and Bala Rajaratnam. *G-AMA: Sparse Gaussian graphical model estimation via alternating minimization*. Tech. rep. Stanford University, May 13, 2014. arXiv: `1405.3034v2 [stat.CO]`.

[55] Kaushik Datta et al. "Optimization and performance modeling of stencil computations on modern microprocessors". In: *SIAM review* 51.1 (2009), pp. 129–159.

[56] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25.

[57] Eliezer Dekel, David Nassimi, and Sartaj Sahni. "Parallel Matrix and Graph Algorithms". In: *SIAM Journal on Computing* 10.4 (1981), pp. 657–675.

[58] J. Demmel and H. D. Nguyen. "Parallel Reproducible Summation". In: *IEEE Transactions on Computers* 64.7 (July 2015), pp. 2060–2070. ISSN: 0018-9340. DOI: `10.1109/TC.2014.2345391`.

[59] James Demmel et al. "Avoiding communication in sparse matrix computations". In: *IPDPS*. IEEE. 2008.

[60] James Demmel et al. "Communication-optimal parallel recursive rectangular matrix multiplication". In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 261–272.

[61] James Demmel et al. "Perfect Strong Scaling Using No Additional Energy". In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 649–660. ISBN: 978-0-7695-4971-2. DOI: `10.1109/IPDPS.2013.32`. URL: `http://dx.doi.org/10.1109/IPDPS.2013.32`.

[62] A. P. Dempster. "Covariance Selection". English. In: *Biometrics* 28.1 (1972),

[63] Aditya Devarakonda et al. "Avoiding communication in primal and dual block coordinate descent methods". In: *CoRR* abs/1612.04003 (2016). arXiv: `1612.04003`. URL: `http://arxiv.org/abs/1612.04003`.

[64] Jack Dongarra et al. *Applied mathematics research for exascale computing*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[65] Michael Driscoll et al. "A Communication-Optimal N-Body Algorithm for Direct Interactions". In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1075–1084. ISBN: 978-0-7695-4971-2. DOI: `10.1109/IPDPS.2013.108`. URL: `http://dx.doi.org/10.1109/IPDPS.2013.108`.

[66] Herbert Edelsbrunner and Dmitriy Morozov. "Persistent Homology: Theory and Applications". In: *Proceedings of the European Congress of Mathematics*. 2012.

[67] Simon B. Eickhoff et al. "Connectivity-based parcellation: Critique and implications". In: *Human Brain Mapping* 36.12 (Sept. 2015), pp. 4771–4792. DOI: `10.1002/hbm.22933`.

[68] Nicole Eikmeier and David F Gleich. "Revisiting Power-law Distributions in Spectra of Real World Networks". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2017, pp. 817–826.

[69] Matthew J Elrod and Richard J Saykally. "Many-body effects in intermolecular forces". In: *Chemical reviews* 94.7 (1994), pp. 1975–1997.

[70] Paul Erdős and Alfréd Rényi. "On Random Graphs". In: *Publicationes Mathematicae* 6.1 (1959), pp. 290–297.

[71] A. Faraj et al. "MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations". In: *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*. 2009.

[72] Robert W Floyd. "An algorithm for coding efficient arithmetic operations". In: *Communications of the ACM* 4.1 (1961), pp. 42–51.

[73] Geoffrey C. Fox and Steve W. Otto. "Algorithms for concurrent processors". In: *Phys. Today* 37N5 (1984), pp. 50–59. DOI: 10.1063/1.2916241.

[74] J. Friedman, T. Hastie, and R. Tibshirani. "Sparse inverse covariance estimation with the graphical lasso". In: *Biostatistics* 9.3 (Dec. 2007), pp. 432–441. DOI: 10.1093/biostatistics/kxm045.

[75] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Sparse inverse covariance estimation with the graphical lasso". In: *Biostatistics* 9.3 (2008), pp. 432–441.

[76] Katsuki Fujisawa, Masakazu Kojima, and Kazuhide Nakata. "Exploiting sparsity in primal-dual interior-point methods for semidefinite programming". In: *Mathematical Programming* 79.1-3 (1997), pp. 235–253.

[77] Samuel H Fuller, Lynette I Millett, et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011.

[78] Srimanth Gadde. *Graph partitioning algorithms for minimizing inter-node communication on a distributed system*. The University of Toledo, 2013.

[79] Hin Hark Gan, Alexander Tropsha, and Tamar Schlick. "Lattice protein folding with two and four-body statistical potentials". In: *Proteins: Structure, Function, and Bioinformatics* 43.2 (2001), pp. 161–174.

[80] R. van de Geijn and J. Watts. "SUMMA: Scalable Universal Matrix Multiplication Algorithm". In: *Concurr. Comput. : Pract. Exper.* 9.4 (1997), pp. 255–274.

[81] Evangelos Georganas et al. "Communication avoiding and overlapping for numerical linear algebra". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 100.

[82] Matthew F. Glasser et al. "A multimodal parcellation of human cerebral cortex". In: *Nature* 536.7615 (Aug. 2016), pp. 171–178.

[83] Gustavo Glusman et al. "Ultrafast Comparison of Personal Genomes via Precomputed Genome Fingerprints". In: *Frontiers in Genetics* 8 (2017), p. 136. ISSN: 1664-8021. DOI: 10.3389/fgene.2017.00136. URL: https://www.frontiersin.org/article/10.3389/fgene.2017.00136.

[84] Neil Zhenqiang Gong and Bin Liu. "You are Who You Know and How You Behave: Attribute Inference Attacks via Users' Social Friends and Behaviors". In: *CoRR* abs/1606.05893 (2016). arXiv: 1606.05893. URL: http://arxiv.org/abs/1606.05893.

[85] G Gournas et al. "Compiling tiled iteration spaces for clusters". In: *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE. 2002, pp. 360–369.

[86] *Graph500 benchmark*. www.graph500.org.

[87] Leslie Greengard and Vladimir Rokhlin. "A fast algorithm for particle simulations". In: *Journal of computational physics* 73.2 (1987), pp. 325–348.

[88] Gero Greiner and Riko Jacob. "The I/O complexity of sparse matrix dense matrix multiplication". In: *LATIN 2010: Theoretical Informatics*. Springer, 2010, pp. 143–156.

[89] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828.

[90] Thomas KR Gross and Zoltán Majó. "A library for portable and composable data locality optimizations for NUMA systems". In: *PPoPP 2015 Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2015, pp. 227–238.

[91] Tsuyoshi Hamada et al. "42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence". In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE. 2009, pp. 1–12.

[92] Stefan Harfst et al. "Performance analysis of direct N-body algorithms on special-purpose supercomputers". In: *New Astronomy* 12.5 (2007), pp. 357–377. ISSN: 1384-1076. DOI: http://dx.doi.org/10.1016/j.newast.2006.11.003. URL: http://www.sciencedirect.com/science/article/pii/S138410760600131X.

[93] Cho-Jui Hsieh et al. "BIG & QUIC: Sparse Inverse Covariance Estimation for a Million Variables". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 3165–3173. URL: https://papers.nips.cc/paper/4923-big-quic-sparse-inverse-covariance-estimation-for-a-million-variables.

[94] Lei Huang et al. "Enabling locality-aware computations in OpenMP". In: *Scientific Programming* 18.3-4 (2010), pp. 169–181.

[95] *Intrepid Guide — Argonne Leadership Computing Facility.* URL: `https://www.alcf.anl.gov/resource-guides/intrepid-and-surveyor-guide`.

[96] Dror Irony, Sivan Toledo, and Alexander Tiskin. "Communication lower bounds for distributed-memory matrix multiplication". In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026.

[97] Jinyuan Jia et al. "AttriInfer: Inferring User Attributes in Online Social Networks Using Markov Random Fields". In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 1561–1569.

[98] Hong Jia-Wei and H. T. Kung. "I/O complexity: The red-blue pebble game". In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing.* STOC '81. Milwaukee, Wisconsin, United States: ACM, 1981, pp. 326–333.

[99] S. Lennart Johnsson. "Minimizing the communication time for matrix multiplication on multiprocessors". In: *Parallel Comput.* 19 (11 Nov. 1993), pp. 1235–1257. ISSN: 0167-8191.

[100] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: a portable concurrent object oriented system based on C++". In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. OOPSLA '93. Washington, D.C., United States: ACM, 1993, pp. 91–108. ISBN: 0-89791-587-9.

[101] Laxmikant V Kale and Sanjeev Krishnan. "Charm++: Parallel programming with message-driven objects". In: *Parallel programming using C+* (1996), pp. 175–213.

[102] Prabhanjan Kambadur and Aurelie Lozano. "A Parallel, Block Greedy Method for Sparse Inverse Covariance Estimation for Ultra-high Dimensions". In: *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Carlos M. Carvalho and Pradeep Ravikumar. Vol. 31. Proceedings of Machine Learning Research. Scottsdale, Arizona, USA: PMLR, 29 Apr–01 May 2013, pp. 351–359. URL: `http://proceedings.mlr.press/v31/kambadur13a.html`.

[103] Kshitij Khare, Sang-Yun Oh, and Bala Rajaratnam. "A convex pseudolikelihood framework for high dimensional partial correlation estimation with convergence guarantees". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 77.4 (2015), pp. 803–825. ISSN: 1467-9868.

[104] Hyunsoo Kim and Haesun Park. "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis". In: *Bioinformatics* 23.12 (2007), pp. 1495–1502.

[105] Jingu Kim and Haesun Park. "Fast nonnegative matrix factorization: An active-set-like method and comparisons". In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3261–3281.

[106] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[107] Nicholas Knight. "Communication-Optimal Loop Nests". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-185.html.

[108] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. "Combined selection of tile sizes and unroll factors using iterative compilation". In: *The Journal of Supercomputing* 24.1 (2003), pp. 43–67.

[109] Penporn Koanantakool and Katherine Yelick. "A Computation- and Communication-optimal Parallel Direct 3-body Algorithm". In: *ACM/IEEE SC'14*. New Orleans, Louisana: IEEE Press, 2014, pp. 363–374. ISBN: 978-1-4799-5500-8.

[110] Penporn Koanantakool et al. "Communication-Avoiding Optimization Methods for Massive-Scale Graphical Model Structure Learning". In: *ArXiv e-prints* (Oct. 2017). arXiv: 1710.10769 [stat.ML].

[111] Penporn Koanantakool et al. "Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 842–853. DOI: 10.1109/IPDPS.2016.117.

[112] D. J. Kuck, Y. Muraoka, and Shyh-Ching Chen. "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup". In: *IEEE Transactions on Computers* C-21.12 (Dec. 1972), pp. 1293–1310. ISSN: 0018-9340. DOI: 10.1109/T-C.1972.223501.

[113] Manaschai Kunaseth et al. "A scalable parallel algorithm for dynamic range-limited $n$-tuple computation in many-body molecular dynamics simulation". In: *SC*. 2013, p. 71.

[114] Monica D Lam, Edward E Rothberg, and Michael E Wolf. "The cache performance and optimizations of blocked algorithms". In: *ACM SIGARCH Computer Architecture News*. Vol. 19. 2. ACM. 1991, pp. 63–74.

[115] Leslie Lamport. "The Parallel Execution of DO Loops". In: *Commun. ACM* 17.2 (Feb. 1974), pp. 83–93. ISSN: 0001-0782. DOI: 10.1145/360827.360844. URL: http://doi.acm.org/10.1145/360827.360844.

[116] Yemin Lan et al. "POGO-DB – a database of pairwise-comparisons of genomes and conserved orthologous genes". In: *Nucleic acids research* 42.D1 (2013), pp. D625–D632.

[117] A. J. Lawrance. "On Conditional and Partial Correlation". In: *The American Statistician* 30.3 (Aug. 1976), p. 146. DOI: 10.2307/2683864.

[118] Dongryeol Lee, Arkadas Ozakin, and Alexander G Gray. "Multibody multipole methods". In: *Journal of Computational Physics* 231.20 (2012), pp. 6827–6845.

[119] Jason D. Lee and Trevor J. Hastie. "Learning the Structure of Mixed Graphical Models". In: *Journal of Computational and Graphical Statistics* 24.1 (Jan. 2015), pp. 230–253. DOI: 10.1080/10618600.2014.900500.

[120] Haiqiong Li. “All-pair comparison of billion-base genome sequences”. PhD thesis. Rensselaer Polytechnic Institute, 2013.

[121] Jianhui Li, Zhongwu Zhou, and Richard. J. Sadus. “A Cyclic Force Decomposition Algorithm for Parallelising Three-Body Interactions in Molecular Dynamics Simulations”. In: *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 1 (IMSCCS’06) - Volume 01*. IMSCCS ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 338–343. DOI: `10.1109/IMSCCS.2006.3`. URL: `http://dx.doi.org/10.1109/IMSCCS.2006.3`.

[122] Jianhui Li, Zhongwu Zhou, and Richard J. Sadus. “Modified force decomposition algorithms for calculating three-body interactions via molecular dynamics”. In: *Computer Physics Communications* 175.11-12 (2006), pp. 683–691.

[123] Jianhui Li, Zhongwu Zhou, and Richard J. Sadus. “Parallel algorithms for molecular dynamics with induction forces”. In: *Computer Physics Communications* 178.5 (2008), pp. 384–392.

[124] Wei Li and Keshav Pingali. *Access normalization: loop restructuring for numa compilers*. Vol. 27. 9. ACM, 1992.

[125] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. “Optimizing data locality for fork/join programs using constrained work stealing”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2014, pp. 857–868.

[126] Chinghway Lim and Bin Yu. “Estimation Stability With Cross-Validation (ESCV)”. In: *Journal of Computational and Graphical Statistics* 25.2 (Apr. 2016), pp. 464–492. DOI: `10.1080/10618600.2015.1020159`.

[127] L. H. Loomis and H. Whitney. “An inequality related to the isoperimetric inequality”. In: *Bull. Amer. Math. Soc.* 55.10 (Oct. 1949), pp. 961–962.

[128] Kamesh Madduri et al. “Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms”. In: *Parallel Computing* 37.9 (2011), pp. 501–520.

[129] Kamesh Madduri et al. “Optimization of Parallel Particle-to-Grid Interpolation on Leading Multicore Platforms”. In: 23.10 (2012).

[130] Gianluca Marcelli. “The role of three-body interactions on the equilibrium and non-equilibrium properties of fluids from molecular simulation”. PhD thesis. Swinburne University of Technology, 2001.

[131] Gianluca Marcelli and Richard J Sadus. “Molecular simulation of the phase behavior of noble gases using accurate two-body and three-body intermolecular potentials”. In: *The Journal of chemical physics* 111.4 (1999), pp. 1533–1540.

[132] Guillaume Marrelec et al. “Partial correlation for functional brain interactivity investigation in functional MRI”. In: *NeuroImage* 32.1 (Aug. 2006), pp. 228–237. DOI: `10.1016/j.neuroimage.2005.12.057`.

[133] O Matsuoka, E Clementi, and M Yoshimine. "CI study of the water dimer potential surface". In: *The Journal of Chemical Physics* 64.4 (2008), pp. 1351–1361.

[134] Rahul Mazumder and Trevor Hastie. "Exact covariance thresholding into connected components for large-scale graphical lasso". In: *Journal of Machine Learning Research* 13.Mar (2012), pp. 781–794.

[135] William F McColl and Alexandre Tiskin. "Memory-efficient matrix multiplication in the BSP model". In: *Algorithmica* 24.3 (1999), pp. 287–297.

[136] Michael McCourt, Barry Smith, and Hong Zhang. "Sparse Matrix-Matrix Products Executed Through Coloring". In: *SIAM Journal on Matrix Analysis and Applications* 36.1 (2015), pp. 90–109.

[137] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. "Improving data locality with loop transformations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.4 (1996), pp. 424–453.

[138] Nicolai Meinshausen and Peter Bhlmann. "Stability selection". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72.4 (July 2010), pp. 417–473. DOI: 10.1111/j.1467-9868.2010.00740.x.

[139] Nicolai Meinshausen and Peter Bühlmann. "High-dimensional graphs and variable selection with the Lasso". In: *The Annals of Statistics* 34.3 (June 2006), pp. 1436–1462. DOI: 10.1214/009053606000000281.

[140] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and many-core processors". In: *Scientific Programming* 2015 (2015), p. 5.

[141] Aiichiro Nakano, Priya Vashishta, and Rajiv K Kalia. "Parallel multiple-time-step molecular dynamics with three-body interaction". In: *Computer physics communications* 77.3 (1993), pp. 303–312.

[142] Sang Oh et al. "Optimization Methods for Sparse Pseudo-Likelihood Graphical Model Selection". In: *NIPS 27*. 2014, pp. 667–675.

[143] Gloria Ortega et al. "Fastspmm: An efficient library for sparse matrix matrix product on GPUs". In: *The Computer Journal* 57.7 (2014), pp. 968–979.

[144] Liam D. O'Suilleabhain. "Three Body Approximation to the Condensed Phase of Water". MA thesis. Berkeley,CA: University of California, Berkeley, 2013.

[145] Richard E. Passingham, Klaas E. Stephan, and Rolf Ktter. "The anatomical basis of functional localization in the cortex". In: *Nature Reviews Neuroscience* 3.8 (Aug. 2002), pp. 606–616. DOI: 10.1038/nrn893.

[146] Guilherme C Pena et al. "An efficient GPU multiple-observer siting method based on sparse-matrix multiplication". In: *ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. ACM. 2014, pp. 54–63.

[147] Jie Peng et al. “Partial Correlation Estimation by Joint Sparse Regression Models”. In: *Journal of the American Statistical Association* 104.486 (2009), pp. 735–746.

[148] James C. Phillips et al. “Scalable molecular dynamics with NAMD”. In: *Journal of Computational Chemistry* 26.16 (2005), pp. 1781–1802. ISSN: 1096-987X. DOI: 10.1002/jcc.20289.

[149] Andrea Pietracaprina et al. “Space-round tradeoffs for MapReduce computations”. In: *ICS*. ACM. 2012, pp. 235–244.

[150] Steve Plimpton. “Fast parallel algorithms for short-range molecular dynamics”. In: *J. Comput. Phys.* 117.1 (Mar. 1995), pp. 1–19. ISSN: 0021-9991. DOI: 10.1006/jcph.1995.1039. URL: http://dx.doi.org/10.1006/jcph.1995.1039.

[151] Steven J Plimpton and Aidan P Thompson. “Computational aspects of many-body potentials”. In: *MRS bulletin* 37.05 (2012), pp. 513–521.

[152] Allan Kennedy Porterfield. “Software methods for improvement of cache performance on supercomputer applications”. PhD thesis. Rice University, 1989.

[153] Giuseppe Profiti, Piero Fariselli, and Rita Casadio. “AlignBucket: a tool to speed up all-against-all protein sequence alignments optimizing length constraints”. In: *Bioinformatics* 31.23 (2015), pp. 3841–3843.

[154] J Ramanujam and P Sadayappan. “Tiling multidimensional iteration spaces for multicomputers”. In: *Journal of Parallel and Distributed Computing* 16.2 (1992), pp. 108–120.

[155] J Ramanujam and P Sadayappan. “Tiling of Iteration Spaces for Multicomputers.” In: *ICPP (2)*. 1990, pp. 179–186.

[156] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* ” O’Reilly Media, Inc.”, 2007.

[157] Gabriel Rivera and Chau-Wen Tseng. “A comparison of compiler tiling algorithms”. In: *Compiler Construction*. Springer. 1999, pp. 1–99.

[158] Guilherme V. Rocha, Peng Zhao, and Bin Yu. “A path following algorithm for Sparse Pseudo-Likelihood Inverse Covariance Estimation (SPLICE)”. In: (July 23, 2008). arXiv: 0807.3734v1 [stat.ME].

[159] Anthony Scemama et al. “Quantum Monte Carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond”. In: *Journal of computational chemistry* 34.11 (2013), pp. 938–951.

[160] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. “An in-depth study of stochastic Kronecker graphs”. In: *ICDM*. IEEE. 2011, pp. 587–596.

[161] David E. Shaw. “A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions”. In: *Journal of Computational Chemistry* 26.13 (2005), pp. 1318–1328. ISSN: 1096-987X. DOI: 10.1002/jcc.20267.

[162] David E. Shaw et al. "Anton, a special-purpose machine for molecular dynamics simulation". In: *Proceedings of the 34th annual international symposium on Computer architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 1–12. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250664.

[163] Stephen M Smith et al. "Resting-state fMRI in the human connectome project". In: *NeuroImage* 80 (2013), pp. 144–168.

[164] Marc Snir. "A Note on N-Body Computations with Cutoffs". In: *Theory of Computing Systems* 37 (2 2004), pp. 295–318. ISSN: 1432-4350.

[165] Edgar Solomonik. "Provably Efficient Algorithms for Numerical Tensor Algebra". PhD thesis. EECS Department, University of California, Berkeley, Sept. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-170.html.

[166] Edgar Solomonik, Aydin Buluc, and James Demmel. "Minimizing communication in all-pairs shortest paths". In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 548–559.

[167] Edgar Solomonik and James Demmel. "Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms". In: *Euro-Par*. Vol. 6853. 2011, pp. 90–109. ISBN: 978-3-642-23396-8.

[168] Edgar Solomonik et al. "Tradeoffs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '14. Prague, Czech Republic: ACM, 2014, pp. 307–318. ISBN: 978-1-4503-2821-0. DOI: 10.1145/2612669.2612671. URL: http://doi.acm.org/10.1145/2612669.2612671.

[169] J. V. Sumanth, David R. Swanson, and Hong Jiang. "A Symmetric Transformation for 3-body Potential Molecular Dynamics Using Force-decomposition in a Heterogeneous Distributed Environment". In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS '07. Seattle, Washington: ACM, 2007, pp. 105–115. ISBN: 978-1-59593-768-1. DOI: 10.1145/1274971.1274988. URL: http://doi.acm.org/10.1145/1274971.1274988.

[170] Adrian Tate et al. "Programming abstractions for data locality". In: PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland. 2014.

[171] Alexander Tiskin. "Bulk-synchronous parallel Gaussian elimination". In: *Journal of Mathematical Sciences* 108.6 (2002), pp. 977–991.

[172] Alexander Tiskin. "Communication-efficient parallel generic pairwise elimination". In: *Future Generation Computer Systems* 23.2 (2007), pp. 179–188.

[173] Alexandre Tiskin. "All-Pairs Shortest Paths Computation in the BSP Model". In: *ICALP*. 2001, pp. 178–189.

[174] Huahua Wang et al. "Large Scale Distributed Sparse Precision Estimation". In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 584–592. URL: `http://papers.nips.cc/paper/5037-large-scale-distributed-sparse-precision-estimation`.

[175] Michael S Warren and John K Salmon. "Astrophysical N-body simulations using hierarchical tree data structures". In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press. 1992, pp. 570–576.

[176] Z. Wen and Y. Zhang. "Block algorithms with augmented Rayleigh-Ritz projections for large-scale eigenpair computation". In: *ArXiv e-prints* (July 2015). arXiv: `1507.06078 [math.NA]`.

[177] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: `10.1145/1498765.1498785`. URL: `http://doi.acm.org/10.1145/1498765.1498785`.

[178] Lucas D Wittwer et al. "Speeding up all-against-all protein comparisons while maintaining sensitivity by considering subsequence-level homology". In: *PeerJ* 2 (2014), e607.

[179] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 30–44. ISBN: 0-89791-428-7. DOI: `10.1145/113445.113449`. URL: `http://doi.acm.org/10.1145/113445.113449`.

[180] M. Wolfe. "More Iteration Space Tiling". In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Supercomputing '89. Reno, Nevada, USA: ACM, 1989, pp. 655–664. ISBN: 0-89791-341-8. DOI: `10.1145/76263.76337`. URL: `http://doi.acm.org/10.1145/76263.76337`.

[181] Andrei P Yershov. "ALPHA – an automatic programming system of high efficiency". In: *Journal of the ACM (JACM)* 13.1 (1966), pp. 17–24.

[182] Ming Yuan and Yi Lin. "Model selection and estimation in the Gaussian graphical model". In: *Biometrika* 94.1 (2007), pp. 19–35. DOI: `10.1093/biomet/asm018`.

[183] Junchao Zhang, Babak Behzad, and Marc Snir. "Optimizing the Barnes-Hut algorithm in UPC". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE. 2011, pp. 1–11.