UNIVERSITY OF CALIFORNIA SAN DIEGO

MATE, a Unified Model for Communication-Tolerant Scientific Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Sergio Miguel Martin

Committee in charge:

Professor Scott Baden, Chair
Professor George Porter, Co-Chair
Professor Tajana Rosing
Professor Sutanu Sarkar
Professor John Weare

2018

Copyright
Sergio Miguel Martin, 2018
All rights reserved.

The dissertation of Sergio Miguel Martin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

Co-Chair

____

Chair

University of California San Diego

2018

EPIGRAPH

Science! true daughter of Old Time thou art!

Who alterest all things with thy peering eyes.

—Edgar Allan Poe

TABLE OF CONTENTS

Signature Page iii

Epigraph iv

Table of Contents v

List of Figures ix

List of Tables xiii

Acknowledgements xiv

Vita xvii

Abstract of the Dissertation xviii

Chapter 1 Introduction 1

Chapter 2 Background and Motivation 5
2.1 Overview 5
2.2 Anatomy of a Supercomputer 5
2.2.1 Node Architecture 6
2.2.2 Interconnect Design 7
2.3 Costs of Communication 8
2.3.1 Intra-Node Communication 8
2.3.2 Network Communication 10
2.4 Distributed Programming Paradigms 12
2.4.1 SPMD 12
2.4.2 PGAS 15
2.4.3 Asynchronous PGAS 16
2.4.4 Dataflow 18
2.5 Communication Tolerant-Programming 20
2.5.1 Communication/Computation Overlap 21
2.5.2 Data Motion Reduction 23
2.6 Future Outlook 26

Chapter 3 The MATE Model 29
3.1 Overview 29
3.1.1 Motivating Example 30
3.2 Communication-Reducing Mechanisms 33
3.2.1 Mechanism I: Domain Overdecomposition 34
3.2.2 Mechanism II: Hierarchical Rank Distribution 35

3.2.3 Mechanism III: Code Region Scheduling 38
3.3 Communication Reducing Effects 48
3.4 Related Work 51
3.4.1 MPI+KLT Model 51
3.4.2 MPI+MPI Model 51
3.4.3 MPI+ULT Model 52
3.4.4 Latency-Hiding Models 54
3.5 Summary 57

Chapter 4 Design and Implementation 58
4.1 Overview 58
4.2 Translation Process 59
4.2.1 Step I: MPI to MATE Call Replacement 60
4.2.2 Step II: Parsing Graph Directives 61
4.2.3 Step III: Parsing For Loop Graphs 63
4.2.4 Step IV: Replacing The Main Function 64
4.3 Runtime Support 66
4.3.1 Runtime System Design 66
4.3.2 Execution Model 68
4.4 Communication Backend 75
4.4.1 Point-To-Point Communication 76
4.4.2 Barriers 79
4.4.3 Collective Communication 81

Chapter 5 Test Case I: Jacobi3D 83
5.1 Overview 83
5.1.1 Computation 84
5.1.2 Verification 85
5.2 Strong Scaling Studies 86
5.2.1 Cori Phase I (Haswell) 88
5.2.2 Cori Phase II (KNL) 94
5.3 Summary 96

Chapter 6 Test Case II: Cannon's Algorithm 98
6.1 Overview 98
6.2 Code Variants 99
6.2.1 Base MPI Algorithm 99
6.2.2 Overlapping MPI Algorithm 102
6.2.3 MATE Variant 104
6.3 Size Scaling Study 107
6.4 Weak Scaling Studies 110
6.4.1 Cori Phase I (Haswell) 111
6.4.2 Cori Phase II (KNL) 112

6.5 Summary 114

Chapter 7 Test Case III: Cloverleaf3D 116
7.1 Overview 116
7.2 Code Variants 117
7.2.1 Base MPI Algorithm 117
7.2.2 MATE Variant 120
7.3 Strong Scaling Studies 124
7.3.1 Cori Phase I (Haswell) 124
7.3.2 Cori Phase II (KNL) 126
7.4 Summary 128

Chapter 8 Load Balancing 129
8.1 Overview 129
8.2 Example: Mpix_flowCart 132
8.3 Rebalancing with MATE 133
8.3.1 Mechanism I: Hierarchical Overdecomposition 133
8.3.2 Mechanism II: Inter-Node Balancing 135
8.3.3 Mechanism III: Intra-Node Balancing 138
8.4 Experimental Results 139
8.5 Concurrency Limitations 141
8.6 Related Work 143
8.7 Summary 144

Chapter 9 Conclusions and Future Work 145
9.1 Research Contributions 145
9.2 Limitations and Future Work 146
9.2.1 Improve Thread Concurrency 146
9.2.2 UPC++ Integration 147
9.2.3 CUDA Integration 147
9.2.4 Global Variables Handling 147
9.2.5 Lightweight Translation 148
9.2.6 Support Fortran Annotations 148
9.2.7 Support for Collective Communication Overlap 149
9.2.8 Automate Graph Generation 149

Appendix A Experimental Environment 150
A.1 Hardware Configuration 150
A.2 Software Configuration 151

Appendix B Code Optimizations 153
B.1 Cache Blocking 153
B.2 Vectorization 154
B.3 Cubic Mapping 155

Appendix C MPI Concurrency Limitation 158
C.1 Problem 1: Thread Serialization 158
C.2 Problem 2: Bandwidth Saturation 160
C.3 Experimental Tests 163
C.3.1 Possible Solution 165

Appendix D Load Balancing Algorithms 166
D.1 Consecutive Rebalancer 166
D.2 Shuffling Rebalancer 167

Appendix E MATE Application Programming Interface 169
E.1 MATE Model Interface 169
E.2 Runtime System Interface 172
E.3 Supported MPI Functions 173

Bibliography 174

LIST OF FIGURES

Figure 2.1: Configuration of a 32-core NERSC Cori Phase I (Haswell) Node. 6
Figure 2.2: Example roofline diagram that compares two applications. Application *A*, limited by memory bandwidth, and Application *B*, limited by the peak CPU performance. 9
Figure 2.3: Execution timeline of an SPMD application. 13
Figure 2.4: Two-sided Communication Protocol. 13
Figure 2.5: One-sided Communication Protocol operations: *put* and *get*. 14
Figure 2.6: Partitioning of a shared array in UPC and examples of language-enabled communication. 15
Figure 2.7: Execution timeline of an asynchronous application. 16
Figure 2.8: Example DAG in a Dataflow application. 19
Figure 2.9: Core usage timeline of a process under the bulk-synchronous model. 21
Figure 2.10: Core usage timeline of a split-phase application that employs separate communication dependent and independent computation to achieve communication / computation overlap. 23
Figure 2.11: A typical deployment of an example MPI application with processes mapped to a single core. 24
Figure 2.12: A typical deployment of a hybrid MPI+KLT application where each MPI process spans a group of kernel-level threads, each mapped to a single core. 25
Figure 2.13: Memory/Performance ratio of supercomputers since 2001. Data source: [114] 27

Figure 3.1: Stencil solver on a 2D Grid. 31
Figure 3.2: SPMD Decomposition of a 2D grid into 4 MPI Ranks. 32
Figure 3.3: MPI pseudo-code of a structured 2D grid stencil solver. 33
Figure 3.4: Traditional Decomposition vs Overdecomposed 2D grid. 34
Figure 3.5: Traditional Decomposition vs MATE Hierarchical overdecomposition. . . . 36
Figure 3.6: Simplified pseudo-code of the 13-point stencil solver. 37
Figure 3.7: Simple example of a MATE-annotated program. 39
Figure 3.8: Dependency graph of the MATE-annotated *for* loop in Fig. 3.8. 39
Figure 3.9: Pseudo-code example of (top) an incorrect description, and (bottom) a correct description of a MATE dependency graph. 41
Figure 3.10: Example use of MATE's inter-rank dependencies. 42
Figure 3.11: Inter-rank dependencies for rank (*0,11*). 43
Figure 3.12: Example of a MATE-annotated *for* loop. 44
Figure 3.13: Dependency graph of the MATE-annotated *for* loop in Fig. 3.12. IR = Inter-rank dependency. 44
Figure 3.14: Simple example of a MATE-annotated program. 45
Figure 3.15: Solver section of the code from 3.6, enhanced with a MATE dependency graph 47
Figure 3.16: Dependency graph generated by the code in Fig. 3.15. IR = Inter-rank dependency. 47
Figure 3.17: Hypothetical core usage timelines. 49

Figure 3.18: Example deployment of a hybrid MPI+SHM application where node co-located MPI processes can communicate through shared memory. 52
Figure 3.19: Example deployment of a hybrid MPI+ULT application where each MPI process is assigned a single core, but spans multiple MPI ranks, implemented as ULTs. 53

Figure 4.1: Annotation and compilation flowchart of a MATE application. 58
Figure 4.2: Annotated section of the code in 3.15. 60
Figure 4.3: Step 1 of translation replaces MPI calls with its equivalent MATE call. 61
Figure 4.4: Step 2 of translation creates scheduling structures for a region-level execution. 62
Figure 4.5: Step 3 of translation creates the structures to support *for*-loop based graphs. 64
Figure 4.6: The final step of translation creates a surrogate *main* function and defines the dependency graphs. 65
Figure 4.7: Decomposition model and implementation of a MATE process. 66
Figure 4.8: Lifetime of a MATE worker. 69
Figure 4.9: Lifetime of a MATE Rank. 71
Figure 4.10: Flowchart of the findNextRegion method. 74
Figure 4.11: Structure of an MPI request. 77
Figure 4.12: Structure of a MATE/MPI request, including local rank identifiers in the *tag* field. 78
Figure 4.13: Flowchart of MATE barrier mechanisms. 80
Figure 4.14: Flowchart of MATE collecive communication operations. 81

Figure 5.1: 13-Point Stencil on a three-dimensional grid. 83
Figure 5.2: Pseudo-code of the solver kernel of Jacobi3D. 84
Figure 5.3: Verification code for Jacobi3D. 85
Figure 5.4: Pseudo-code of the manually overlapping variant of Jacobi3D. 87
Figure 5.5: Strong Scaling results for Jacobi3D on 4k to 32k Cori Phase I cores. The number above each bar represents the total speedup compared to *Basic-MPI*. 89
Figure 5.6: Time spent on different phases of our solver on 32k Cori Phase I cores. . . 90
Figure 5.7: Core Timelines. (Top) Basic MPI (8 Ranks), (Bottom) MATE (64 Ranks). . 93
Figure 5.8: Timeline of local rank 0 transitioning across the 8 cores in the MATE process. 93
Figure 5.9: Strong Scaling results for Jacobi3D on 8k to 64k Cori Phase II cores. . . . 94
Figure 5.10: Time spent on different phases our solver on 64k Cori Phase II cores. . . . 94

Figure 6.1: Baseline Cannon2D algorithm where ranks shift the A and B submatrices along rows and columns of the processor geometry, in a ring topology. . . . 100
Figure 6.2: MPI pseudo-code of Cannon2D's solver. 101
Figure 6.3: A rank achieves overlap by receiving A^{i+1} and B^{i+1} submatrices for the next step while updating the value of C with $A^i \times B^i$ in the current step. 103
Figure 6.4: MPI pseudo-code of overlapping Cannon2D's solver. 103

Figure 6.5: Communication in the MATE variant of the Cannon's solver using a hierarchical decomposition. We simplified this figure to show MPI messages only across boundary ranks. However, every rank exchanges MPI messages with neighboring processes. 104
Figure 6.6: Pseudo-code of Cannon's algorithm enhanced with MATE annotations and calls. 105
Figure 6.7: Rank (0,11) declares inter-rank dependencies only along its same column/row. 106
Figure 6.8: Matrix Size Scaling of our three Cannon2D variants on 4k Cori Phase I nodes. 108
Figure 6.9: Execution breakdown for Cannon2D on 4096 Haswell cores with matrix sizes (top) n=12288, (middle) n=24576, and (bottom) n=40960. 109
Figure 6.10: Weak Scaling results for Cannon2D on 4k to 16k Cori Phase I cores. 111
Figure 6.11: Execution breakdown on 16k Cori Phase I cores. 111
Figure 6.12: Weak Scaling results for Cannon2D on 64, 256, and 1024 Cori Phase II nodes. 112
Figure 6.13: Execution breakdown on 64k Cori Phase II cores. 113

Figure 7.1: CloverLeaf3D's staggered grid with cell and node centric variables. 116
Figure 7.2: CloverLeaf's main solver's kernel and exchange operations. 118
Figure 7.3: CloverLeaf3D's *update_halo()* 3-stage boundary exchange procedure. . . . 119
Figure 7.4: MPI pseudo-code of Cloverleaf3D's *update_halo()* routine (simplified). . . 120
Figure 7.5: MPI pseudo-code of Cloverleaf3D's *exchange_faces()* routine (simplified). 120
Figure 7.6: MATE hierarchical variant of Cloverleaf3D's halo exchange, represented in 2D for simplicity. Local ranks exchange boundary field information by direct copy while inter-process communication requires buffer packing/unpacking. 121
Figure 7.7: MATE's pseudo-code of Cloverleaf3D's *exchange_faces()* routine (simplified). 122
Figure 7.8: Graph derived from MATE annotations on the *exchange_faces()* routine. . . 122
Figure 7.9: Strong Scaling results for Cloverleaf3D on 4k to 16k Cori Phase I cores. . . 125
Figure 7.10: Execution breakdown on 16k Cori Phase I cores. 125
Figure 7.11: Strong Scaling results for Cloverleaf3D on 8k to 32k Cori Phase II cores. . 126
Figure 7.12: Execution breakdown at 512 Cori Phase II nodes. 127

Figure 8.1: Execution timeline of an imbalanced application. 130
Figure 8.2: Workload distribution of applications A and A'. 131
Figure 8.3: Computation time distribution of nodes 26, 27 and 28 on a 2048 Haswell core-run. 132
Figure 8.4: Computation time distribution of the 32 cores on node 27, *(top)* without overdecomposition, *(middle)* using an overdecomposition factor of 4, and *(bottom)* using a hierarchical decomposition with 4 workers per MATE process. 134
Figure 8.5: Workload imbalance across three nodes. 135
Figure 8.6: Rank-to-Rank communication distribution in mpix_flowCart. Darker spots indicate a higher communication volume; clearer spots indicate zero or little communication. 136

Figure 8.7: Workload distribution from 8.5 after inter-node rebalancing. Node 26 donated one rank to node 27 which, in turn, donated six ranks to Node 28, moving rank distribution (thick gray lines) to the left. 137
Figure 8.8: Computation time distributions of 16 MATE processes after rank re-shuffling. 138
Figure 8.9: Running time breakdown of 2048 Cori Phase I cores for (*top*) the baseline variant, and (*bottom*) the MATE rebalanced variant. 140
Figure 8.10: Execution breakdown for different multi-threading levels on Cori Phase I. . . 141
Figure 8.11: Message size histogram for *mpix_flowCart*. 142

Figure B.1: Cache-blocking Jacobi3D Pseudo-code. 154
Figure B.2: Vectorized and cache-blocking Jacobi3D Pseudo-code. 155
Figure B.3: Linear rank mapping across nodes. 156
Figure B.4: Cubic rank mapping across nodes in Jacobi3D. 156

Figure C.1: Different level of multi-threading in MPI. 159
Figure C.2: Rendezvous (BTE protocol) and Eager (FMA protocol) strategies employed by the MPI library. 162
Figure C.3: Performance of the baseline Flat-MPI compared to multi-threaded MATE variants, as message sizes decrease and total message count increase. . . . 163
Figure C.4: Multi-threaded UPC++ can own different personas to avoid the need for process-wide mutual exclusion mechanisms. 165

Figure D.1: Consecutive Balancing Algorithm. 167
Figure D.2: Shuffling Balancing Algorithm. 168

LIST OF TABLES

Table 2.1: Comparison between programming languages and libraries. 28

Table 3.1: Comparison of the distributed programming models analyzed in this chapter. 56

Table 5.1: Comparison of computation times and L2 cache misses on 2k cores. 91
Table 5.2: Ratio of computation, buffering, and communication costs on 1024 nodes of Cori Phase I (32k cores) vs Cori Phase II (64k cores) for the Basic-MPI variant. 95
Table 5.3: Ratio of computation, buffering, and communication costs on 1024 nodes of Cori Phase I (32k cores) vs Cori Phase II (64k cores) for the MATE variant. 95

Table 6.1: Ratio of computation and communication on 64, 256, and 1024 nodes for the Basic-MPI and MATE variants. 113

Table 7.1: Ratio of computation, buffering, and network communication operations on 128, 256, and 512 nodes for the Basic-MPI and MATE variants on Cori Phase II. 127

Table A.1: Side by side description of our three computational testbeds. 150

ACKNOWLEDGEMENTS

I owe a debt of gratitude to my Ph.D. advisor, Prof. Scott Baden. He taught me to think like a scientist, write like an academic, and behave like a professional. In times of complacency, he was stern and demanding; in times of self-doubt, he was kind and encouraging. He gave me enough leeway to develop my ideas but always made sure I stayed on track. He has generously dedicated countless hours to reviewing, discussing, and providing me with feedback. I am ever thankful for his guidance and friendship, and I aspire to become a scholar of his stature someday.

I am grateful to Prof. George Porter for kindly agreeing to serve as my committee's Co-Chair, and giving me valuable advice on my research and qualifying examinations. I am also thankful to my committee members, Profs. Tajana Rosing, Sutanu Sarkar, and John Weare for reviewing this dissertation and ensuring it meets the highest scientific standards.

I have been blessed with the opportunity to be part of the Computer Languages and Systems Software (CLaSS) Group at the Lawrence Berkeley National Laboratory, under the direction of Prof. Baden. At Berkeley, I worked alongside world-class scientists and researchers in the high-performance computing area. I am especially grateful to Dr. Tan Nguyen, Dr. Dan Bonachea, Dr. Paul Hargrove, Dr. Hadia Ahmed, Dr. Costin Iancu, and John Bachan who gave me invaluable feedback on my research and parts of this dissertation. I am also happy to have spent a summer with the ROSE Group at the Lawrence Livermore National Laboratory, under the supervision of Dr. Dan Quinlan.

Throughout this journey, I met a multitude of amazing people, a handful of whom I now consider my friends. My colleagues in arms from UCSD's CSE Department, Rami Kıcı whose invaluable friendship I could always count on; John Sarracino and Marc Andrysco with whom I shared many beers, conversations, and chess games; and Marc's wife-to-be, Suzanne Dunai, whose positive spirit and support I cherish. I am also lucky to have met my friends from the MAE department, Prakrit Dhillon and Aekaansh Verma (now in Stanford) with whom I shared many adventures, including carrying a mini-fridge for miles at night.

I want to thank Prof. Alex Orailoglu for giving me the opportunity to serve as his teaching assistant for CSE240A (Computer Architecture) during the Fall '16 quarter. I learned many valuable lessons from Prof. Orailoglu both personally and through his teaching methodology and had a fantastic experience guiding students through the course's material.

I received the support of amazing professionals: Dr. Maiken Gale, in La Jolla, and Dr. Arlene Marcus, in Berkeley, that helped me find a work/life balance while keeping my sights on the goal. I am also thankful to Dr. Rhonda Hackshaw from UCSD's CAPS office and to Prof. Victor Vianu for their continuous care and concern for my success in the Ph.D. program.

I am in debt to my first mentors, Profs. Graciela de Luca and Pepe Casas. Ever since I was an undergrad student at the Universidad Nacional de La Matanza, they have given me every opportunity to grow as a scientist and encouragement to pursue my academic goals.

Lastly, I could not have gotten this far without support from my family, friends, and girlfriend (now wife) in Argentina. Even from afar, their love and encouragement gave me a reason to stay motivated, pulled me through difficult moments, and reminded me why and who am I doing this for. They have my unconditional love.

Chapters 3, 4, 5, 6, and 7 are, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

Chapter 4 is, in part, a reprint of the material contained in the article: "*Toucan - A Translator for Communication Tolerant MPI Applications*", by Sergio M. Martin, Marsha J. Berger, and Scott B. Baden, which appears in the Proceedings of 1st International Parallel and Distributed Processing Symposium (IPDPS 2017), Orlando, FL, USA, June 2017. This dissertation's author was the primary investigator and author of this paper.

This research was supported by the Advanced Scientific Computing Research office

of the U.S. Department of Energy under contract No. DE-FC02-12ER26118 and DE-FG02-88ER25053. This research was also supported in part by the Fulbright Foreign Student Program grant from the U.S. Department of State.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), located at the Lawrence Berkeley National Laboratory and supported by the Office of Science of the U.S. Department of Energy.

VITA

2010	B. Eng. in Informatics, Universidad Nacional de La Matanza
2010-2014	Teaching Assistant, Universidad Nacional de La Matanza
2013	M. Eng. in Informatics, Universidad Nacional de La Matanza
2018	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

S. Martin, S. B. Baden, "MATE, a Unified Model for Communication-Tolerant Scientific Applications", *31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018)*, Salt Lake City, UT, United States. October 2018.

S. Martin, M. J. Berger, S. B. Baden, "Toucan - A Translator for Communication Tolerant MPI Applications", *31st International Parallel and Distributed Processing Symposium (IPDPS 2017)*, Orlando, FL, United States. June 2017.

F. Tinetti, S. Martin, "Optimizing a GPU Algorithm through Hardware Profiling Analysis", *2014 International Conference on Computational Science and Computational Intelligence (CSCI 2014)*, Las Vegas, United States. March 2014.

S. Martin , F. Tinetti, N. Casas, G. De Luca, D. Giulianelli, "N-Body Simulation Using GP-GPU: Evaluating Host/Device Memory Transference Overhead", *XIX Congreso Argentino de Ciencia de la Computación (CACIC 2013)*, Buenos Aires, Argentina. October 2013

ABSTRACT OF THE DISSERTATION

MATE, a Unified Model for Communication-Tolerant Scientific Applications

by

Sergio Miguel Martin

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Scott Baden, Chair
Professor George Porter, Co-Chair

We present MATE, a new model for developing communication-tolerant scientific applications. MATE employs a combination of mechanisms to reduce or hide the cost of network and intra-node communication. While previous approaches have been proposed to either source of communication overhead separately, the contribution of MATE is demonstrating the symbiotic effect of reducing both forms of data movement taken together in a single unified model. We explain the rationale behind our model and show its effectiveness in three scientific computing motifs on up to 64k cores of the NERSC Cori supercomputer. Lastly, we show how MATE can improve the workload balance of an irregular multigrid solver.

Chapter 1

Introduction

Supercomputers play a fundamental role in a wide range of scientific fields such as weather forecasting [72], medicine [39], computer-aided design [45], military [74], and simulation of natural disasters [29]. As scientific applications become larger and more sophisticated, however, so does their need for increasingly powerful supercomputers.

Fortunately, continual developments in both computer architecture and interconnects have led to an exponential growth in the performance of top supercomputers, increasing by an order of magnitude roughly every decade [114]. Currently, the most powerful computer, Oak Ridge National Laboratory's *Summit*, delivers a staggering peak performance of 187 *petaflops* (1.87×10^{17} floating point operations per second). The upcoming milestone, the first *Exascale* (10^{18} flops) supercomputer is on the horizon.

Exascale supercomputers will enable new breakthroughs in science and industry [38, 9]. However, due to their unprecedented number of processing elements and the complexity of memory hierarchies and interconnect, these computers will demand extraordinary efforts from system designers, network architects, and application programmers alike [28]. Of the many hurdles involved, coping with the ever-growing costs of communication is, perhaps, one of the most daunting, especially on applications that are memory-bound or sensitive to network latency.

The goal of our research is to facilitate the reduction of *Intra-Node* and *Network* communication costs on large-scale scientific applications. These costs have a notable impact in the performance at large scales (*e.g.*, communication cost makes up to 68% of the running time in non-optimized *De Novo* Genome assembly codes [43]).

Traditional programming models and libraries, such as *MPI* [103], have provided intuitive and scalable ways to develop distributed memory scientific applications for decades. However, these models have become increasingly incapable of providing mechanisms for tolerating the large communication overheads in modern supercomputers. Although several new models have been proposed to help reduce either (intra-node or network) source of communication cost, none of them have established a way to tackle both costs taken together. As a consequence, communication-tolerant applications require the use of hybrid strategies that combine multiple models, making them difficult for domain-area experts to develop.

To address these shortcomings, we have developed *MATE*, a programming model that reduces all sources of communication overhead in scientific applications. MATE provides hierarchical decomposition and dependency-driven semantics that support communication reducing performance programming within a single unified model. MATE's unified model enhances its communication-reducing benefits while avoiding the complexity of mixing two programming models. We have implemented MATE as a programming framework comprising an annotation model, a source code translator, and a runtime system library.

In this dissertation, we explain the rationale behind MATE's mechanisms and show its benefits on three scientific computing motifs on up to 32k cores of the NERSC Phase I and 64k cores of the NERSC Cori Phase II supercomputers. Additionally, we show that MATE can improve the workload balance of an irregular multigrid-based solver.

Research Contributions

- We introduce MATE, a new approach to developing distributed scientific applications that integrates communication-reducing mechanisms into a single programming model, providing a benefit that is greater than the sum of the parts.
- We show how our model supports hierarchical decomposition and locality models that enhances the ability of overdecomposition to deliver communication/computation overlap, while managing the growth in intra-node data motion without the need for a hybrid programming model.
- A programming framework comprising of a source-to-source translator, annotation syntax, and runtime system that exposes MATE's communication-reducing mechanisms while requiring modest change to an MPI application source code.
- We show that MATE can realize a noticeable reduction in communication cost in three large-scale application motifs, exceeding the performance of other communication-tolerant models.
- We show how MATE's mechanisms can reduce the cost of communication of complex applications, such as Cloverleaf3D, in which a manual refactoring for overlapping communication and computation would be impractical.
- A new approach to reducing load imbalance in irregular applications that combines the benefits of static and dynamic load balancing approaches.

Dissertation Outline

Chapter 2 introduces the background and motivation for our work, including the challenges and main approaches to developing communication-tolerant applications and the need for a unified model for communication-tolerant programming.

Chapter 3 describes the MATE model and its communication-reducing mechanisms and explains how MATE improves over prior work on communication-tolerant models. Chapter 4 details the implementation and design of MATE's programming framework.

Chapters 5, 6, and 7 provide experimental results on three scientific application motifs on up to 64k cores of the Cori Phase II supercomputer. The first test case, *Jacobi3D* represents a Jacobi-based stencil solver for the Poisson equation, the second test case, *Cannon's Algorithm* represents a dense matrix-matrix multiplication algorithm, and the third case *Cloverleaf3D*, represents a Fortran-based hydrodynamics benchmark.

Chapter 8 introduces the cost of work imbalance and how MATE can reduce its impact.

Chapter 9 enumerates the current limitations of the MATE model, discusses future research directions, and provides concluding statements.

Chapter 2

Background and Motivation

2.1 Overview

In this chapter, we describe the anatomy of modern supercomputers and introduce the most widely used programming paradigms for developing scientific applications. Second, we discuss the sources of communication cost in large-scale computing and how to measure them. Third, we describe the challenges and proposed solutions for developing communication-tolerant applications. Lastly, we provide a future outlook on the costs of communication and how they motivate our work.

2.2 Anatomy of a Supercomputer

Today's largest Petascale supercomputers comprise $O(10^7)$ processor cores, and studies estimate that future Exascale supercomputers will require $O(10^8)$ cores [7]. To achieve this degree of concurrency, supercomputers need to grow in two ways. First, they need to increase the number of nodes connected through a high-speed interconnect. Second, they need to increase the number and efficiency of the processor cores inside each node.

2.2.1 Node Architecture

A *node* is the main computing component of a supercomputer. Nodes are server-grade computers that contain: (a) one or more multi-core processors and/or many-core devices, (b) a memory hierarchy, and (c) a network interface controller (NIC). Fig. 2.1 shows a representative example of the processor topology and memory hierarchy of a computing node from *Cori Phase I* [107], a Cray XC40 system located at the National Energy Research Scientific Computing Center (NERSC).

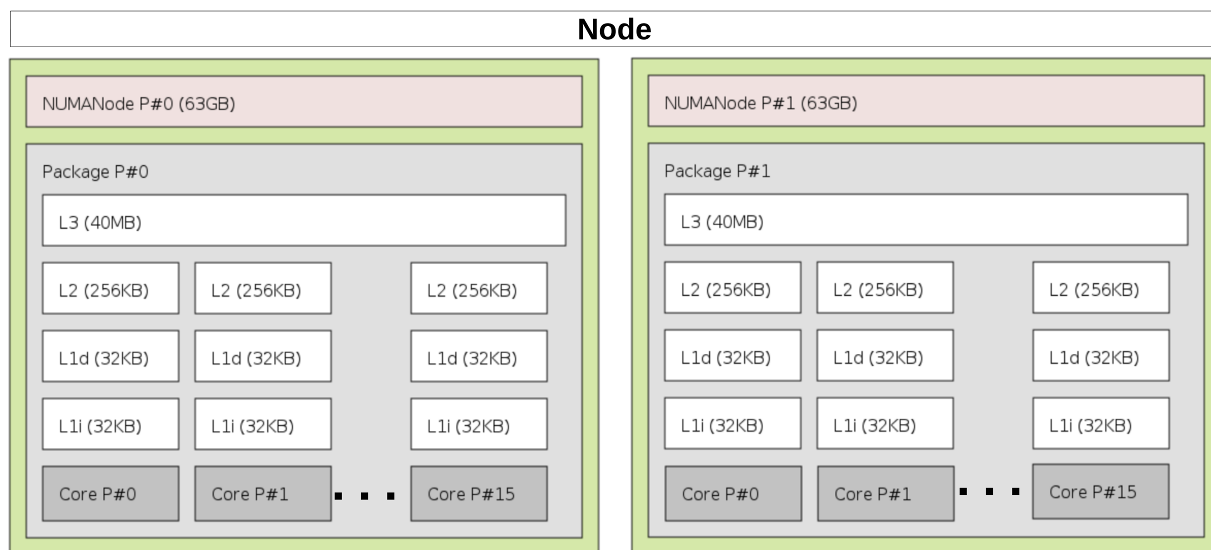

Figure 2.1: Configuration of a 32-core NERSC Cori Phase I (Haswell) Node.

In Cori Phase I, each node contains two 16-core Intel "Haswell" processors, identified as packages P#0 and P#1 in the figure. A *package* or socket corresponds to a processor chipset on the node's motherboard that provides connectivity to other components of the node (*e.g.*, RAM, PCI-e channels, NIC). Some supercomputers also employ massively parallel *many-core* processors (*e.g.*, NVIDIA Volta [108] and Intel Xeon Phi [100] architectures) as part of their compute nodes. These devices base their potential in implementing processors with large number of lightweight cores that excel at vector operations.

Similar to conventional computers, supercomputer nodes expose a memory hierarchy

comprised, at its top level, by a node-wide RAM accessible by all the cores in the node. This memory is distributed into different segments, called *Non Uniform Memory Access* (NUMA) domains. The organization of NUMA domains mirrors physical constraints. For example, in the case of Cori Phase I nodes, each of its two sockets is directly connected to an array of memory modules (totaling 63GB each, for a 126GB total), representing a NUMA domain, while a fast *data bus bridge* connects the two NUMA domains. Consequently, cores accessing memory residing in a different NUMA domain will suffer from a higher latency, hence the non-uniformity. At the lowest level, each Haswell processor contains core-specific L1 instructions and data and L2 cache structures (32kb and 256kb, respectively) and an L3 cache structure (40MB) shared between all cores.

2.2.2 Interconnect Design

Unlike conventional workstations or personal computers, large-scale supercomputers do not provide a cache-coherent global memory address space. Instead, the total available system memory comprises the interconnected union of the disjointed memories of its constituent nodes. This configuration requires the use of an efficient network interconnect that allows for a fast exchange of data among any two nodes. To enable this connectivity, network architects design and employ *network topologies*. These topologies describe the way in which to organize nodes, routers[1], and cabling. The job of a network architect is to design topologies that minimize the cost of communication while keeping power and money budgets constrained.

Traditional network topologies, such as the *Folded Clos* (fat-tree) [62], proven to be efficient in smaller scales, incur a prohibitive cost for exascale supercomputers because of the number of routers and cabling complexity required. Such a network would dominate the costs of a supercomputer, both in budget and energy costs [55]. On the other hand, highly-scalable topologies for Petascale and Exascale supercomputers have been proposed to reduce the com-

[1]Network devices that forward data packets between nodes.

plexity of their interconnect. The *Flattened Butterfly* [56] and *Dragonfly* [57] topologies have proven to be less costly in the number of nodes thanks to the use of optical cables in global channels which have been proved to achieve high-bandwidth connectivity with reduced router and cabling costs.

The *Aries* interconnect [106] in Cray XC40 supercomputers, for example, uses a dragonfly topology that is organized in a four-level hierarchy. *Level 0* consists of four nodes allocated inside one custom-designed integrated circuit (blade). Communication inside a blade is driven through a high-speed router that serves as the main gateway for all four nodes. *Level 1* consists of 16 blades allocated inside one chassis. Communication is routed through high-bandwidth wires across a circuit board that connects all blade routers. *Level 2* consists of 3 chassis allocated inside one cabinet. Communication is managed by a blade router connected by copper cables. Finally, *level 3* represents the entire supercomputer and uses high-bandwidth optical cables and routers. This configuration guarantees a maximum of four hops[2] required to communicate a message from one node to another.

2.3 Costs of Communication

In this section, we analyze the two primary sources of the cost of communication in a distributed memory scientific application: *Intra-Node*, and *Network* communication.

2.3.1 Intra-Node Communication

We define intra-node communication as data transfers between cores within the same node. This operation brings a negative impact on performance as processor cores spend time producing and waiting for data that could otherwise be spent in performing useful computation.

The cost of intra-node data motion has been growing steadily over time due to tech-

[2]The re-transmission of a data packet between intermediate routers connecting any two nodes.

nologic factors. Until about a decade and a half ago, increasing computational performance depended almost exclusively on single core processors to deliver higher clock speeds. However, after the Intel Pentium's 4 Tejas processor was canceled in 2004 [40], it became evident that increasing the complexity and power of single-core processors to meet the growing demands was no longer possible; we had hit the infamous power wall [84], the point at which heat output and energy consumption made it impossible to keep increasing the clock frequency of CPUs. This limitation motivated the development of multi-core and many-core processors architectures where multiple, yet simpler, cores collaborate under the same memory space.

However, as multi and many-core processors become a staple in modern supercomputers, increasing their computational power, this progress has not been accompanied by a similar increase in main memory performance. The asymmetry between computational and memory performance is known as the *memory wall problem* [118] and still represents a limiting factor for many large-scale scientific applications.

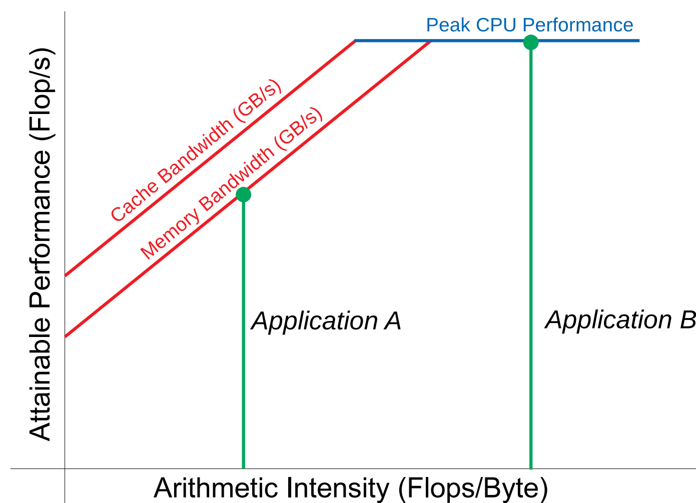

Figure 2.2: Example roofline diagram that compares two applications. Application *A*, limited by memory bandwidth, and Application *B*, limited by the peak CPU performance.

The relationship between processor and memory performance on a scientific application can be measured using the *Roofline Performance Model* [116]. This model uses three inputs: (1) the arithmetic intensity of an application (*i.e.* the number of floating point operations computed

per byte of data transmitted from/to memory, measured in flops/byte), (2) the peak CPU performance, and (3) the peak memory bandwidth. The output, a roofline diagram as shown in Fig. 2.2, offers an intuitive way to visualize the maximum attainable performance of an application executing in a multi-core system.

The point at which arithmetic intensity meets the roofline function - delimited by the maximum memory bandwidth and CPU peak performance - indicates the application's maximum attainable performance (measured in flop/s). The example in Fig. 2.2 represents the estimated performance of two applications. Application B has a higher arithmetic intensity (right side of the diagram) and is only limited by the peak CPU performance. On the left, application A has a lower arithmetic intensity (requires a higher transfer rate from/to memory) and its performance is therefore limited by memory bandwidth.

The negative impact of the memory wall problem in scientific application grows as the performance gap between processors and main memory keeps widening [46]. This phenomenon has a severe impact when on performance, especially on applications that are limited by memory bandwidth. The ever-increasing core per node counts exacerbate this problem as each core puts additional pressure on main memory. Current multi- and many-core systems are particularly affected by this problem, forecasting a performance bottleneck in Exascale systems as well.

2.3.2 Network Communication

The cost of network communication comes from the time spent on data transfers among processes residing in different nodes across the network. As supercomputers grow larger in the number of nodes and the size of their interconnects, this cost overhead becomes a more significant part of an application's running time. On distributed memory applications, processes residing in different physical address spaces need to move data explicitly over the network. Since these operations transcend the boundaries of a node's memory, their cost typically exceeds that of intra-node communication. For this reason, reducing the impact of network communication

on large-scale applications will be crucial to achieving Exascale performance.

The *LogP* model [30] offers a general approach to estimate the impact of network communication in a distributed application. The name of the model, *LogP*, is formed by the four arguments that it takes as input: L represents an upper bound on the *latency* incurred in communicating a message containing word from one node to the other; o represents the *overhead*, defined as the time a processor is busy in the transmission or reception of the message; g represents the *gap*, defined as the interval between consecutive message transmissions or receptions at a processor, and; P represents the number of processors or nodes in execution.

LogP provides an in-depth analysis of the impact of network communication, even in complex applications such a distributed FFT algorithm [27]. A simpler model for estimating this cost, the $\alpha\beta$ model [87], however, provides enough insights to support our explanation. The $\alpha\beta$ model represents the cost of network communication as the time taken (T) for any data transfer to be transmitted from one node a to another node b as described by the formula: $T_{a,b} = \alpha + \beta \times n$.

Where: $\alpha = L_{a,b}$, and $\beta = 1/B_{max}$. The value of $L_{a,b}$ represents the latency between two nodes a and b, n represents the size of the message, and B_{max} represents the maximum bandwidth capacity of the interconnect. The latency factor can be estimated from the number of routing hops ($H_{a,b}$) required to connect both nodes multiplied by an estimated cost per hop (h), plus a fixed overhead per message (s), as shown in Eq. 2.1. The fixed per-message overhead (s) is caused by both software (*e.g.*, a communication library can require filling a software buffer before sending) and hardware factors (*e.g.*, network interface start-up time)

$$L_{a,b} = H_{a,b} \times h + s \tag{2.1}$$

One observation we can draw from the $\alpha\beta$ model is that, for small message sizes, the cost of latency dominates the overall cost of communication. It follows that a large per-message overhead can significantly impact performance on algorithms that require sending many small-sized messages. Indeed, we have observed this phenomenon in some of our test cases (see

chapter 8). We have also observed that contention for access to the communication library or the network interface in multi-threaded applications can also increase per-message overheads, causing an adverse effect on performance. We analyze this phenomenon in further detail in Appendix C.

2.4 Distributed Programming Paradigms

To muster the computational power of a supercomputer, developers design their applications guided by a *distributed programming paradigm*. These paradigms describe the way in which a distributed memory parallel application computes a solution, how it distributes its workload among system's resources, and prescribes the logic for communication and synchronization. Several of these paradigms have been proposed, each with their own set of advantages and disadvantages. A careful choice of programming paradigm plays an essential role in attaining the maximum performance for a given application.

2.4.1 SPMD

Single (Same) Program, Multiple Data (SPMD) [10, 33] is one of the most used paradigms in scientific computing. The fundamental characteristic of an SPMD application is that it distributes its workload among smaller partitions that can be solved in parallel by multiple processes executing the same program. Thanks to this principle -assuming no bugs or semantic errors–, an SPMD application that executes correctly on a multicore processor can also execute correctly on the millions of cores of a supercomputer [13].

A typical SPMD application decomposes its workload evenly[3] among the system's computational resources. The workload is split into a set of n partitions, each solved by one of n

[3]Even distribution is a preferable condition rather than a prescription of the SPMD paradigm. Indeed, some problem domains cannot be evenly divided due to their irregular topologies. In those cases, the problem of load imbalance occurs, where some nodes or cores perform more work than others and causing an overall decrease in performance. We discuss this phenomenon in Chapter 8.

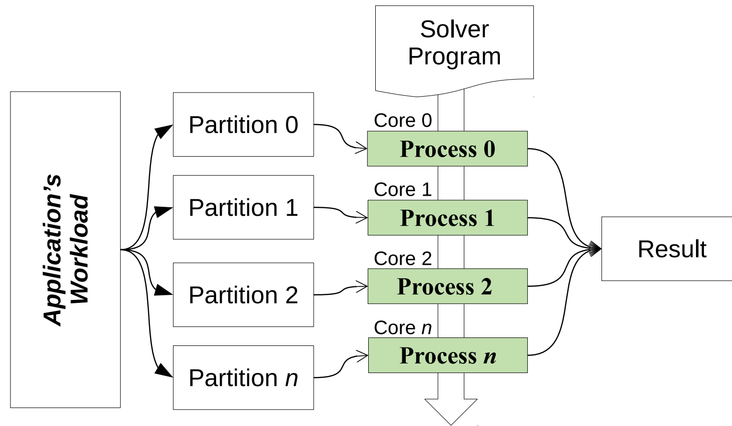

Figure 2.3: Execution model of an SPMD application.

processes, as shown in Fig. 2.3. Each process is presumed to use a different computational resource (*e.g.* a processor core) and live in its own address space in which its partition is stored. During the lifetime the application, processes regularly exchange data through message passing. An SPMD application finishes its execution when all processes have finished their part of the workload, contributing to the final result.

The *Message Passing Model*, expressed through its community-based standard *Message Passing Interface* (MPI) [103], is the most widely used model for writing SPMD applications. Most implementations of the MPI interface (*e.g.* *MPICH* [104], *MVAPICH* [105], *openMPI* [110], *Intel MPI* [99], *Cray MPI* [97]) instantiate each MPI rank as a separate operating system process, providing a support for address space isolation.

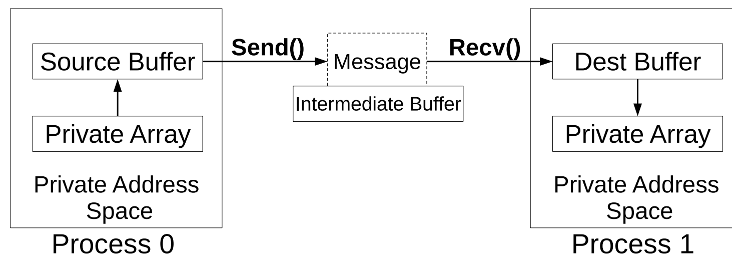

Figure 2.4: Two-sided Communication Protocol.

Since MPI applications run across separate memory spaces, processes exchange data using a *two-sided* (a.k.a., *Message Passing*) communication protocol, which provides a way to

exchange messages between two processes while also providing a synchronization mechanism. In the *Two-Sided Communication* protocol, two processes, *sender* and *receiver*, explicitly agree to participate in the exchange of a message. The protocol requires that every send request issued by a sender process matches a receive request in the receiver process, as illustrated in Fig. 2.4. The synchronization aspect of this protocol comes from guaranteeing that the receiver process is ready before copying the data into the destination buffer[4].

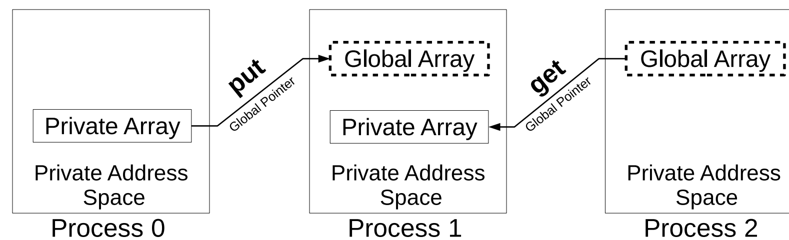

Figure 2.5: One-sided Communication Protocol operations: *put* and *get*.

Later specifications of MPI (now at MPI-4.0 [102]) expose globally-shared allocations, called *windows* that enable *one-sided communication* (a.k.a., *Remote Memory Access*) operations. These operations allow a process to perform read or write operations on the memory space of another, typically by specifying a global pointer, without the need of a reciprocal request from the remote process. A process can perform two operations, *put* and *get*, that communicate data to/from private and global pointers. A *put* operation copies data from a private array in the issuing process onto a global array in the receiving process. A *get* operation pulls data from a global array in another process into the private space of the issuing process. Both operations require the use of global pointers to indicate the remote memory position they access.

[4]Under certain conditions, MPI allows the sender to issue a message earlier, storing it in an intermediate buffer until the receiver is ready. Although this approach allows the process to continue earlier, it also incurs in additional data motion, which is one of the primary costs of communication (see section 2.3.1).

2.4.2 PGAS

The *Partitioned Global Address Space* (PGAS) model [31] provides a framework for global memory that is meant to execute in multiple disjoint physical memory spaces. The PGAS model seeks to integrate the simplicity and scalability of the SPMD paradigm with the benefits of a shared memory communication model by introducing the concept of a *global address space*. A global address space allows a PGAS program to make a distinction between memory allocations which are *private* (*i.e.*, only accessible by a given process) and those which are *global* (*i.e*, accessible by all tasks in the execution).

A global allocation may be located either within the address space of a single process or partitioned across multiple address spaces. A process accesses a global allocation by obtaining its *global pointer* plus an offset. The global pointer encapsulates the information of the allocation, required to resolve the owner process and local pointer of the referenced element. The actual location of a partition (*i.e.*, the process that contains it) does not affect the correctness of a PGAS program since global pointers are equally accessible by all processes.

Unified Parallel C (UPC) [12], is a PGAS extensions to the C language (similar extensions have been developed for other languages as well, such as: *Coarray Fortran* [53], and *Titanium* [74] for Java]). UPC allows the creation of global arrays that are split into partitions, each belonging to a particular process, specified through its *affinity*.

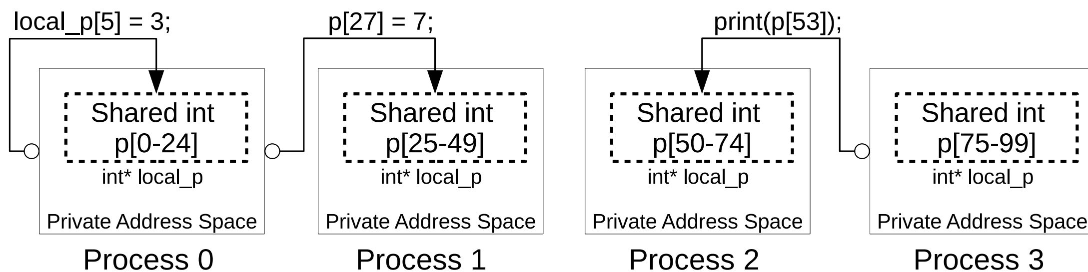

Figure 2.6: Partitioning of a shared array in UPC and examples of communication.

UPC's model takes advantage of language structures to define communication. Instead

of having the programmer insert explicit calls to communication routines, UPC performs a one-sided communication operation implicitly every time the code reaches an expression or assignment statement involving a remote partition. Fig. 2.6 shows an example of a UPC application that partitions a globally accessible allocation containing 100 integer elements among four processes, each containing 25 elements in their private address space. For large messages, however, individual accesses can produce large overheads. To address this issue, UPC also provides support for one-sided communication that only requires a single operations.

2.4.3 Asynchronous PGAS

The *Asynchronous Partitioned Global Address Space* (APGAS) model extends the PGAS model with ideas from task-based asynchronous execution model [83], which describes the semantics of an application as a hierarchy of tasks that are dynamically created and scheduled during the execution time.

A task-asynchronous application starts its execution as a set of initial tasks. At certain points of execution, tasks reach a bifurcation point in which sub-tasks are created. Sub-tasks execute a specific function or subroutine and can be instantiated in either the same process as their parent task or in a remote process. A programmer can specify the parent tasks to suspended until every generation of successive subtask finish. This behavior can be nested so that each subtask themselves can extend the execution hierarchy.

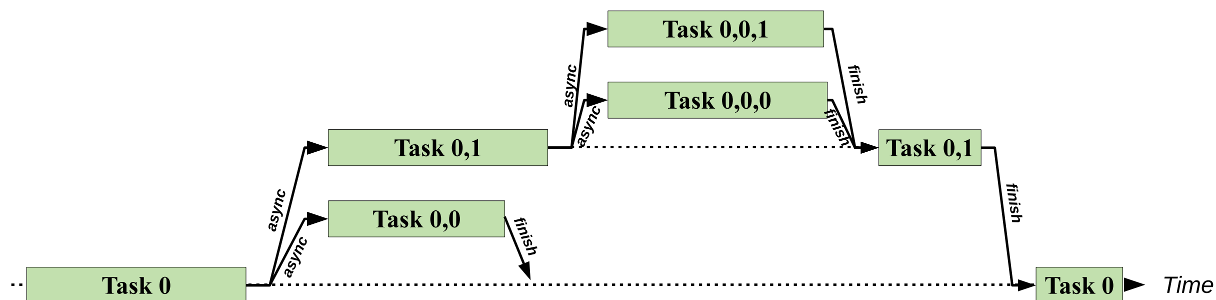

Figure 2.7: Execution timeline of an asynchronous application.

Fig. 2.7 shows an execution graph of a task-based asynchronous program with multiple levels of descendant tasks. Parent tasks suspend their execution only to resume after all their descendant task finish. For example, Task 0,1 resumes immediately after Tasks 0,0,1 and 0,0,0 finish execution. The same happens with Task 0, after Task 0,0 and 0,1 finish.

X10 [23] is an task-based object-oriented PGAS language that introduces the notion of an *async* function. Async represents a *Remote Procedure Call* (RPC)[5], operations that execute a function at a local or remote process. X10 also defines a *finish* function in which a parent task can then be set to wait for one or multiple children tasks. The finish function represents an RPC incoming from a remote process to signal the completion of a previously instantiated task. X10 expresses locality through the notion of *places*, logical abstractions that identify the physical resource where a task is executing. By specifying a place, allocations can be performed locally (same place as calling task), or remotely (in a different place).

The *Habanero* [21] project implements the concepts derived from X10 into standard languages such as Java, C, and C++. Habanero also serves as a platform to test the performance of task-based asynchronous paradigm on new communication libraries [60].

UPC++ [121] is a C++ library to that abstracts the complexity of the underlying PGAS communication layer by providing an intuitive programming interface. In UPC++, processes exchange data only through one-sided communication operations. To verify that a one-sided operation has finished, UPC++ returns every *rput* and *rget* operation as a future object that stores the status of the operation. The underlying communication layer updates the completion status of a future as soon as the operation finished, while the future exposes methods to verify its completion.

Charm++ [53] is a parallel programming framework that extends the standard C++ syntax with language artifacts based on the *Charm* programming language [78]. Charm++ provides an object-based API that defines tasks as *chares*, programmer-defined C++ classes that inherit

[5]RPCs are similar enough in concept to other terms coined in the literature such as *Active Messages* [91] and *Remote Method Invocation* (RMI) [53], and we consider them as synonyms in the context of our work.

a set of base chare methods. Chares evaluate locality by residing in a *processing element* (PE). PEs indicate the physical node or processor that is in charge of executing the chare. Communication occurs during the remote invocation (RPC) of a chare. When the invocation passes a global pointer as a parameter, the Charm++ runtime system will make sure the data referenced by the pointer is available to the remote chare's processing element (PE) before invoking the procedure at the receiving end.

2.4.4 Dataflow

The *Dataflow* paradigm defines the semantics of a program as a producer/consumer relationship. This paradigm has been adopted from the design of out-of-order processors (the earliest instances of such processors were the *IBM System/360*, implementing *Tomasulo's Algorithm* [89], and the *CDC 6600*, implementing the *Scoreboard algorithm* [88].) and then proposed for high-level parallel and distributed applications.

A Dataflow program runs as a set of *tasks*. A task represents a section of the code (statements, regions, or entire procedures) that produces data elements as a result of its execution upon which other tasks depend. Tasks do not execute until all their data dependencies become available [1, 36]. A Dataflow program provides a static description of a directed acyclic graph (DAG), where tasks represent nodes and edges represent data dependencies. The DAG prescribes the execution order and concurrency of the tasks, rather than their placement order in the code.

Fig. 2.8 shows an example application that describes an iterative loop with three tasks (*t0*, *t1*, and *t2*), and four data elements (*A*, *B*, *C*, and *D*). The initial *t0* produces *A* and *B*, which are consumed by tasks *t1* and *t2*, respectively. In turn, tasks *t1* and *t2* produce the data elements required by *t0* to execute again. The termination of a program can be prescribed as a condition inside the code of any such tasks.

Most Dataflow models handle communication implicitly, without the intervention of the programmer. That is, the compiler or the runtime system decides the flow of data and which

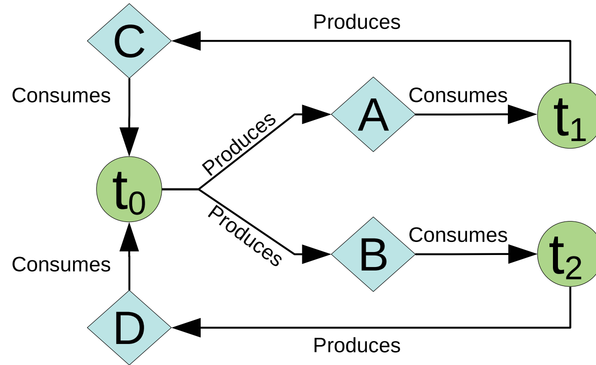

Figure 2.8: Example DAG in a Dataflow application.

communication mechanisms to use. This high-level description of a distributed application facilitates the programmer's job, since it only requires a description semantics of the program and its data dependencies, without the need for communication semantics. On the other hand, since Dataflow applications typically resolve their communication strategy in real-time, this may result in lower performance when compared to an application that uses explicit communication operations.

Swift [117] is a C-like language and compiler that constructs the underlying DAG of a program through a fine-grained analysis of its statements. These statements will not execute in the order they are placed but instead scheduled to execute based entirely on their inter-dependencies. During execution, Swift's runtime system creates one task for each statement in the program and manages their data dependencies in real-time, while optimizing data and task locality.

The *Legion* [16] programming language schedules entire tasks based on data availability. A Legion program makes a distinction between data and task semantics. The program groups data elements into regions with different privilege levels (exclusive, simultaneous, among others) that determine whether data can be freely accessed by multiple tasks simultaneously or in strict order. The order in which data objects are processed is defined by the dependency graph as described statically by the programmer in the code.

SMPSs [76] is a task-based dependency model now used in OpenMP Tasking [73]. The

SPMSs model describes the semantics of a program as a set of functions that can be executed as soon as (and only if) their input dependencies are satisfied. Programmers specify dependencies by prepending *#pragma* to the definition of a function. Each function may have *input*, *output*, or *input/output* dependencies. An input dependency is satisfied when a function with an output dependency of the same name finishes execution.

The *Tarragon* model [26] uses a coarse-grain dataflow model [11] where tasks represent entire atomically-executed procedures, instead of single statements. The DAG in Tarragon connects different tasks (nodes) with directed edges among them. Each edge specifies a message to be sent or received by a set of two tasks. The set of incoming edges comprise a task's *firing rule*. When the firing rule is triggered (*i.e.* all messages have arrived), Tarragon's runtime enables the task to execute. Research with the Tarragon model shows that, by describing data dependencies as incoming/outgoing network communication, a Dataflow application can hide the cost of latency communication [25].

2.5 Communication Tolerant-Programming

Although several programming models and paradigms have been proposed for developing distributed memory applications, the SPMD paradigm (and more specifically, MPI) continues to be the most widely used. The appeal of the SPMD paradigm it that it prescribes a way to distribute the workload that is independent of the way the solution is computed. In this chapter, however, we explain why this traditional approach is becoming mostly unable to handle the ever-increasing costs of communication in Peta- and Exascale applications. We discuss these limitations and the difficulties of developing communication-tolerant scientific applications.

2.5.1 Communication/Computation Overlap

Many SPMD applications use the *Bulk-Synchronous* model [92] to specify the way in which these processes communicate and execute. This model represents a powerful tool to describe the iterative nature of the majority of scientific application motifs. Its main contribution is the idea that an application can perform computation and communication in separate, independent steps. This abstraction between communication from computation logic makes it easy to develop new distributed applications as well as adapting legacy/sequential codes to distributed execution.

In a bulk-synchronous application, the behavior of each process is defined in *supersteps*. At each superstep, every process performs one, or a combination of two *substeps*: (1) compute, and (2) exchange partial results. The data required for the next superstep will not be available until *all* processes finish their current superstep. This behavior implies a barrier at the end of each superstep where all processes synchronize.

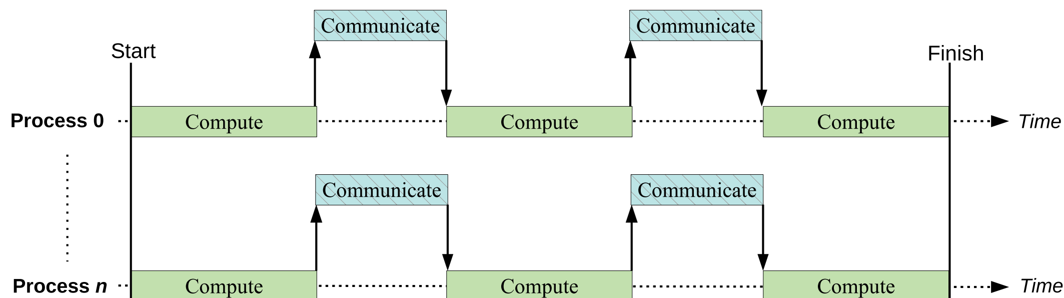

Figure 2.9: Core usage timeline of a process under the bulk-synchronous model.

Fig. 2.9 illustrates a hypothetical execution timeline of *n* processes under the bulk-synchronous model. The figure shows the effective core usage (*i.e.*, the time spent performing actual computation) solid blocks. The diagonally hatched blocks represent communication operations. An upwards arrow indicates the start of a message exchange request, and a downwards arrow represents its completion. Since all computation in this iterative solver is presumed to depend upon the arrival of data, the core remains idle during the communication phase.

The problem with this traditional approach is that, during communication operations, the computational resource (core) remains idle in wait for the arrival of data. Fig. 2.9 shows that none of the processes perform any useful work during the communication phase. This situation represents a waste of computational power since, hypothetically, cores could be instead used to perform useful computation during that time. Therefore, an application naively programmed under the bulk-synchronous model suffers from the full cost of network latency. The impact in performance due to core under-usage during the communication phase can be estimated with the formula in Eq. 2.1.

Some computational motifs are particularly susceptible to the cost of latency. Spectral methods (FFT) [18], for instance, require all-to-all communication patterns where all nodes need to communicate with each other regularly, making these algorithms difficult to scale efficiently. Therefore, it will be necessary for large-scale scientific applications to employ techniques or models that reduce or hide the cost of network communication as much as possible.

One way to reduce the impact of network latency is to manually restructure a program to enable communication/computation overlap by splitting the computing sections of the program into distinct communication dependent and independent sections, *e.g.* via *split-phase* coding [59]. This approach improves CPU usage and reduces the impact of communication and delays by keeping cores performing useful computation while data is being transmitted without the need for overdecomposition, resulting in a decrease in running time. Fig. 2.10 shows the effect of these transformations where data-independent computation is performed during the communication phase, resulting in better core usage and performance.

Although reducing the cost of network communication data motion involves primarily overlapping computation with communication, other approaches include communication reordering [61], concurrency optimizations [22], and communication avoiding algorithms [15, 44]. The main hurdle in implementing these approaches is, however, that they require significant domain-specific refactoring of the source code that may prove to be impractical in large ap-

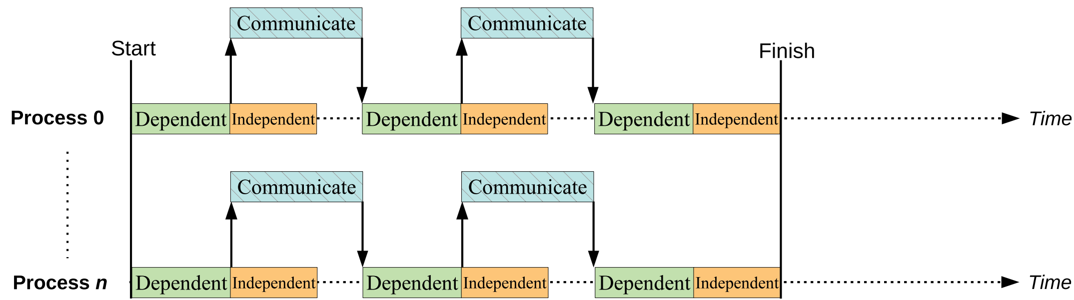

Figure 2.10: Core usage timeline of a split-phase application that employs separate communication dependent and independent computation to achieve communication / computation overlap.

plications. Furthermore, these approaches entangle application logic with implementation policy, refuting the main appeal of the bulk-synchronous model. As a consequence, implementing communication-tolerant applications by hand requires a painstaking amount of effort by the programmers.

2.5.2 Data Motion Reduction

The growing gap between the computational performance of multi/many-core processors and memory hierarchies demands scientific application developers to develop new solutions that reduce the impact and volume of data motion to improve intra-node performance. There are two ways in which a memory-bound application can improve its intra-node performance.

First, by making better use of the memory hierarchy. Many well-known techniques improve an algorithm's memory access patterns to reuse data in cache lines as much as possible [58], enabling them to reach cache bandwidth-bound performance (In Fig. 2.2, this means that application *A* extends upwards). These techniques often require refactoring (*e.g.* cache blocking) of the computational parts of the application.

Second, by reducing the volume of intra-node data motion. Unnecessary or excessive transference of data inside a node's physical memory space decreases an application's arithmetic intensity (moving application *A* leftwards, an adverse effect), especially on those that are already

memory bound. Reducing this cost represents a challenge since it requires refactoring both communication and computation aspects of the application.

Most implementations of the MPI interface instantiate each rank as a separate operating system process. The benefit of this configuration is that not only simplifies implementation but also guarantees that every process will operate in a private address space. Fig. 2.11 illustrates how $k \times m$ MPI ranks assigned to their own OS process across k nodes containing m cores and connected through a network.

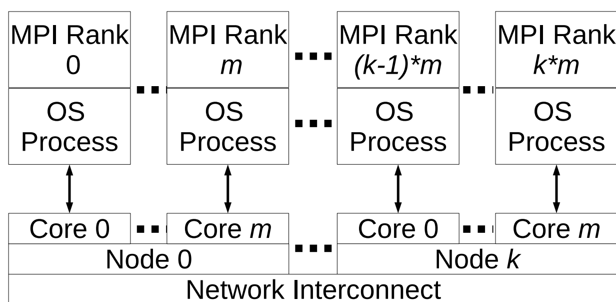

Figure 2.11: A typical deployment of an example MPI application with processes mapped to a single core.

The problem with process-isolation is, however, that it does not provide an intuitive way for MPI applications to make use of shared memory structures. MPI processes typically exchange messages involve in a series of steps: (1) the sender process packs data into a buffer, (2) sends the buffered message to the receiver's memory space, and (3) the receiver unpacks the message into its memory space. All these steps are necessary for communicating data between disjoint memory spaces, but impose unnecessary memory bandwidth pressure when performed inside a node, where data is already available in physical memory space, resulting in a loss of arithmetic intensity.

Threading libraries, such as OpenMP [73] and POSIX Threads [112], have been proposed and widely used to develop parallel applications for multicore architectures. These libraries operate by instantiating *kernel-level threads* (KLT), OS-scheduled structures that contain the execution state of a processor and lives inside the scope of an OS process. KLTs share the

same program and memory space with other KLTs living in the same process. Instantiating multiple KLTs that execute in parallel allows processes to make use of multiple cores collaboratively. The OpenMP interface provides a simple means to instantiate and synchronize the execution of threads, making it easy for a programmer to create a multi-threaded parallel application.

The motivation behind threaded models is that they enable the use of shared memory for communication. When two threads execute on the same node, they can operate on the same address space allowing neighboring threads to access data directly by without the need for buffering or explicit communication. KLT-based libraries are, however, limited to express parallelism within a single node and are thus unable to extend across distributed memory. Moreover, multi-threaded applications require synchronization mechanisms to prevent data hazards.

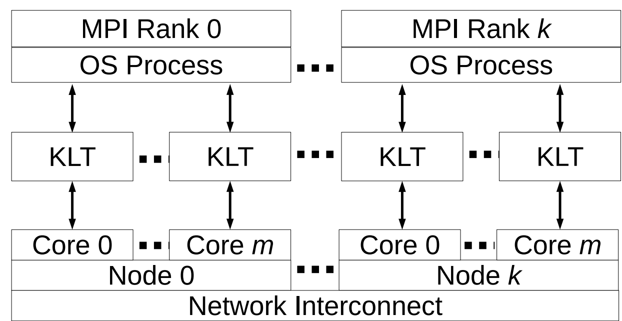

Figure 2.12: A typical deployment of a hybrid MPI+KLT application where each MPI process spans a group of kernel-level threads, each mapped to a single core.

Hybrid (*aka* MPI+X) models, such as the *MPI + Kernel Level Threads* (MPI+KLT) model [35], have been proposed as a way to address the intra-node data duplication problem while enabling a distributed execution. These models employ two levels of communication in which MPI manages inter-node communication while a threading model manages shared memory among threads inside each node, as shown in Fig. 2.12.

The MPI+KLT approach reduces the cost of intra-node data motion by having fewer MPI processes per node (typically, one per NUMA domain) which, in turn, instantiate as many threads as cores in the node. Threads belonging to the same process communicate through shared memory without the need for message passing. Some work has also been proposed to instantiate

MPI ranks themselves as kernel-level threads, also providing native shared memory access [85].

The disadvantage of the hybrid approach is that it requires combining a threading model with a messaging model for communication across a network, which poses two problems. First, it requires the design of data exchange and synchronization logic for both levels of communication, complicating the design of the application. Second, it may expose inefficiencies in the interaction between the two communication models (*e.g.*, require all threads to synchronize before issuing MPI calls and forcing serialization of communication operations).

2.6 Future Outlook

Across the last two decades, supercomputers have seen a steady increase in computational power due to a combination of more powerful multi-core/many-core processors and a scaling in node performance. However, as we reach the end of Moore's Law [69] era, this trend can no longer be supported by increments in the power of conventional processors [82]. This prospect suggests that, other than investing in alternative technologies, there will be an increased dependency in the node scaling of supercomputers to reach Exascale performance.

As the size of supercomputers grows, however, it becomes more likely that any two communicating nodes will be separated enough to require communication through global links. Therefore, messages will require, on average, a higher number of routing hops *(H)* to reach their destination, resulting in an increased overall cost of latency. As a consequence, the cost of network latency is expected to become a significant component of an application's running time. We can, therefore, infer that the separation between computation and communication phases during execution in the bulk-synchronous model will represent a critical constraint to the performance of Exascale applications.

Another trend in supercomputers is that their memory capacity fails to scale at the same pace as computing capacity. According to the Top500 supercomputer list [114], the current

#1 in the world, Oak Ridge National Laboratory's *Summit* supercomputer [109] provides 0.015 byte/flops ratio, whereas 2001's #1 supercomputer, Lawrence Livermore National Laboratory's ASCI White supercomputer, provided a 0.5 bytes/flop ratio. Fig. 2.13 shows the ratio for the top #1 supercomputers since 2001. The trend shown in the figure carries two consequences:

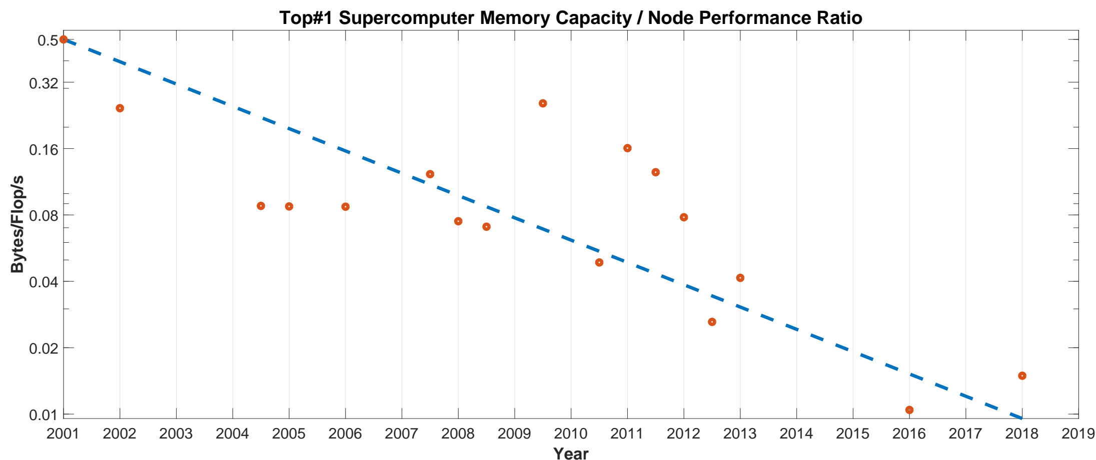

Figure 2.13: Memory/Performance ratio of supercomputers since 2001. Data source: [114]

First, future supercomputers will have steeper memory-bound rooflines due to the increased difference between peak CPU performance and peak memory bandwidth, making Exascale applications more likely to be memory-bound. Therefore, any loss in arithmetic intensity due to inefficient or redundant intra-node data motion will have a more detrimental impact.

Second, since the memory capacity of a node is decreasing relative to peak performance, distributed applications will compute their workload in shorter bursts, requiring a higher frequency and volume of inter-node communication. Therefore, the cost of latency will represent a more significant part of their running time. In conclusion, Exascale applications will suffer from both intra-node and network communication costs, forcing developers to tackle both overheads.

Several programming paradigms and communication models have been proposed to describe distributed memory applications that enable scientists to take advantage of the power of modern supercomputers. Table 2.1 shows a comparison of the features of the most widely used programming models for scientific applications. A wide range of distributed scientific applica-

Table 2.1: Comparison between programming languages and libraries.

Model	Paradigm	Communication				
		RPC	One-Sided	Two-Sided	Language-Driven	Implicit
MPI	SPMD		✓	✓		
UPC	PGAS	✓			✓	
Coarray Fortran	PGAS		✓		✓	
Titanium	PGAS		✓		✓	
UPC++	APGAS	✓	✓			
X10	APGAS	✓			✓	
Habanero	APGAS	✓			✓	
Charm++	APGAS	✓				✓
Swift	Dataflow					✓
Legion	Dataflow					✓
SMPSs	Dataflow					✓
Tarragon	Dataflow					✓

tions (both legacy and current) remain developed under the SPMD paradigm (MPI). Despite their success traditional SPMD applications will fall short in addressing communication demands.

Reducing network communication costs requires distributed applications to implement communication/computation overlap or communication avoiding mechanisms to reduce the cost of network latency. Although prior work has proposed ways to reduce the impact of either source of communication cost have been proposed, some questions remain unanswered:

1. How to combine both shared memory and latency-hiding mechanisms.
2. What is the combined effect of those mechanisms on large-scale performance?
3. How to provide these benefits via an intuitive, unified model.

The contribution of this work is to answer those questions by introducing *MATE*, a new model that integrates multiple mechanisms to tackle all sources of communication overhead simultaneously. We describe the MATE model's rationale in detail in the next chapter.

Chapter 3

The MATE Model

3.1 Overview

We introduce *MATE*[1], a new programming model that reduces the cost of communication of large-scale scientific applications both within and across node boundaries. MATE provides a simple interface that allows domain area scientists to develop new distributed applications that benefit from the communication-tolerance benefits of our model. Similarly, scientists can enhance an existing application to benefit from MATE's potential while requiring minimal intervention on its source code.

MATE integrates a combination of useful mechanisms that work synergistically. First, MATE provides *overdecomposition*, a well-known partitioning technique for hiding the latency of network communication by overlapping communication with computation. Second, it distributes the application's workload using a *hierarchical rank decomposition*, which enables the use of shared memory. This is a novelty of our model that helps reducing the additional intra-node data motion costs associated with overdecomposition. Third, it provides a light-weight synchronization mechanism, another novelty, that prevents serialization of local operations. This

[1]Named after *Mate* (pronounced '*mah-tay*'), a popular South American infusion.

mechanism is essential to maximizing concurrency while benefiting from shared memory use. Finally, it supports a code scheduling model that further partitions a program's execution to expose additional potential for communication/computation overlap.

These mechanisms are orthogonal to domain-specific communication-avoiding techniques that can be manually implemented on a given application. Indeed, a developer could apply MATE to a manually optimized application to further enhance its tolerance to communication costs.

In this chapter, we first introduce an example application upon which we will apply our model. Second, we explain the rationale behind MATE's mechanisms. Third, we describe the effects of each mechanism in the performance of our example application. Finally, we analyze prior work in communication-reducing models.

3.1.1 Motivating Example

Throughout this chapter, we use an example structured grid solver code that computes the solution to a partial differential question on an *n*-dimensional grid. This solver uses a Jacobi [68] finite difference method to iteratively compute a solution that satisfies a set of equations within a specified error margin. The solver uses a *stencil* operator to calculate the new value of every cells in the grid as the weighted function of neighboring cell values (Fig. 3.1).

The strategy for this solver is to divide the grid into smaller partitions that can be calculated simultaneously by multiple processes. Fig. 3.2 illustrates the problem decomposition using, for example, four processes to solve a 2D grid. In this case, the grid is decomposed equally into sub-grids of size $N/2 \times N/2$ that are distributed evenly among the processes. The solver approximates the solution iteratively in a bulk-synchronous fashion. At every iteration, each process applies the sequential solver to all the elements of its sub-grid and then communicates boundary cells to/from neighboring processes, required by the stencil in the next iteration.

Fig. 3.3 shows the MPI pseudo-code for the algorithm that solves Poisson's equation on

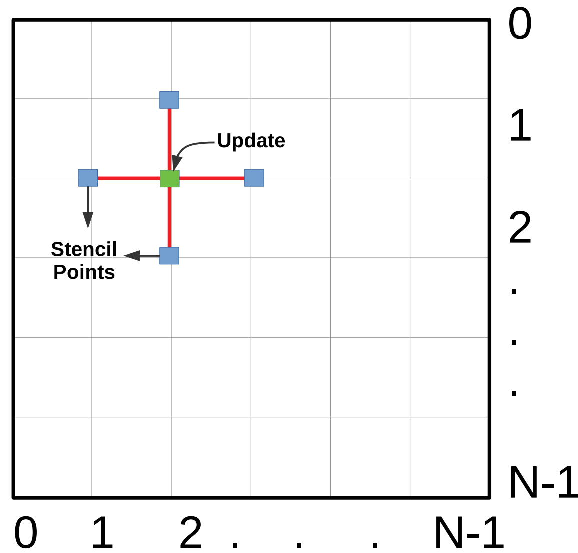

Figure 3.1: 5-Point stencil update on a 2D Grid.

a 2D grid using the Jacobi method. *N*, the number of elements per side of the global solution is provided as input. At the beginning of execution, each rank obtains its unique rank identifier and the total number of ranks (*lines 3-4*). This information is required to calculate the processor geometry. For simplicity, this example assumes a square processor geometry *P* that equals $\sqrt{nRanks}$, the linear dimension of the geometry (*line 6*). Each rank calculates its own (*x,y*) coordinates (*line 7-8*) to determine which subdomain it owns, and determines its neighboring ranks based on axial proximity.

Ranks create and initialize a list of immediate neighboring ranks (*Neighbors*) to exchange boundary data with (*line 10*). The number of neighboring ranks to insert into the *Neighbors* list depends on the placement of the rank (*lines 11-14*). If a rank's subdomain has no faces coinciding with the outer domain's boundary, then we add four neighbors (*left, right, up* and *down*). If one of the rank's faces coincides with a physical boundary, then the rank will have three neighbors, and so on for more faces.

After determining its coordinate and neighbors, each rank calculates the number of elements per side of its subgrid (*n*) by dividing the original grid size (*N*) by the number of processes

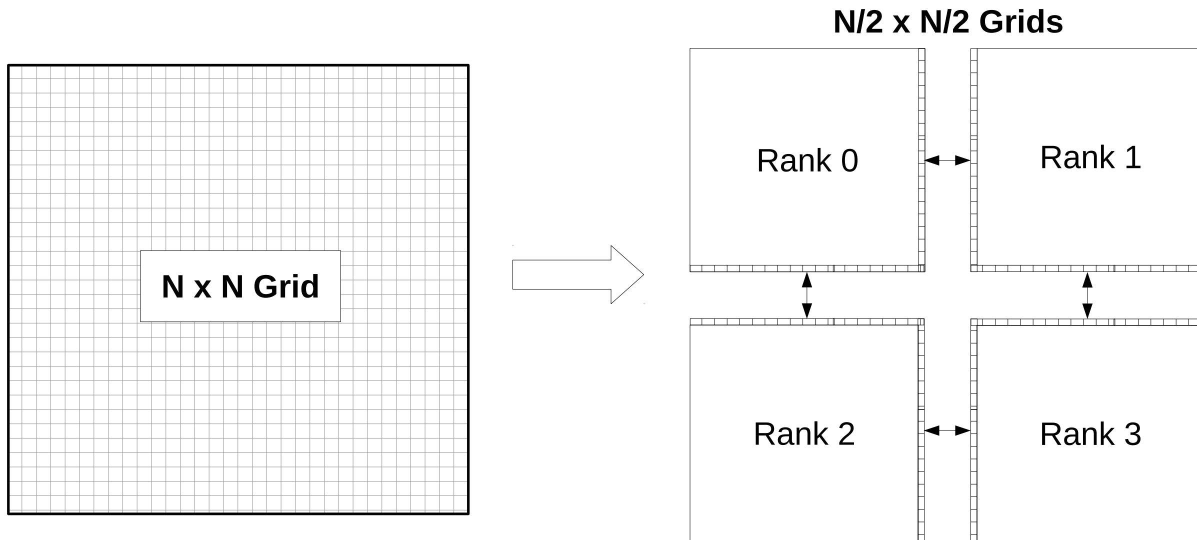

Figure 3.2: SPMD Decomposition of a 2D grid into 4 MPI Ranks.

per grid side (*P*) (*line 16*). Since this example uses a Jacobi solver, it requires the allocation of two arrays to store the state of their subgrid on the current and previous (*uNew* and *uOld*) loop iteration (*lines 17-18*). The algorithm initializes the values of its current subgrid (*line 19*).

The iterative solver part of the algorithm starts with the *for* loop (*line 21*). At the beginning of every iteration (*line 23*), the solver updates the its solution by sweeping the current subgrid (*uNew*) using the stencil and then swaps its pointer with that of the previous subgrid (*uOld*). After computing, the solver enters its communication phase and issues receive requests for boundary cells[2] (*line 24)*. We illustrate these operations using an incoming arrow from the neighbor *x* to the receive buffer corresponding to *x*. Ranks issue as many receive requests as neighbors in the *Neighbors* list.

Ranks pack their boundary cells into neighbor-specific buffers (*line 25*) and issue asynchronous send requests to transmit the buffered boundary data to their neighbors (*line 26*). After issuing these requests, ranks suspend, waiting for communication requests to finish (*line 27*).

[2]Receive requests are often placed before the computation part to inform MPI of the receive buffer address as soon as possible to avoid the need for intermediate message storage. However, we assume that the initial ghost cells have been set up during grid initialization. In doing so, we can put all communication operations together to simplify the discussion.

```
1  Solver(N)
2  {
3   MPI_Comm_size(MPI_COMM_WORLD, &nRanks);
4   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
5
6   P = sqrt(nRanks); // Assumes a square 2D decomposition
7   myRankX = myRank % P;
8   myRankY = myRank / P;
9
10  Neighbors = new List();
11  if(myRankX > 0) Neighbors.Add({myRankY, myRankX-1}); // Left
12  if(myRankX < P) Neighbors.Add({myRankY, myRankX+1}); // Right
13  if(myRankY > 0) Neighbors.Add({myRankY-1, myRankX}); // Up
14  if(myRankY < P) Neighbors.Add({myRankY+1, myRankX}); // Down
15
16  nP = N / P;
17  uNew = malloc(nP*nP);
18  uOld = malloc(nP*nP);
19  initialize(uNew, nP);
20
21  for(int i = 0; i < Iterations; i++)
22  {
23   ApplyStencil(nP, uNew, uOld); Swap(&uNew, &uOld);
24   for(x in Neighbors) MPI_Irecv(recvBuf(x)←x);
25   for(x in Neighbors) MPI_Pack(uNew→sendBuf(x));
26   for(x in Neighbors) MPI_Isend(sendBuf(x)→x);
27   MPI_Waitall();
28   for(x in Neighbors) MPI_Unpack(uNew←recvBuf(x));
29  }
30 }
```

Figure 3.3: MPI pseudo-code of a structured 2D grid stencil solver.

After all the request complete, ranks unpack the incoming neighboring boundary cells into ghost cell positions (*line 28*) required by the computation phase in the next iteration.

3.2 Communication-Reducing Mechanisms

The MATE model re-interprets an application via three mechanisms (1) a *domain overdecomposition*, (2) a *hierarchical rank decomposition*, (3) a *code scheduling* logic that exposes additional concurrency and provides efficient inter-rank synchronization. Together, these mechanisms enable the programmer to alter program semantics to reduce the cost of communication. In these section, we provide the rationale for these mechanisms.

3.2.1 Mechanism I: Domain Overdecomposition

Domain overdecomposition [50] or *virtualization* [52] is a technique for overlapping communication with communication operations that pipelines the execution of multiple ranks. An overdecomposed application splits its workload into a number of ranks larger than the number processor cores, preempting a rank when it is waiting for communication to permit another to execute in its place, maximizing core usage.

The main hurdle in enabling overdecomposition on traditional MPI libraries is that they delegate the responsibility of scheduling MPI processes to the operating system's kernel, which is unaware of message/data dependencies. Applied naively, overdecomposition will cause destructive interference of core usage. For this reason, the optimal configuration for MPI applications requires that the number of ranks (t) be not larger than to the number of cores (c) in the system[3]. Instantiating $t > c$ may cause MPI processes to destructively compete for a core, continually preempting one another from execution and causing cache/TLB thrashing.

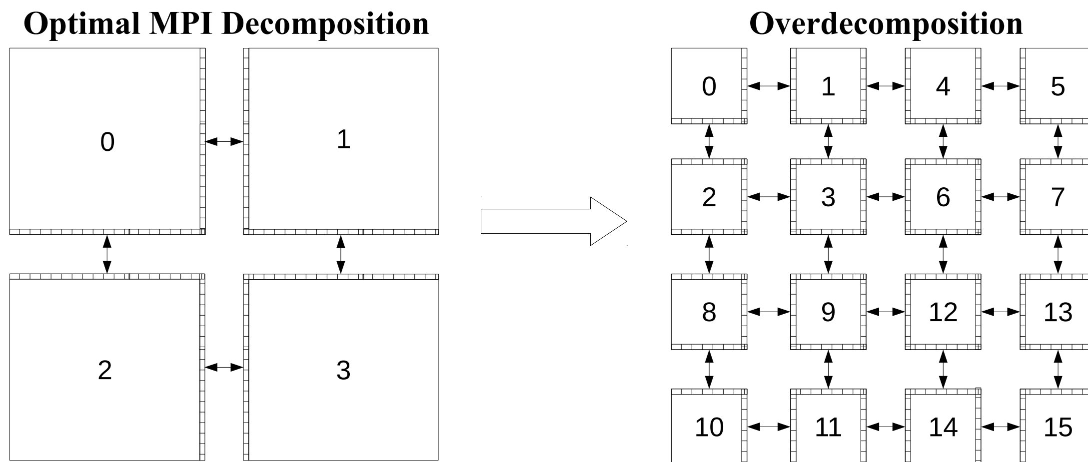

Figure 3.4: Traditional Decomposition vs Overdecomposed 2D grid.

Contrary to MPI, MATE schedules ranks entirely through its user-level runtime system (see Chapter 4). A user-level scheduler enables MATE to preempt a rank so that it can wait for

[3]Although exceptions exist where it is more convenient to use fewer MPI processes than available cores. For example, applications that change their workload during runtime may choose to operate with lesser cores to reduce the volume of communication. Also, MPI+X models may use fewer MPI ranks per node to enable multi-threading.

the arrival of data, allowing another rank to execute instead. This can be accomplshied without the intervention of the OS kernel. Consequently, MATE applications can instantiate more ranks than cores without destructive interference. Furthermore, the context switch overhead for user-level threads is typically much lower than that of OS processes, since it involves no system calls or interrupts (*e.g.,* INT 0*x*80 or CPU clock interrupts).

Fig. 3.4 compares the optimal decomposition of a traditional MPI application to an overdecomposed execution under MATE. The traditional MPI application will divide the grid into four subgrids, each processed by its MPI process, assuming four cores. On the other hand, a MATE application can divide the grid even further into, for example, 16 smaller subdomains.

Although effective in reducing the cost of inter-node communication, overdecomposition leads to an increase in intra-node data motion overhead due to a higher surface-to-volume ratio in exchange buffers. Indeed, Fig. 3.4 (*right*) shows that the internal faces of overdecomposed ranks require additional boundary exchanges. As a consequence of the increase in boundary surface, ranks need to perform additional buffer packing and transferring overheads. This effect can significantly offset the benefits obtained from overlap. In the next section, we explain how the MATE model deals with this problem using a hierarchical rank distribution.

3.2.2 Mechanism II: Hierarchical Rank Distribution

One way to reduce the cost of intra-node data motion is to enable ranks located in the same node to communicate via shared memory. Co-located ranks can either exchange their boundary information directly without the need for packing/unpacking or messaging operations or, better yet, operate on the same subdomain so that their boundary information is directly accessible by their neighboring ranks.

MATE supports shared memory by exposing a 2-level memory hierarchy-aware rank decomposition. At the top level, a MATE application distributes the workload onto a set of processes. A *MATE Process* represents a logical entity that contains its own *virtual address*

space. At the second level, each process is further distributed among multiple local *MATE Ranks*. Unlike overdecomposed MPI ranks, local MATE ranks assigned to the same MATE process live in the same virtual address space.

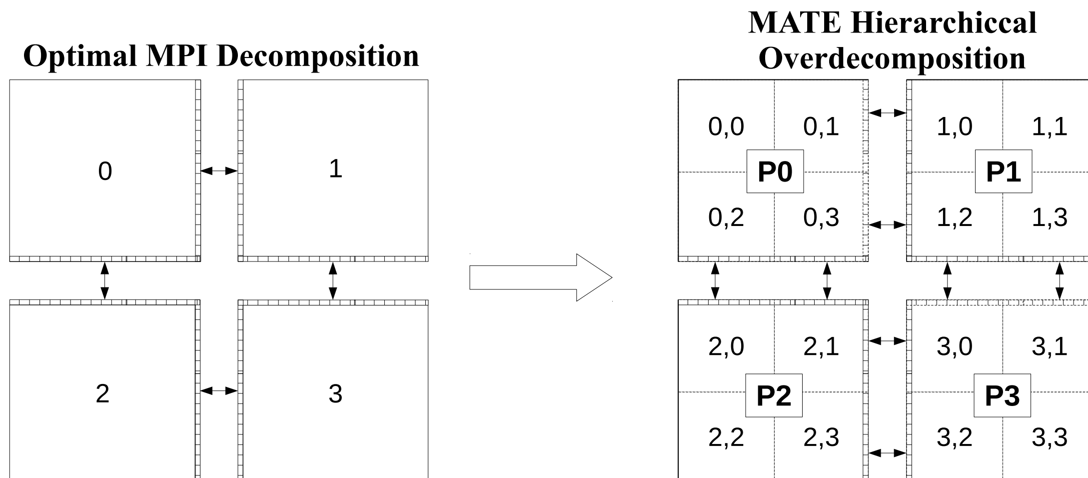

Figure 3.5: Traditional Decomposition vs MATE Hierarchical overdecomposition.

Fig. 3.5 shows how local ranks belonging to the same process can work collaboratively on the same process-wide grid. The process allocates its subgrid as a single segment of memory and subdivides it logically (dotted line) among process-local ranks. Since local MATE ranks work in the same address space, their updates to local boundary cells will be immediately visible by its local neighbors, without the need of messaging or explicit data movement. In contrast, ranks transmit boundaries that cross the process subdomain (solid line) to remote ranks via message passing (arrows). This approach keeps the volume of boundary data to be transmitted across nodes fixed, even when using overdecomposition.

To implement a hierarchical decomposition, a programmer needs to add calls to the MATE API (see appendix E) to the source code. The first step is to obtain each MATE rank's *process-level* and *local-level* identifier pair. The purpose is to identify a MATE rank by combining its parent MATE process identifier, and its unique local identifier within the process. Fig. 3.5 shows ranks can query (*line 3*) their local id, their containing process id (*line 4*), the number of local ranks (*line 5*), and the number of MATE processes (*line 6*).

To determine each rank's part of the subgrid, the solver determines the number of MATE processes per side P (*line 8*), and the process' x and y coordinates (*lines 9 and 10*). Subsequently, the solver calculates the number of local ranks per side within the process L (*line 12*) and each rank's local x and y coordinates (*lines 13 and 14*) within the process domain.

```
1  Solver(N)
2  {
3   Mate_local_rank_id(&myLocalId);
4   Mate_global_process_id(&myProcessId);
5   Mate_local_rank_count(&localCount);
6   Mate_global_process_count(&processCount);
7
8   int P = sqrt(processCount); // Assumes a square 2D global decomposition
9   gRankX = myProcessId % P; // Global Rank X
10  gRankY = myProcessId / P; // Global Rank Y
11
12  int L = sqrt(localCount); // Assumes a square 2D local decomposition
13  lRankX = myLocalId % L; // Local Rank X
14  lRankY = myLocalId / l; // Local Rank Y
15
16  Neighbors = new List();
17  if(gRankX > 0 && lRankX == 0)   Neighbors.Add({gRankY, gRankX-1}, {lRankY, L-1}); // Left
18  if(gRankX < P && lRankX == L-1) Neighbors.Add({gRankY, gRankX+1}, {lRankY, 0  }); // Right
19  if(gRankY > 0 && lRankY == 0)   Neighbors.Add({gRankY-1, gRankX}, {L-1, lRankX}); // Up
20  if(gRankY < P && lRankY == L-1) Neighbors.Add({gRankY+1, gRankX}, {0,   lRankX}); // Down
21
22  int nP = N / P;
23  if (myLocalId == 0) uNewProcess = malloc(nP*nP);
24  if (myLocalId == 0) uOldProcess = malloc(nP*nP);
25  Mate_LocalBcast(&uNewProcess, 0);
26  Mate_LocalBcast(&uOldProcess, 0);
27
28  int nL = nP / L;
29  uNew = &uNewProcess[lRankY*nL*nL + lRankX*nL];
30  uOld = &uOldProcess[lRankY*nL*nL + lRankX*nL];
31  initialize(uNew, nL);
32
33  for (int i = 0; i < Iterations; i++)
34  {
35   ApplyStencil(nL, uNew, uOld); Swap(&uNew, &uOld);
36   Mate_LocalBarrier(); // Required to prevent RAW Hazards
37
38   for (x in Neighbors) MPI_Irecv(recvBuf(x)←x);
39   for (x in Neighbors) MPI_Pack(uNew→sendBuf(x));
40   for (x in Neighbors) MPI_Isend(sendBuf(x)→x);
41   MPI_Waitall();
42   for (x in Neighbors) MPI_Unpack(uNew←recvBuf(x));
43  }
44 }
```

Figure 3.6: Pseudocode of the MATE hierarchically decomposed grid stencil solver.

MATE ranks exchange ghost cells explicitly if, and only if, two conditions apply: (1) the ranks are not local to the same process, and (2) the boundary does not coincide with the

physical boundary. If condition (1) fails, then the rank neighbors to a local rank and can access boundary information via shared memory. If condition (2) fails, then there are no ghost cells to communicate since the boundary is part of the physical boundary. Only if both conditions apply, then the neighboring rank belongs in a different MATE process and thus requires message passing communication. In this case, the rank identifies its remote neighbors (*i.e.*, neighbor ranks that live in a different MATE process) by their process/local id pair and registers them into its list of neighbors (*lines 16-19*).

At this point, we have not changed the solver code, except by adding a process-wide barrier to synchronize all the local ranks just after they finish computing (*line 36*). This synchronization is necessary to prevent neighboring ranks from advancing to the next iteration prematurely, accessing old boundary values and exposing read-after-write hazards.

The consequence of using a barrier is that all local ranks are forced to synchronize before they can communicate. This operation serializes computation and greatly reduces opportunities for overlapping communication with computation since it encompasses all the ranks/threads in the process, even those that are not neighbors nor mutually dependent (*e.g.*, ranks 0,1 and 0,2 in Fig. 3.5), forcing some ranks to wait even if all their boundary data is ready to be used. In the next section, we see how the MATE model provides a way to subdivide the execution of ranks and enable lightweight synchronization mechanism that avoids local rank serialization and maximizes parallelism.

3.2.3 Mechanism III: Code Region Scheduling

MATE supports a third level of decomposition that is orthogonal to the process-rank organization. This feature allows programmers split a program into a logical grouping of contiguous code statements that are scheduled independently, guided by a dependency graph. The effect is to subdivide the execution of a rank into smaller steps, exposing further opportunities for overlapping communication with communication, via improved pipelining. This principle is similar

to that used in *instruction-level-parallel* (ILP) processors [89], in which instructions break into smaller sub-instructions with a subset of dependencies that expose further parallelism.

```
1  int main(int argc, char** argv)
2  {                                          //    --------------
3    printf("a");                             //   |  Sequential   |
4    printf("b");                             //   |  Block        |
5                                             //    --------------
6    #pragma mate graph
7    {                                        //    --------
8      #pragma mate region(D) depends(C)      //   |        |
9        printf("d");                         //   |        |
10                                            //   |  MATE  |
11     #pragma mate region(E) depends(C)      //   |        |
12       printf("e");                         //   |  Graph |
13                                            //   |        |
14     #pragma mate region(C)                 //   |  Block |
15       printf("c");                         //   |        |
16   }                                        //    --------
17
18                                            //    -------------
19   printf("f");                             //   | Sequential  |
20   printf("g");                             //   | Block       |
21 }                                          //    -------------
```

Figure 3.7: Simple example of a MATE-annotated program.

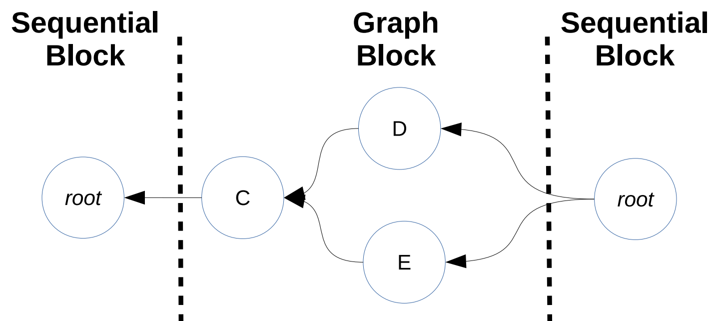

Figure 3.8: Dependency graph of the MATE-annotated *for* loop in Fig. 3.8.

To take advantage of this feature, programmers enhance the source program with MATE annotations. Fig. 3.7 shows a simple example of a valid MATE-annotated program. A programmer defines MATE dependency graphs by enclosing a part of the program within a block preceded by a *#pragma mate graph* directive (*line 6*). Within the dependency graph, the programmer defines *MATE regions*, sections of code that MATE can schedule individually.

A programmer defines regions statically, by annotating the source code with the *#pragma*

mate region(name) directive (*lines 8, 11, 14*), where *name* serves as a user-defined region identifier. The statements enclosed by a *region* directive execute in-order but will not be scheduled until the regions identified inside the *depends* clause have themselves finished.

Programmers define the dependency graph by appending a *depends(region1, region2,...)* clause to each *region* directive. Every region name included in the *depends* clause will create a new region-to-region dependency. Fig. 3.8 shows the dependency graph generated by the code in Fig. 3.7. Upon reaching the graph block, the only region that can execute (has no dependencies) in this example is '*C*'. After '*C*' finishes, the two other regions will have satisfied all their dependencies, and thus both could execute in any order. The outcome of this program is, therefore, nondeterministic[4] and may print either: "*abcdefg*", or "*abcedfg*".

MATE supports incremental development by allowing a programmer to annotate only parts of the application. That is, a MATE application may contain blocks of code that execute sequentially. To execute un-annotated code, MATE implicitly introduces a *root* region that represents the whole program outside user-defined regions. Upon finding a *#pragma mate graph* directive, the *root* region yields execution until all regions inside the graph finish, and then resumes execution. The MATE model ensures that the generated code is semantically equivalent to the source code (it is possible that some arithmetic may be re-ordered), as long as the region dependencies introduced by the programmer do not violate the original program's semantics.

As in traditional dataflow [36, 8, 25] models, the flow of data in MATE's dependency graph determines in part the order in which regions execute. To ensure the availability of data, MATE guarantees that all communication requests (*e.g. MPI_Irecv*, *MPI_Isend*) that a region issues during execution will have completed before dependent regions can execute. For this reason, regions within a graph block require no explicit calls to waiting operations and MATE will remove any calls to *MPI_Wait() MPI_Waitall()*. However, code outside a MATE graph block will suspend upon finding such blocking operations. This is to guarantee that normal MPI

[4]By its implementation, MATE resolves nondeterminism by executing regions in the order that reside in the code. However, this is not part of the model and the programmer should assume no particular ordering.

semantics remain unaltered in non-annotated sections of the program.

```
1  #pragma mate graph
2  {
3    #pragma mate region (A)
4    {
5      MPI_Irecv(data_x);
6      p = use(data_x);   // Incorrect: data_x has not arrived yet.
7    }
8
9      #pragma mate region (B) depends (A)
10     printf(p);         // Will print an incorrect result
11 }
```

```
1  #pragma mate graph
2  {
3    #pragma mate region (A)
4      MPI_Irecv(data_x);
5
6    #pragma mate region (B) depends (A)
7    {
8      p = use(data_x);   // Correct: data_x has arrived
9      printf(p);         // Will print a correct result
10   }
11 }
```

Figure 3.9: Pseudo-code example of (top) an incorrect description, and (bottom) a correct description of a MATE dependency graph.

Because MATE enables the programmer to reorder code blocks, the programmer is responsible for ensuring that their annotations preserve correctness. Fig. 3.9 (*top*) represents an example of an incorrect dependency graph. This code declares two regions, *A* (*line 3*) and *B* (*line 9*. Region *A* contains a request for incoming data to be stored in the *data_x* buffer (*line 5*), and a call to *use()* that accesses the buffer to produce a result *p* (*line 6*). The result is that, since *use* will execute immediately after issuing the *receive* request, possibly accessing the buffer before data has arrived. Region *B* will, therefore, print a possibly incorrect value of *p* (*line 10*).

Fig. 3.9 (*bottom*), represents correct description of the example code's dependency graph. In this case, region *A* only produces a receive request for *data_x*, while *B* uses *data_x* and prints the value of *p*. Since MATE will not execute region *B* until the request generated during the execution of *A* has finished, the *use* function will always access relevant data and produce a correct result.

Inter-rank Dependencies

Because neighboring local ranks work on a mesh in a single address space, classic shared memory correctness issues arise that were not present in the original MPI program. MPI couples explicit data motion and synchronization through reciprocal *send/recv* requests, avoiding the issue. As mentioned previously, the use of process-wide barriers prevents ranks from advancing independently, reducing the potential for communication-computation overlap. To address this issue, MATE introduces provides support for *inter-rank dependencies*.

Inter-rank dependencies are lightweight synchronization mechanisms that allows programmers to specify dependencies between regions among local ranks. These dependencies are essential to maximize performance in applications programmed under the MATE model since they guarantee correctness while preserving concurrency exposed by the dependency graph.

```
1  int main(int argc, char** argv)
2  {
3    if (myLocalId > 0) Mate_AddLocalNeighbor(myLocalId-1);
4
5    #pragma mate graph
6    {
7      #pragma mate region(A) depends(A@)
8        print(myLocalId);
9    }
10 }
```

Figure 3.10: Example use of MATE's inter-rank dependencies.

Fig. 3.10 shows a simple example of a MATE application that uses inter-rank dependencies. Programmers define inter-rank dependencies by (1) calling *Mate_AddLocalNeighbor()*, which indicates to the runtime system which local ranks represent neighbors that share data with the current rank, and (2) then appending the '@' modifier to any region in the *depends* clause (*line 7*). This application declares a single region *A* that prints the local identifier of the rank, and depends on the execution of same region (*A*) from its neighbor ranks. All local ranks (except local rank 0) declare a single neighbor representing the local rank with a preceding identifier (*line 3*). The output of this application is the ordered ids of all the local ranks ("*0 1 2 3 4...*").

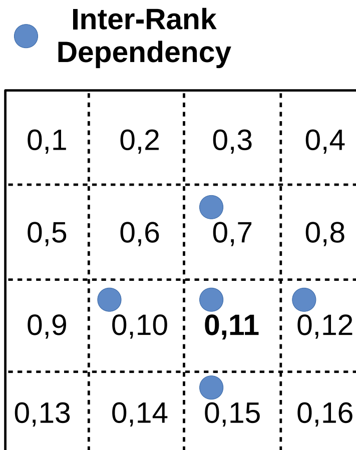

Figure 3.11: Inter-rank dependencies for rank (*0,11*).

Fig. 3.11 shows the local neighbors for rank (*0,11*) inside a MATE process that uses a 4x4 local rank distribution for our example 2D solver with stencil points on 1-deep manhattan directions, where ranks specify their local *left/right/up/down* neighbors.

For Loop Graphs

Since iterative solvers are an important target of the MATE model, we enable programmers to expose additional concurrency from *for* loops through *Previous-iteration dependencies*. Programmers specify these dependencies appending the '*' modifier to regions in the *depends* clause. This modifier tells the scheduler that the dependency will be satisfied if the region has executed in the previous iteration[5]. As an exception to avoid deadlocks, MATE ignores dependencies with the '*' modifier in the first iteration. Wherever the '*' is not specified, then the dependency refers to the current iteration.

Our model allows the combined use of the '*' and '@' modifiers in the *depends* clause. This notation will define a previous-iteration inter-rank dependency, instructing the runtime system to only schedule the current region after all the neighboring local ranks have finished exe-

[5]Although our model could support dependencies than span across more than one iteration, we have not yet found an example that requires this flexibility. Algorithms that employ wave-front parallelism [65], however, may benefit from this mechanism. This case would require us to re-formulate our syntax to allow constant (*e.g*, *3) or variable (*e.g*, *n) numerical values.

cuting the depended region on the previous iteration.

```
1  int main(int argc, char** argv)
2  {                                                      //    --------------
3    printf("a");                                         //   |              |
4    printf("b");                                         //   |  Sequential  |
5                                                         //   |  Block       |
6    Mate_AddLocalNeighbor(myLocalId+1);                  //   |              |
7    Mate_AddLocalNeighbor(myLocalId-1);                  //   |              |
8                                                         //    --------------
9    #pragma mate graph
10   for (int i = 0; i < n; i++)
11   {                                                    //    --------
12     #pragma mate region(D) depends(C)                  //   |        |
13       printf("d");                                     //   |  MATE  |
14                                                        //   |        |
15     #pragma mate region(E) depends(C, D@)              //   |  For   |
16       printf("e");                                     //   |  Loop  |
17                                                        //   |  Graph |
18     #pragma mate region(C) depends (D*, C*@)           //   |  Block |
19       printf("c");                                     //   |        |
20   }                                                    //    --------
21
22                                                        //    -------------
23   printf("f");                                         //   | Sequential  |
24   printf("g");                                         //   | Block       |
25 }                                                      //    -------------
```

Figure 3.12: Example of a MATE-annotated *for* loop.

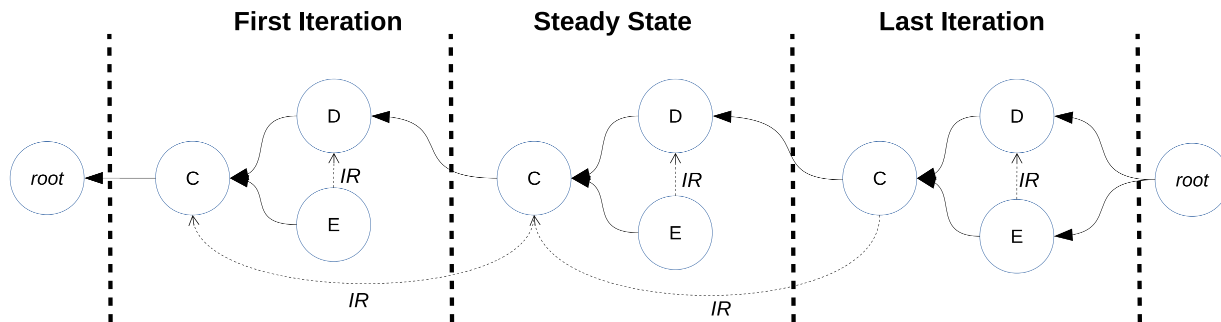

Figure 3.13: Dependency graph of the MATE-annotated *for* loop in Fig. 3.12.
IR = Inter-rank dependency.

Fig. 3.12 shows an example of a valid MATE program with an annotated *for* loop. Fig. 3.13 shows the dependency graph generated by the MATE annotations in Fig. 3.12. In this example, each rank first registers its nearest neighbor ranks as local neighbors (*lines 6-7*). The *graph* pragma before the *for* loop (*line 9*) instructs MATE to interpret the loop as an iterative graph block. In this case, region *E* (*line 15*) not only depends on region *C* (*line 18*), but also

on the execution of region *D* (*line 12*) of its neighbor local ranks. Region *C* depends on the execution of its region *D* from the previous iteration, and its neighbor local ranks' region *C* from the previous iteration.

The execution of this generated code will repeatedly produce the same output as the code in Fig. 3.7, with the following added constraints: (1) ranks cannot print "C" before they print "D" in the previous iteration (due to the use of the * modifier), (2) ranks cannot print "E" before its neighbors have printed "D" during the same iteration (due to the use of the @ modifier), and (3) no ranks can print "C" until all its neighbor ranks have printed "C" in the *previous* iteration (due to the combined use of the *@ modifiers).

Semantic Considerations

One semantic limitation with a MATE-annotated *for* loop is that its induction variable should not be modified or accessed within a region's body since these changes invisible to other regions. This limitation is a consequence of the way MATE translates the source to allow regions to advance independently from the others in the *for* loop. For example, the code in Fig. 3.14 will exhibit unpredictable behavior since it modifies the iterator (*i*) variable (*line 5*). Furthermore, even read-only accesses (*line 8*) to the iterator in the region's body may also cause conflicts since regions may execute in different iterations at any given point. On the other hand, the code in Fig. 3.12 is acceptable because none of statements within its *for* loop modify or access the iteration variable.

```
1  #pragma mate graph
2  for (int i = 0; i < n;)
3  {
4    #pragma mate region(A) depends(B)
5      i++; // Not visible to other regions
6
7    #pragma mate region(B) depends(A*)
8      printf("%d", i); // Possible out-of-order conflict
9  }
```

Figure 3.14: Simple example of an invalid MATE-annotated *for* loop.

Structured Grid Example

Fig. 3.15 shows how MATE regions and dependencies can be used to expose the underlying dependency graph of the hierarchically decomposed variant of our structured grid solver from Fig. 3.6. This variant defines a single *graph* block (*line 39*) which spans the iterative part of the solver (*line 40-56*), where the application spends most of its running time. We leave the rest of the code un-annotated.

These annotations turn the *for* loop into a dependency-graph comprised of 5 MATE regions: the *compute* region (*line 42*) contains calls to *ApplyStencil* and *Swap* functions; the *receive* region (*line 45*) contains calls to *MPI_Irecv*; the *pack* region (*line 48*) contains calls to *MPI_Pack*; the *send* region (*line 51*) contains calls to *MPI_Isend*, and; the *unpack* region (*line 54*) contains calls to *MPI_Unpack*.

The semantics of this solver requires that ranks synchronize their computation phase with their neighbors in the grid with calls to *Mate_AddLocalNeighbor()* (*lines 34-37*). As mentioned before, Fig. 3.11 shows the local neighbors for rank (*0,11*) for a MATE process with 4x4 local ranks running this code.

Fig. 3.16 represents the dependency graph generated by the MATE annotated program in Fig. 3.15. We can see that, during the first iteration, both *receive* and *compute* regions can begin executing, and in no particular order, since no data hazards exist between them (computation modifies the grid, and receive requests modify message buffers). The *pack* region can start only after the *compute* region has finished producing new boundary data to be sent to its neighbors. The *send* region can issue *MPI_Isend* operations once the *pack* region has finished packing the buffers. Finally, the *unpack* region cannot unpack data from the receive buffers into the grid until (1) all the *MPI_Irecv* requests instantiated during the execution of *request* region have finished, and (2) the *compute* region has finished using the ghost cell data from the last iteration.

After the first *for* loop iteration (*i.e.,* upon reaching steady state), previous-iteration dependencies are activated. This means, for example, that the *compute* region will not execute until

```
32  ...
33
34  if(lRankX > 0) Mate_AddLocalNeighbor(myLocalId-1); // Local Left
35  if(lRankX < L) Mate_AddLocalNeighbor(myLocalId+1); // Local Right
36  if(lRankY > 0) Mate_AddLocalNeighbor(myLocalId-L); // Local Up
37  if(lRankY < L) Mate_AddLocalNeighbor(myLocalId+L); // Local Down
38
39  #pragma mate graph
40  for (int i = 0; i < Iterations; i++)
41  {
42   #pragma mate region(compute) depends(pack*, unpack*, compute*@)
43    ApplyStencil(nL, uNew, uOld); Swap(&uNew, &uOld);
44
45   #pragma mate region(receive) depends(unpack*)
46    for (x in Neighbors) MPI_Irecv(recvBuf(x)←x);
47
48   #pragma mate region(pack) depends(compute, send*)
49    for (x in Neighbors) MPI_Pack(uNew→sendBuf(x));
50
51   #pragma mate region(send) depends(pack)
52    for (x in Neighbors) MPI_Isend(sendBuf(x)→x);
53
54   #pragma mate region(unpack) depends(compute, receive)
55    for (x in Neighbors) MPI_Unpack(uNew←recvBuf(x));
56  }
57 }
```

Figure 3.15: Solver section of the code from 3.6, enhanced with a MATE dependency graph

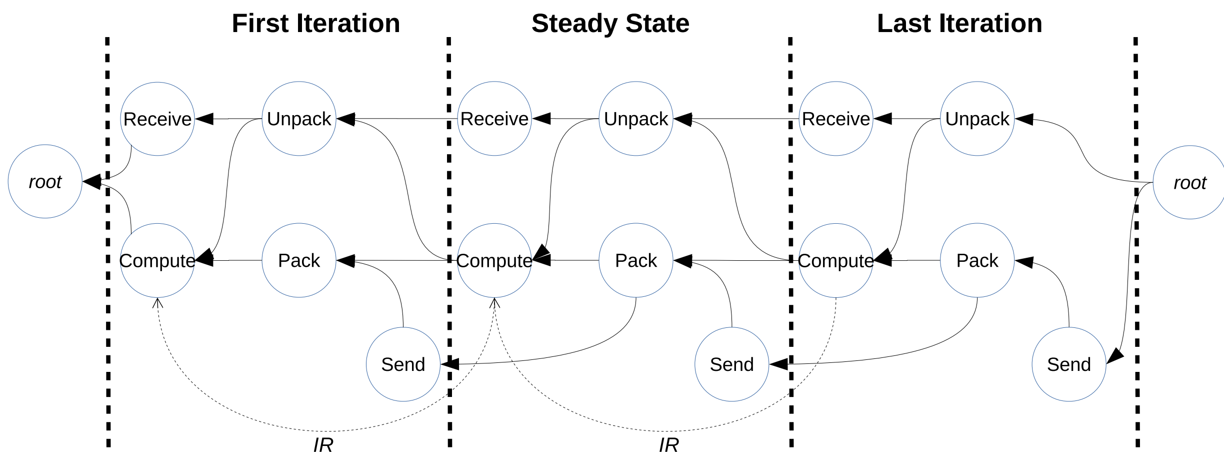

Figure 3.16: Dependency graph generated by the code in Fig. 3.15.
IR = Inter-rank dependency.

(1) the *unpack* region from the previous iteration has finished unpack ghost cell data into the grid, (2) the *pack* region has packed boundary data into the send buffers, and (3) local neighbor tasks have finished their *compute* region from the last iteration. The latter inter-rank dependency allows us to remove the *Mate_LocalBarrier* synchronization from Fig. 3.6, exposing all the potential parallelism in the solver. Finally, the *pack* region cannot overwrite the send buffers until the *MPI_Isend* operations from the *send* in the previous iteration signal that it is safe to reuse the buffers.

3.3 Communication Reducing Effects

To understand the effect of each of MATE's optimizations on core utilization, we employ a *core usage diagram*. We created these diagrams, shown in Fig. 3.17, to illustrate a possible timeline of a single core performing four iterations of the structured grid example. Solid blocks (containing the rank number) indicate segments of time in which the core is performing useful computation, unnumbered (hatched) blocks indicate intra-node data motion, while the dotted line indicates that the core is idle, waiting for data to arrive. Additionally, this diagram shows whenever a rank initiates inter-node communication requests (upwards arrow) and how long they take to finish (dotted line blocks with the word *Network*).

Fig. 3.17 *(A)* shows the execution of the original MPI structured grid solver described in Fig. 3.3 that does not attempt to overlap communication and computation. This variant suffers from two communication costs. First, since it operates in separate phases, it does not overlap communication and computation and thus suffers from the full cost of network communication. Second, since MPI processes living in the same node communicate through message passing, they incur additional overheads from copying data between ranks in the same node. This timeline shows that the core efficiency can potentially be very low given these costs.

Fig. 3.17 *(B)* shows the effect of executing the same code using overdecomposition.

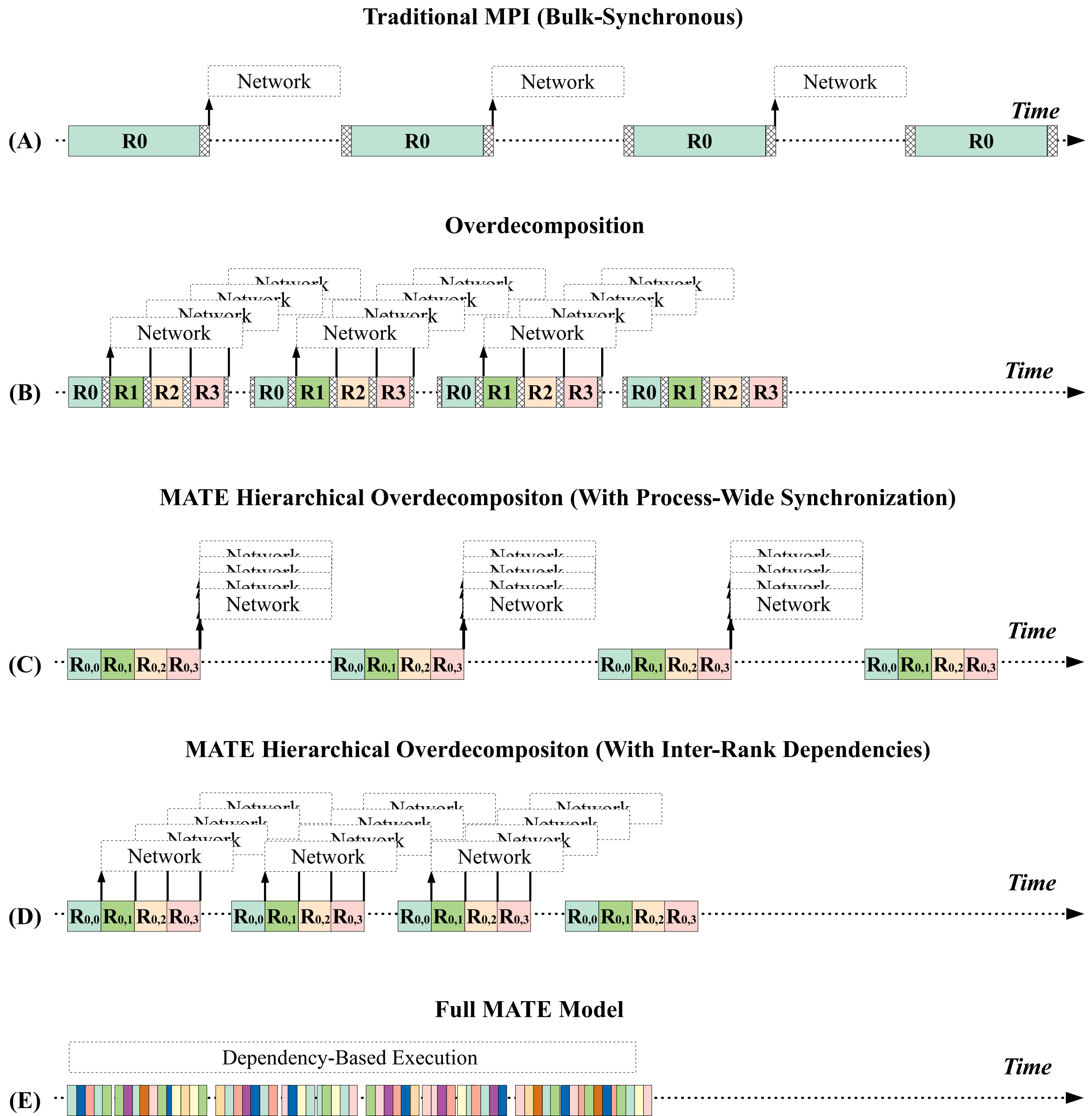

Figure 3.17: Hypothetical core usage timelines.

This case instantiates four ranks per original rank in *(A)*, and assigns them to a specific core[6]. In this case, whenever a rank (*e.g.* R0) suspends to wait on network communication, another (*e.g.* R1) can be scheduled for execution. Having multiple ranks executing enables communication/computation overlap, hiding most of the inter-node communication cost. However, since overdecomposed ranks need to exchange additional internal boundaries, the cost of intra-node communication increases compared to the original application.

Fig. 3.17 *(C)* shows the effect of using overdecomposition in a hierarchically decomposed application while still using a process-wide synchronization mechanism. In this case, since all overdecomposed ranks need to wait until the others finish computing before they can issue communication, they cannot achieve any communication/computation overlap. Therefore, this variant yields a similar execution as the *(A)* timeline, minus the overheads from data motion.

Fig. 3.17 *(D)* shows the effect of inter-rank dependencies on a hierarchically overdecomposed execution. In contrast to *(C)*, this variant does not serialize computation, and so it can benefit from the combined benefits of communication/computation overlap and reduced intra-node data motion.

Finally, Fig. 3.17 *(E)* shows the effect of applying the full MATE model. This variant benefits from the combination of mechanisms employed in *(D)*. In addition, the execution of each rank is split into smaller regions that can be scheduled independently by the MATE runtime system. Using the full scope of MATE's mechanisms exposes more concurrency that can be extracted by the underlying dependency graph and maximizes the opportunities for overlap. Furthermore, this figure shows that the MATE model enables cores to execute regions from any rank in the process (hence the additional colors), yielding higher core usage.

[6]In reality, a MATE process can migrate among different cores to improve overlap. See Chapter 4.

3.4 Related Work

Several programming models and libraries have been proposed to cope with the overheads of communication in large-scale scientific applications. However, none of them propose a way to integrate all the mechanisms described in this chapter into a single unified model. Their experiences and limitations have, nevertheless, influenced our work in the MATE model.

3.4.1 MPI+KLT Model

As mentioned earlier, the MPI + Kernel Level Thread (MPI+KLT) model provides a way to reduce intra-node data motion while enabling a distributed execution, by combining a communication model with a threading (shared memory) model. However, we have also seen that the two interfaces do not compose well as they are unaware of each other. MATE eliminates these complications by providing a single, unified interface to developing a distributed application that generates a hybrid execution model that is similar to MPI+KLT. Furthermore, MATE provides support for overdecomposition and light-weight synchronization. Integrating these mechanisms manually into an MPI+X application would entail a painstaking effort.

3.4.2 MPI+MPI Model

Another hybrid approach, *MPI+MPI* [48], proposes the use of MPI *windows* to enable shared memory communication within a node. This model preserves portability since only the MPI interface is required. An MPI+MPI application instantiates as many MPI processes as cores as the traditional model. However, while the application uses message passing for inter-node communication, processes within the same node communicate through the locally-mapped shared memory windows, as illustrated in Fig. 3.18.

Although MPI+MPI succeeds in reducing intra-node data motion by avoiding the need for messaging, it has limitations. First, it does not provide any mechanisms for overlapping com-

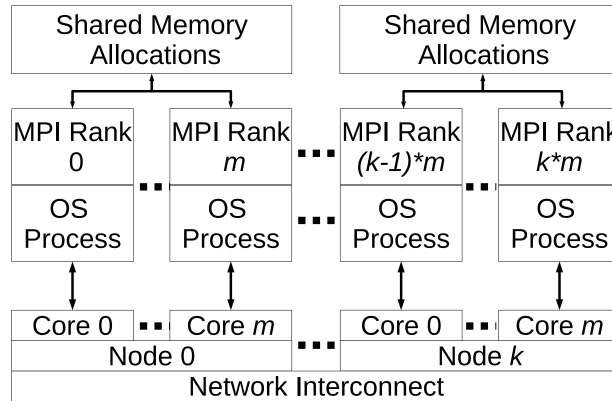

Figure 3.18: Example deployment of a hybrid MPI+SHM application where node co-located MPI processes can communicate through shared memory.

munication with computation. Second, MPI processes have non-symmetric heaps that do not guarantee that the shared memory segment will map to the same virtual address in every other process. Therefore, programmers need to derive a global pointer or index+offset logic to reference memory in another process. As a consequence, this solution entails a higher complexity than MPI+KLT approaches.

3.4.3 MPI+ULT Model

The *MPI + User Level Threads* (MPI+ULT) [63] model, also called *Fine-Grained MPI* [54] has been proposed to enable overdecomposition in MPI applications without destructive interference. This model instantiates MPI ranks as *user-level threads* (ULT), re-entrant functions that can be suspended and resumed at any point of their execution. Since ULTs are created and scheduled by the underlying library, a single OS process can contain more than one ULT. The execution of ULTs inside a process is concurrent, yet not parallel. An MPI+ULT process, assigned to a single core, will execute only one ULT for execution at a time, while other ULTs remain suspended.

Since multiple ULTs can live in a process without destructive interference, the MPI+ULT model relaxes the limitation on the granularity of traditional MPI applications, and therefore

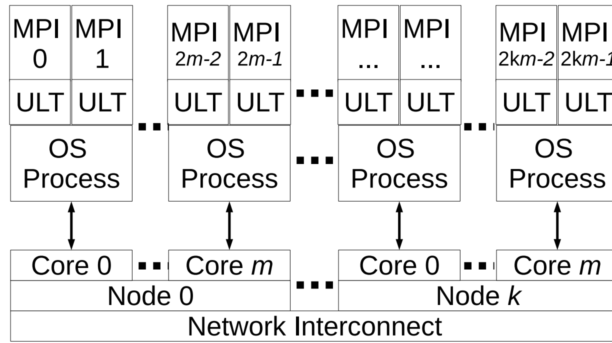

Figure 3.19: Example deployment of a hybrid MPI+ULT application where each MPI process is assigned a single core, but spans multiple MPI ranks, implemented as ULTs.

provides support for efficient overdecomposition. Fig. 3.19 shows an application that instantiates two ULTs per process, executing a total of $2 \times k \times m$ MPI ranks.

AMPI [49] also supports overdecomposition through *virtualization*. AMPI is an implementation of MPI that supports dynamic load balancing and fault tolerance for MPI applications. An AMPI-virtualized rank runs as a stream of Charm++ objects (chares) that execute, perform communication, and suspend, guided by the MPI communication and synchronization requests in the source code.

Although MPI+ULT models and AMPI enable an efficient overdecomposition of MPI ranks in the same way as MATE, they do not implement any mechanisms to reduce the cost of intra-node data motion. As a consequence, enabling overdecomposition through these models will cause the application to suffer from additional overheads that may offset the benefits obtained from overlapping communication and computation. Indeed, our experiments with AMPI and overdecomposition-only approaches revealed that, although they can succeed in hiding the cost of network communication, they are unable to obtain any significant speedups. MATE, on the other hand, implements a hierarchical overdecomposition strategy that reduces both sources of communication overhead, obtaining a net performance gain.

3.4.4 Latency-Hiding Models

Latency-Hiding programming models define the semantics of a program as a collection of statements (or groups thereof) that are scheduled in part by data-dependencies with the purpose of hiding the cost of network communication costs. Like MATE, these models schedule sections of a program (either a function or a region of code) out-of-order, based on data dependencies. These sections execute non-preemptively (atomically), and the statements contained therein execute in-order.

Latency-Hiding models interpret and analyze the content of code regions, as well as programmer-introduced annotations, to rearrange program execution, either statically (at compilation time) or dynamically (at runtime), to execute out-of-order based on their data dependencies. The intended effect is to maximize computation/communication overlap, thus reducing the impact of network communication latency.

MPI/SMPSs [66] integrates the MPI model with SMPss. MATE provides a similar interface to MPI/SMPSs: in both models, programmers use annotations to build a valid MPI program that executes based on data/execution dependencies. However, MATE provides three main distinctive improvements: (1) because it employs source-to-source translation, MATE can break individual functions into code regions without the need for programmers to split and outline code into functions manually, as is the case in SMPSs (2) unlike MATE, MPI/SMPSs provides no integrated support for overdecomposition, delegating the burden of implementing this mechanism to the programmer, and (3) MPI/SMPSs provides no mechanisms for a hierarchical decomposition or use of shared memory to reduce the cost of intra-node data motion.

Bamboo [71] is a source-to-source translator that reinterprets C/C++ MPI applications to execute as a dataflow program. In Bamboo, the description of the dependency graph is *implicit* [70]. Bamboo provides a fixed set of region names, with fixed (implicit) rules for expressing dependencies. There are four region names: *(i) Overlap* regions indicate the sections of the code that contain communication, *(ii) Send/Receive* regions enclose MPI send/receive operations,

respectively, and *(iii) Compute* regions enclose the section of computation that depends on the data arriving from the requests in the receive region, while producing data for the send region.

To enable communication/computation overlap, Bamboo performs a static analysis of programmer-introduced annotations that describe regions of code and the MPI calls therein to generate a data dependency graph. Bamboo uses this graph to perform transformations in the source code that enable regions to execute based on their dependencies. The output code produced by Bamboo targets the *Tarragon* task-driven runtime [25], where each MPI rank executes a set of Tarragon tasks. Tarragon provides a user-level runtime system that allows Bamboo to execute and schedule ranks independently from the operating system scheduler, enabling efficient overdecomposition.

One of the main limitations of Bamboo is, however, that its translation process requires extensive function inlining and static code relocation, which can bloat (in some cases up to 10x the number of lines) the translated code, making it harder to debug and requiring extended translation times. The rationale behind these transformations is that, in order to convert regions of C/C++ code containing MPI code into a dependency graph that can be executed by the Tarragon, functions need to be broken down and replicated (inlined) up to the main scope of the program. For this reason, Bamboo is unable to translate recursive code, since function inlining would generate an infinite loop. Finally, all communication in Bamboo requires annotation, which does not allow an incremental adaptation of an existing application to its latency-hiding form.

Our previous project, the *Toucan* translator [67], overcomes some of Bamboo's limitations. Toucan avoids static code transformations by employing a co-designed runtime system/-translator team that encapsulate most of the task-scheduling. The runtime system keeps track of code regions and their dependencies and automatically resolves when a region is ready to execute based on the arrival of MPI messages. Toucan simplifies the translation of MPI applications by inserting a modest number of calls to its runtime system's API, dynamically creating the dependency graph during execution, as opposed to restructuring the code statically. This

improvement allows Toucan to translate recursive code and enable an incremental development of Toucan applications.

One of the key contributions of the Toucan is showing the possibility of decomposing the execution of a rank into code regions without the need of outlining code into separate functions during translation. The Toucan translator transforms dependency graphs into *switch* statements enclosed in a *while* loop. Inside the *switch* statement, each region, as annotated in the source code, represents a different case. The *while* loop will run until all regions have finished while the *switch* statement will choose which *case* (region) to execute, based on a call to the runtime system's scheduler. We adopted these contributions into MATE as a successor to Toucan.

MATE introduces three significant improvements over Toucan. First, it generalizes Toucan's dependence model. Whereas Toucan provides a fixed set of region types, MATE admits user-defined region types. Second, it adds a the hierarchical locality-based decomposition model, which substantially enhances the benefit of overdecomposition. Third, MATE adds inter-rank dependencies, which enable efficient local synchronization without serializing effects.

Table 3.1: Comparison of the distributed programming models analyzed in this chapter.

	Mechanisms				
Model	Shared Memory	Over Decomposition	Dependency Graph	Graph Scheduling	Inter-Rank Dependencies
MPI+KLT	✓				
MPI+MPI	✓				
MPI+ULT		✓			
AMPI		✓			
MPI/SMPSs			✓	Dynamic	
Bamboo		✓	✓	Static	
Toucan		✓	✓	Dynamic	
MATE	✓	✓	✓	Dynamic	✓

3.5 Summary

We introduced MATE, a new programming model to develop communication-tolerant applications. The novelty in our model is that it integrates multiple mechanisms to reduce the costs of intra-node and network communication simultaneously, providing a single unified interface.

In this chapter we examined MATE's syntax and rationale in applying it to a common structured grid solver motif. We analyzed the effects of each mechanism on the performance of the solver, and how they produce a synergistic effect that maximizes the potential for communication/computation overlap.

Finally, we have seen that although prior approaches and programming models for communication tolerance provide subsets of MATE's functionality (except for inter-rank dependencies which is a novelty of our model), none of them target both the overheads of intra-node data motion and network communication taken together. Table 3.1 compares the features of these models with those provided by the MATE model.

Acknowledgements

This chapter is, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

Chapter 4

Design and Implementation

4.1 Overview

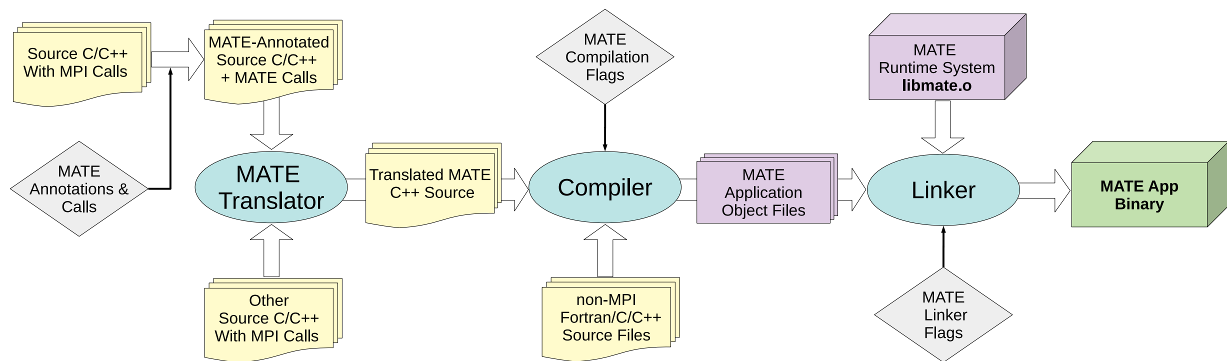

Figure 4.1: Annotation and compilation flowchart of a MATE application.

We have developed a programming framework comprised of a source-to-source translator and a runtime system to aid programmers in developing applications under the MATE model. We co-designed the components of our framework to minimize the effort required from programmers to either enhance an existing MPI program or develop a new MATE application.

Fig. 4.1 shows the steps required in building a MATE application, starting from a valid C/C++ MPI program[1]. The first step is to augment the application's source files with MATE

[1]Although we only currently support C++ code, our model could also support other languages (*e.g.*, Fortran).

annotations[2], as explained in Chapter 3.

The next step is to translate the MATE-annotated MPI program using MATE's translator. The translator will interpret annotations in the source code to produce a C++ program that contains a handful of calls to our runtime system's API that can be compiled with a conventional C++ compiler. Our translator makes modest transformations to the source code since the runtime system encapsulates most of the logic required for rank/region/dependency scheduling.

The final step involves linking the object files obtained from compiling the MATE program with the MATE runtime system library. This step will produce an executable binary file that runs as a regular MPI application.

4.2 Translation Process

We built our translator using the ROSE compiler infrastructure [51]. ROSE relies on the Edison Design Group front-end [98] to parse and generate the abstract syntax tree (AST) of an MPI C/C++ program. ROSE stores the AST in memory during the translation process and provides the tools to build a translator capable of analyzing and modifying input source code and producing modified source files.

One of the primary goals in designing our programming framework was to facilitate incremental development. For this reason, MATE applications can contain a combination of annotated and non-annotated source code. The only requirement is that MATE translates all source files containing calls to MPI, *#pragma mate* directives, or the *main* function. Source files without any references to MPI or MATE can be passed directly to the C++ compiler, as with conventional C/C++ source, and require no translation with MATE.

MATE does not support separate translation but instead requires that all annotated source files be translated together along with the file containing the *main* function. This design was

[2]For the purpose of brevity, when we refer to annotations, we include not only pragma directives, but also calls to the MATE API.

made in the interest of reducing runtime overheads. The effect is to describe MATE graphs only once at the beginning of execution (*main*), rather than at every graph block. With this rationale, we move scheduling overheads out of the application's critical path. Although this means longer translation times, we decided to prioritize runtime efficiency.

Translation produces one output C++ file per input C/C++ source file. These files can be compiled in parallel using the *mate-cxx* command, which redirects to the C++ compiler and provides flags required for compilation and linkage of MATE translated files.

Our translator processes input files in a series of steps. We will use the example code from Fig. 4.2, which represents the MATE-annotated portion of the example in Fig. 3.15, to illustrate the effects of each step.

```
1   #pragma mate graph
2   for (int i = 0; i < Iterations; i++)
3   {
4    #pragma mate region(compute) depends(pack*, unpack*, compute*@)
5     ApplyStencil(nL, uNew, uOld); Swap(&uNew, &uOld);
6
7    #pragma mate region(receive) depends(unpack*)
8     for (x in Neighbors) MPI_Irecv(recvBuf(x)←x);
9
10   #pragma mate region(pack) depends(compute, send*)
11    for (x in Neighbors) MPI_Pack(uNew→sendBuf(x));
12
13   #pragma mate region(send) depends(pack)
14    for (x in Neighbors) MPI_Isend(sendBuf(x)→x);
15
16   #pragma mate region(unpack) depends(compute, receive)
17    for (x in Neighbors) MPI_Unpack(uNew←recvBuf(x));
18  }
```

Figure 4.2: Annotated section of the code in 3.15.

4.2.1 Step I: MPI to MATE Call Replacement

MATE operates by 'intercepting' calls to the MPI library and redirecting them to its runtime system. This behavior is necessary to keep track of the execution of MATE regions, which will not advance until all the communication operations instantiated therein have finished. It is also required to support multiple local ranks since MPI blocking waits can suspend the

whole MATE process or result in deadlock, even if other ranks are ready to execute.

```
1  #pragma mate graph
2  for (int i = 0; i < Iterations; i++)
3  {
4   #pragma mate region(compute) depends(pack*, unpack*, compute*@)
5   { ApplyStencil(nL, uGrid, bGrid); Swap(&uGrid, &bGrid); }
6
7   #pragma mate region(receive) depends(unpack*)
8   { for (x in Neighbors) Mate_Irecv(recvBuf(x)←x); }
9
10  #pragma mate region(pack) depends(compute, send*)
11  { for (x in Neighbors) Mate_Pack(uGrid→sendBuf(x)); }
12
13  #pragma mate region(send) depends(pack)
14  { for (x in Neighbors) Mate_Isend(sendBuf(x)→x); }
15
16  #pragma mate region(unpack) depends(compute, receive)
17  { for (x in Neighbors) Mate_Unpack(uGrid←recvBuf(x)); }
18 }
19 }
```

Figure 4.3: Step 1 of translation replaces MPI calls with its equivalent MATE call.

Our translator traverses the AST of every source file in search for any function call with the '*MPI_*' prefix. Upon finding one such call, it validates whether MATE supports it. If that is the case, MATE replaces the call for its equivalent '*MATE_*' prefixed function call. Otherwise, it exits translation with an error. Fig. 4.3 shows the result of applying this step in the example code from Fig. 3.15. The translator replaces MPI calls (appendix E contains a detailed list of supported MPI functions) with their equivalent MATE call (*lines 8, 11, 14, 17*). We explain the semantics of these calls in section 4.3.

4.2.2 Step II: Parsing Graph Directives

The next step in the translation process parses all *#pragma mate graph* directives in the code. MATE's syntax requires that these directives be followed by either a basic block (*i.e.:* a single statement, or a group of statements enclosed within brackets) or a *for* loop statement[3]. If that is not the case (*e.g.,* the translator finds two or more non-enclosed statements), the translator

[3]We currently only support regular loops with a single monotonically increasing/decreasing accumulator. We have not yet encountered a test case that requires relaxing this requirement, although we could allow this feature in future developments.

will output an error message and exit. The translator stores the details of each *graph* block in memory to to preserve all the region and dependency information contained within.

```
1  Mate_EnableRegions({0, 1, 2, 3, 4});
2  int _nextRegionID;
3
4  // for (int i = 0; i < Iterations; i++)
5  while((_nextRegionID = Mate_GetNextRegionID()) ≠ ROOT_REGION)
6  switch(_nextRegionID)
7  {
8   case 0: // Compute Region
9     ApplyStencil(nL, uGrid, bGrid); Swap(&uGrid, &bGrid);
10    Mate_DisableRegion(); break;
11
12  case 1: // Receive Region
13    for (x in Neighbors) Mate_Irecv(recvBuf(x)←x);
14    Mate_DisableRegion(); break;
15
16  case 2: // Pack Region
17    for (x in Neighbors) Mate_Pack(uGrid→sendBuf(x));
18    Mate_DisableRegion(); break;
19
20  case 3: // Send Region
21    for (x in Neighbors) Mate_Isend(sendBuf(x)→x);
22    Mate_DisableRegion(); break;
23
24  case 4: // Unpack Region
25    for (x in Neighbors) Mate_Unpack(uGrid←recvBuf(x));
26    Mate_DisableRegion(); break;
27
28  default: Mate_ExitWithError('Invalid Region ID'); break;
29 }
```

Figure 4.4: Step 2 of translation creates scheduling structures for a region-level execution.

Fig. 4.4 shows the state of the code as a result from the transformations involved in this step. Our translator performs these transformations to the code surrounding each *graph* block to enable out-of-order execution. First, it replaces the *#pragma graph* directive with a *while* loop (*line 5*) to execute the enclosed partially ordered regions until all have finalized. It also inserts a call to *Mate_GetNextRegionID()* inside the *while*'s conditional to poll for identifier of an available region to execute (stored in the _nextRegionID variable declared at *line 2*). A distinguished *ROOT_REGION* value indicates that all regions have completed, causing the program to continue onto the following statement after the *while* loop. Second, it inserts a *switch* statement inside the *while* loop (*line 6*). This statement takes the value of *_nextRegionID* to 'jump' to the corresponding *case* statement, as defined in the next step.

Next, MATE parses the AST within each *graph* block in search of *mate region* directives. If it finds any stray statements located outside a MATE region, it will exit with an error. Otherwise, it inserts one *case* statement for each region within the *graph* block into the *switch* statement. It also inserts a *default* case, whose purpose is to catch any errors that may occur (*e.g.,* an unlikely exception in MATE's runtime system).

MATE assigns each region a unique identifier incrementally, corresponding to the order in which it finds them while traversing the AST. The translator prepends calls to *Mate_Enable Regions()* (*line 1*) that, when executed, inform the runtime system that (1) this is the beginning of a *graph* block, and (2) which regions are contained in this graph.

MATE inserts the original statements from each region that it encounters in the source code inside their corresponding *case* statement with an appended call to *Mate_DisableRegion()* (*lines 10, 14, 18, 22, and 26*), which prevents the region from executing again during the current instance of the *graph* block. Finally, a *break* statement is added to prevent other regions from executing.

4.2.3 Step III: Parsing For Loop Graphs

In case the *graph* directive annotates a *for* loop, we perform an additional step to support independent advancement of regions inside the loop, as shown in Fig. 4.5. This step entails replicating the original *for* induction variable, once for each region (*lines 10, 14, 18, 22, 26*), and initializing them with the same expression as that of the source code's *for* loop initializer.

To avoid memory conflicts, region-specific induction variables are formed by appending the region's id to the variable's name (*line 2*). Just like in a C/C++ *for* loop, MATE uses each region's iterator to apply the *for* increment and evaluate the loop conditional. However, unlike a C or C++ *for* loop, region-specific induction variables advance independently from each other, allowing one region to progress onto the next iteration while others remain in the current iteration.

```
1  Mate_EnableRegions({0, 1, 2, 3, 4});
2  int _i0 = 0, _i1 = 0, _i2 = 0, _i3 = 0, _i4 = 0;
3
4  int _nextRegionID;
5  while((_nextRegionID = Mate_GetNextRegionID()) ≠ MATE_ROOT_REGION)
6  switch(_nextRegionID)
7  {
8   case 0: // Compute Region
9     ApplyStencil(nL, uGrid, bGrid); Swap(&uGrid, &bGrid);
10    _i0++; if (_i0 < Iterations == false) Mate_DisableRegion(); break;
11
12  case 1: // Receive Region
13    for (x in Neighbors) Mate_Irecv(recvBuf(x)←x);
14    _i1++; if (_i1 < Iterations == false) Mate_DisableRegion(); break;
15
16  case 2: // Pack Region
17    for (x in Neighbors) Mate_Pack(uGrid→sendBuf(x));
18    _i2++; if (_i2 < Iterations == false) Mate_DisableRegion(); break;
19
20  case 3: // Send Region
21    for (x in Neighbors) Mate_Isend(sendBuf(x)→x);
22    _i3++; if (_i3 < Iterations == false) Mate_DisableRegion(); break;
23
24  case 4: // Unpack Region
25    for (x in Neighbors) Mate_Unpack(uGrid←recvBuf(x));
26    _i4++; if (_i4 < Iterations == false) Mate_DisableRegion(); break;
27
28  default: Mate_ExitWithError('Invalid Region ID'); break;
29 }
```

Figure 4.5: Step 3 of translation creates the structures to support *for*-loop based graphs.

MATE inserts the iterator conditional test at the end of every case statement. If the conditional results *false*, the *for* loop execution has ended for that particular region and MATE inserts a call to *Mate_DisableRegion()*[4].

4.2.4 Step IV: Replacing The Main Function

To initialize the MATE runtime system before the application starts, we intercept the initial call to the *main* function. Fig. 4.6 shows the two modifications involved in this step. First, the translator renames the original *main* function to *__mate_execute*, which accepts and returns the same arguments as *main*. Second, it creates a surrogate *main* function.

The surrogate *main* function's purpose is to instantiate and initialize the runtime system. MATE prepends calls to the runtime system's API at the beginning of *__mate_execute* (former

[4]MATE also performs this verification at the beginning of the region for the special case in which the conditional proves *false* in the first iteration. We omit this detail to simplify the explanation and the code.

```
1  #include <mate_process.h>
2
3  int main(int argc, char* argv[])
4  {
5   MATEProcess mate;
6   return mate.initialize(argc, argv);
7  }
8
9  int __execute_mate(int argc, char* argv[]) // The original main before translation
10 {
11  Mate_AddRegion(0); // Compute Region
12  Mate_AddRegion(1); // Receive Region
13  Mate_AddRegion(2); // Pack Region
14  Mate_AddRegion(3); // Send Region
15  Mate_AddRegion(4); // Unpack Region
16
17  Mate_AddDependency(0, 2, true);
18  Mate_AddDependency(0, 4, true);
19  Mate_AddDependency(1, 4, true);
20  Mate_AddDependency(2, 0, false);
21  Mate_AddDependency(2, 2, true);
22  Mate_AddDependency(3, 2, false);
23  Mate_AddDependency(4, 0, false);
24  Mate_AddDependency(4, 1, false);
25  Mate_AddInterRank(0, 0, true);
26
27  ...
28  Solver(N);
29  ...
30  Mate_Finalize();
31 }
```

Figure 4.6: The final step of translation creates a surrogate *main* function and defines the dependency graphs.

main function) to describe all the MATE graphs in the program. Since MATE graphs are user-defined and do not change during runtime, they only require to be defined once here.

To describe MATE graphs, we call *Mate_AddRegion()* (*lines 11-15*) passing the identifier of the region to add as argument. The *Mate_AddDependency()* and *Mate_AddInterRank()* functions (*lines 17-24*) accept three arguments: (1) the identifier of the source (dependent) region, (2) the identifier of the destination (depended) region, and (3) a boolean to indicate whether this is dependency on the previous iteration. We describe the semantics of these function calls in further detail in section 4.3.

Finally, any calls to *MPI_Finalize* are translated to *Mate_Finalize* to register the finalization of the current rank and yield execution. MATE will also perform this operation upon reaching the end of *__mate_execute* or upon reaching a *return* statement.

4.3 Runtime Support

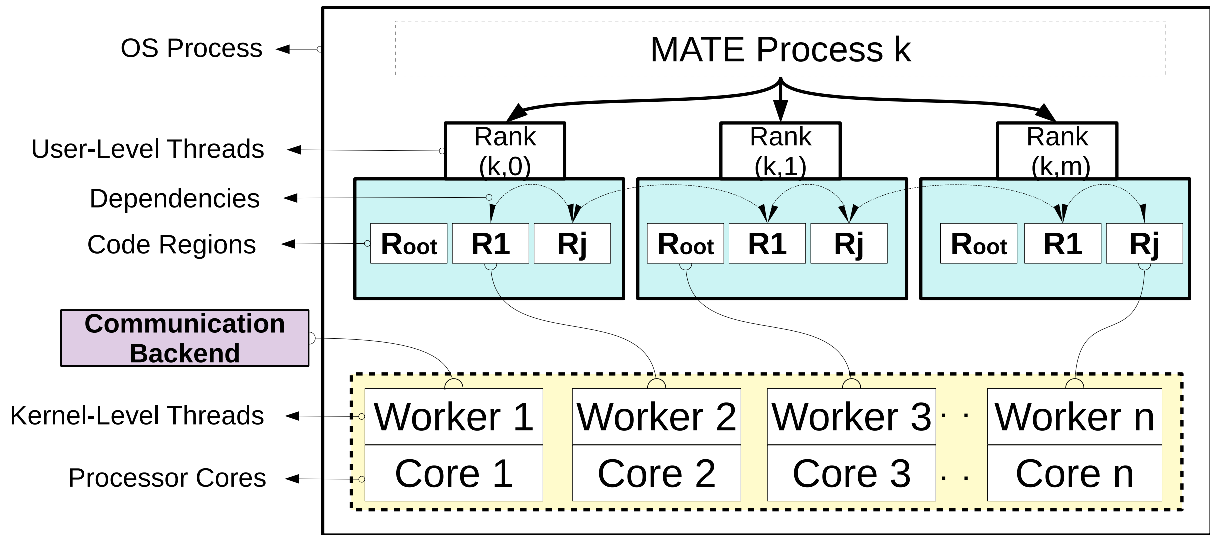

Figure 4.7: Decomposition model and implementation of a MATE process.

4.3.1 Runtime System Design

MATE applications execute as a set of k processes distributed across the system. Each such process contains a pool of m local MATE ranks, and a pool of n MATE workers, as shown in Fig. 4.7. The user specifies the number of MATE processes to instantiate and the number of workers and ranks per process at launch time using command-line arguments.

A user launches a MATE application using the same command for running MPI applications (*e.g.*, *srun*, *mpirun*, *aprun*). The total number of ranks across the system will be equal to the number of local ranks times the number of MATE processes (*i.e.*, $k \times m$). A user achieves overdecomposition by instantiating more ranks than available cores (c) in the system (*i.e.:* $k \times m > c$). To achieve optimal performance in a MATE application, a user should instantiate exactly as many MATE workers as available cores (or processing units) in the node.

The runtime system maintains all its scheduling metadata in memory as C++ objects. During initialization, the runtime creates a *singleton* (*i.e.,* a single instance of a class that is globally accessible) of the *MATEProcess* class that contains all the ranks, regions, and workers

in execution.

Ranks contain a set of user-defined regions (Fig. 4.7). The *root* region, created by default, represents all un-annotated code and executes sequentially. At the beginning of execution, all regions (except for *root*) start as *disabled*, meaning that they are not scheduled by the runtime system since the execution has not reached their containing graph block. Once execution reaches a MATE graph block, all the regions contained therein are *enabled*, meaning that they are now scheduled by the runtime system.

A region may exist in one of three states: *ready*, when all its dependencies are satisfied; *waiting*, when at least one of its dependencies are pending, or; *executing*, when it is currently executing. To determine whether a dependency has been satisfied, MATE keeps track of region's *step*, *i.e.*, the number of times that a region has executed. Every region's step starts at *zero* and increases by one every time the region executes. Additionally, we keep track of every MPI request produced by a region. A region does not advance (*i.e*, its *step* does not increase) until all its MPI requests have finished, even if it has yielded execution.

MATE Ranks may exist in one of five states: *finished*, when it has finished its execution; *ready*, when at least one of its regions is ready to execute; *barrier*, when it is suspended at a MATE Barrier (see section 4.4); *waiting*, when it is suspended for the completion of communication requests, and; *execute*, when it is being executed by one of the worker threads. At the beginning of a MATE application, all ranks start in a *ready* state since the *root* region (*i.e.*, unannotated code) starts *ready* by default.

MATE executes local ranks through the use of user-level threads (ULTs)[5]. ULTs are essential to our runtime system's design since they enable MATE ranks to suspend at different points of execution without the need to restructure code. A rank suspends its execution by calling

[5]We use the BOOST C++ *Symmetric Coroutine* library [94] to provide the yield/call semantics required to suspend and resume the execution of a MATE rank. Coroutines contain a pointer to a private stack allocation and the execution state of the processor at the moment of suspension. When resumed, the execution state moves back into the processor. This context switch is a low-latency operation (∼50 CPU cycles [111]) that does not require a system call.

the ULT's *yield()* method, which preserves its current state of execution.

MATE workers are kernel-level threads[6] whose purpose is to continuously look for ranks owned by the process that are ready to execute. When it finds a *ready* rank, the worker will resume its user-level thread and execute until no more *ready* regions remain. Whenever a worker is not executing a rank, it will check for the completion of the pending MPI requests produced by each of the rank's regions and re-evaluate their readiness.

4.3.2 Execution Model

Worker Execution

At the beginning of a MATE application, each MATE process executes a single (*master*) thread. This thread is in charge of runtime initialization and the creation of MATE workers. Fig. 4.8 shows the lifetime of a MATE worker.

MATE workers start executing their *execute()* method as soon as their kernel-level thread is instantiated by the *master* thread with a call to *pthread_create()* **(1)**. The first order of business is to set core affinity by calling *pthread_setaffinity()* **(2)** and yielding the execution of the current core (via *pthread_yield()*) to ensure the new affinity takes effect **(3)**. Concurrently, the master thread calls *MPI_Barrier()* **(4)** to guarantee a consistent start across all MATE processes for the purposes of timing measurements. In the final step of its initialization, the worker synchronizes with the master thread **(5a)** and the rest of the workers **(5b)** by calling *pthread_barrier_wait()*.

After passing the barrier, the worker will scan for unfinished ranks and execute them **(7)**, until all have finished **(6)**. Since workers share a common pool of local ranks, they use a rank-specific mutex lock **(8)** to ensure unique assignment of ranks to workers. A worker will operate on the rank only if it takes ownership of its lock **(9)**. No other worker can perform any operations on the rank's object or execute its ULT until the worker thread releases its lock.

[6]These are threads scheduled by the operating system and mapped to different cores to achieve parallelism.

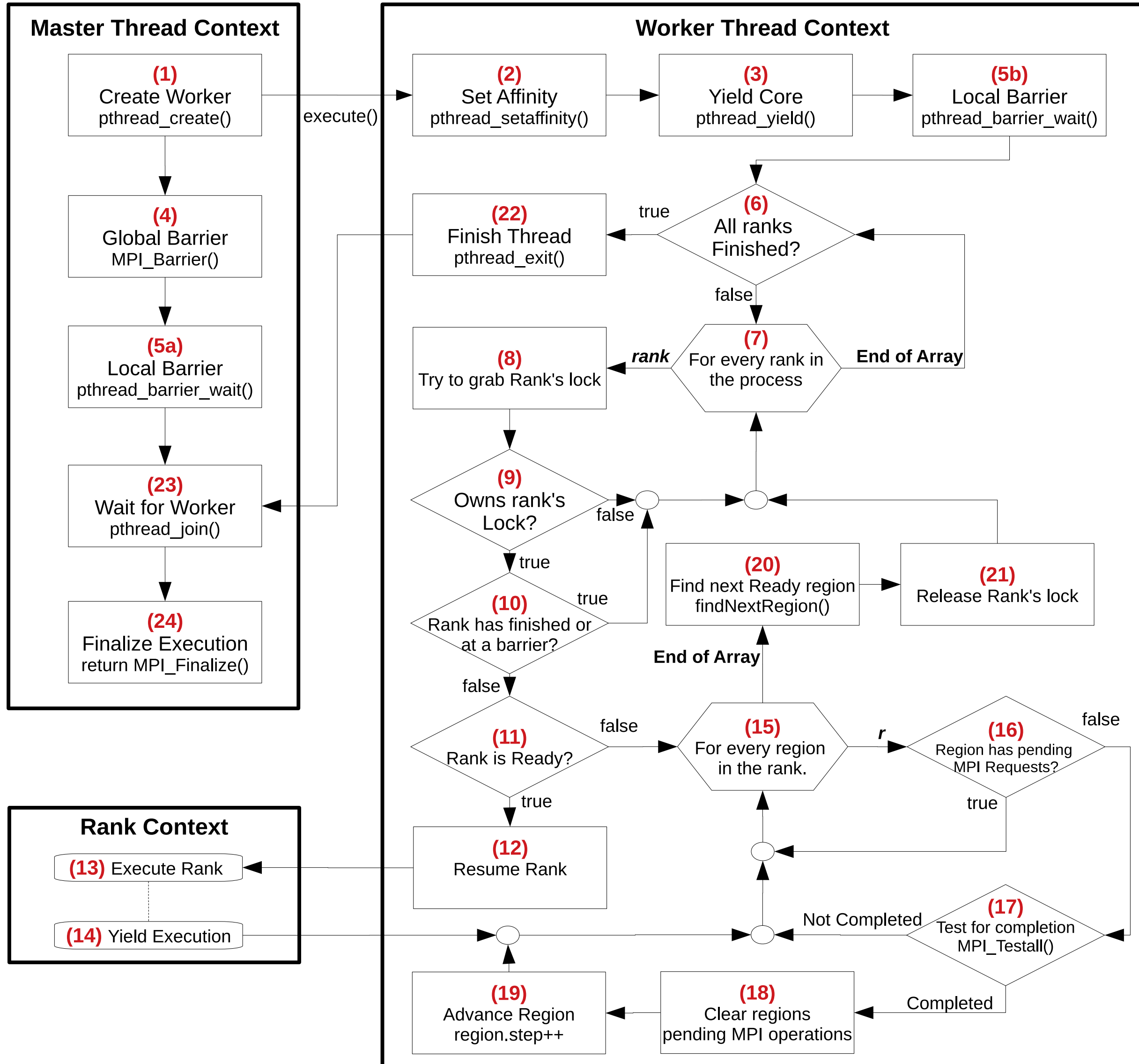

Figure 4.8: Lifetime of a MATE worker.

Once a worker takes ownership of a rank, it will query whether the rank has finished or is waiting at a barrier by determining if the rank is in the *barrier* state **(10)**. If either of these conditions fails, then the worker continues to the next rank **(7)**. Otherwise, the worker checks whether the rank is ready to execute a region **(11)**.

If the rank is ready, the worker will resume the rank's execution **(12)**. If the rank is not ready to execute, it skips this step **(15)**. When a worker calls the rank's ULT *resume()* method,

the execution context will switch from that of the MATE worker to that of the MATE rank **(13)**. The inverse switch will occur when the rank calls its own ULT's *yield()* method **(14)**.

An important fact in our implementation is that different workers (cores) can resume the execution of any given rank during its lifetime (although one at a time). This flexibility in migrating ranks onto idle workers as soon as possible maximizes MATE's potential for overlapping computation and communication.

Regardless of whether the rank was ready to execute or not, the worker thread's next goal is to query the rank's pending MPI operations **(15)**, by evaluating every of its regions' pending MPI requests **(16)**. In case a region does not have pending requests, the worker will continue onto the next region **(15)**. Whenever the worker finds a region with at least one pending request, it will call *MPI_Testall()* **(17)** to poll for completion. This function will return *true* if, and only if, all of the region's requests have finished. In that case, the worker thread clears the region's pending requests **(18)** and advances its *step* counter by one **(19)**, meaning that the region has finally finished their previous stint. If *MPI_Testall* returns *false*, it means that at least one of the requests have not yet and that the region cannot advance yet.

The worker will continue polling for available regions **(20)**. When all regions have been polled, it surrenders the rank's ownership by releasing its lock **(21)** and looks for the next rank **(7)**. If all ranks in the process have finished, the worker thread exits (via *pthread_exit()*), removing the KLT from the OS scheduler **(22)** and signaling the *pthread_join()* **(23)** operation in the master thread to indicate the current thread has finished. Once all threads exit (*i.e.*, every call to *pthread_join* meets a reciprocal *pthread_exit* call from each worker thread), the master thread finalizes the process execution **(24)**. A MATE application finishes when all MATE processes finalize.

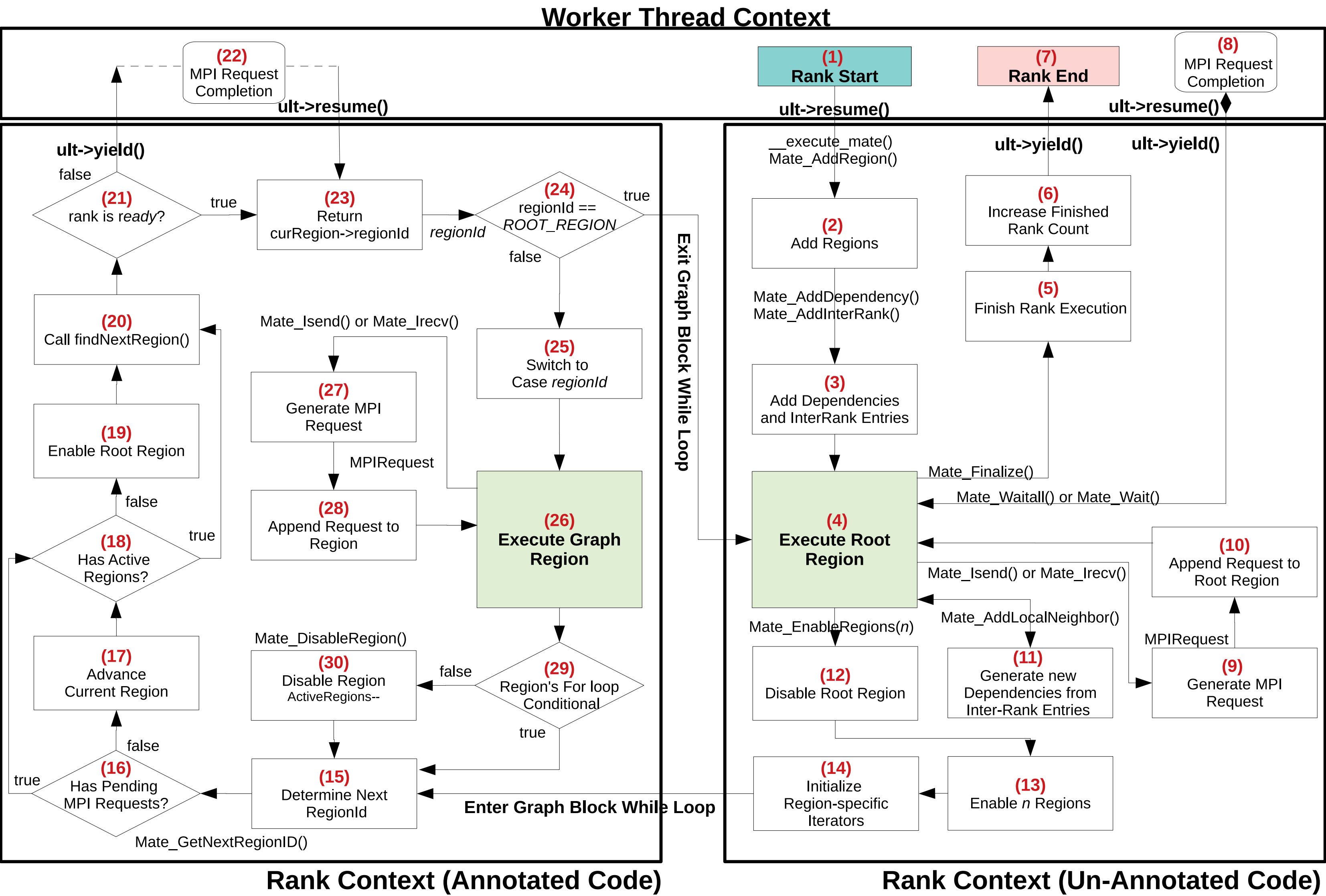

Figure 4.9: Lifetime of a MATE Rank.

Rank Execution

Fig. 4.9 illustrates the lifetime of a MATE rank. The figure distinguishes between the execution context of a MATE worker (top), and that of a MATE rank while it is executing un-annotated code (bottom-right), and while it is executing annotated code (bottom-left). We identify each step with a number for easy reference.

MATE defines *__execute_mate()* as the initial entry point for every rank's ULT. The first time a worker executes a rank **(1)**, it will first execute the graph initialization calls contained therein. Each call to *Mate_AddRegion()* **(2)** creates a new region; *Mate_AddDependency* **(3)** adds a new region dependency, and; *Mate_AddInterRank()* **(3)** creates a new inter-rank *pre-dependency*. Inter-rank pre-dependencies are not actual dependency yet since the rank has not declared any local neighbor ranks. They will become actual dependencies when a rank executes the *Mate_AddLocalNeighbor* **(11)** function. This function declares a new local neighbor which will inherit all the inter-rank pre-dependencies declared in **(3)**.

After creating the description of the program's graphs blocks, the rank continues to execute what formerly was the *main* function of the program. That is, the rank will start executing its *root* region **(4)**. At this point, the rank executes un-annotated code.

Executing Un-Annotated Code

Calls to *Mate_Isend* and *Mate_Irecv* **(9)** will generate new MPI requests that the rank appends to the *root* region's **(10)**. Since the *root* region executes under the same semantics as a normal MPI program, the only way to verify the completion of its request vector is via calls to *Mate_Wait()* or *Mate_Waitall()*. These calls will suspend the execution of the *root* region by calling the rank's ULT *yield()* method and returning to the worker's execution context. The rank will only resume after one of the worker threads tests the completion **(8)** of all of the MPI requests in the *root* region.

Calling *Mate_Finalize()*, executing a *return* statement in the scope of *__execute_mate()*,

or simply reaching its closing bracket will prompt the runtime system to finalize the rank. At this point, the rank sets its state to *finished* **(5)**, and increases the process's finished rank counter by one **(6)**. This counter determines whether the application has finished. Finally, it yields execution back to the worker's context **(7)**. After finalization, the rank will not be resumed again.

Other MATE API calls, such as the ones used to identify the rank, do not produce major changes in the execution of the rank and thus are not shown in the diagram. We also exclude collective calls, such as *Mate_LocalBarrier*, *Mate_Barrier*, *Mate_LocalBcast*, *Mate_Bcast*, from the diagram as we explain them in detail in section 4.4.

Upon finding a MATE-annotated graph block, the *root* region will call *Mate_Enable Regions(n)*, with *n* being a list of regions to enable (see Fig. 4.5, *line 1*). Upon calling this function, the rank disables its *root* region **(12)** by setting it as *inactive*, enables each one of the regions passed by argument **(13)**, and increases the rank's *activeRegions* counter with as many regions contained in the graph. If this is a *for* loop graph block, the rank will also initialize the region-specific induction variables **(14)** (see Fig. 4.5, *line 2*). At this point, the rank enters the graph block's *while* loop.

Executing Annotated Code

At the beginning of the block's *while* loop, and every time a region finishes execution, the rank calls *Mate_GetNextRegionID()* (see Fig. 4.5, *line 5*) to determine the next region to execute or determine the termination of the *while* loop **(15)**. This function performs three operations:

1. It checks whether the currently executing region (*i.e.*, the one that has just finished executing) has no pending MPI requests **(16)**. If that is the case, then it advances the region by increasing its *step* counter by one **(17)**, meaning that the region has completely finished executing this iteration. Otherwise, it skips this step.

2. It checks whether the rank has any active regions by evaluating *ActiveRegions* > 0 **(18)**. If

this is *false*, then it means that the rank is ready to exit the graph block, and thus the rank reactivates its *root* region **(19)**. Otherwise, it performs no action.

3. It calls the rank's *findNextRegion()* to define the next region to execute **(20)**.

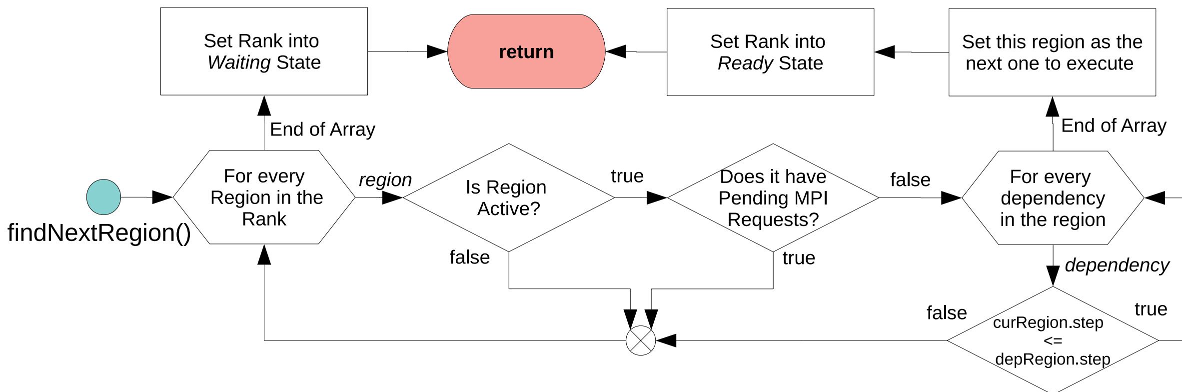

Figure 4.10: Flowchart of the findNextRegion method.

Fig. 4.10 shows the operation of a rank's *findNextRegion()* method. This method iterates over all the regions in the rank. If a region is inactive or has pending MPI requests, then it skips it and continues onto the next region. Otherwise, it evaluates whether all the region's dependencies are satisfied. A dependency is satisfied if and only if the region's *step* is smaller or equal than the depended region's step[7]. If *false*, the depended region has not executed recently enough to enable this region and the method falls back to evaluate the next region. If *true* for all dependencies, it means that the region is ready to execute.

findNextRegion() succeeds as soon as it finds a region that is ready to execute. If that's the case, it sets the rank to *ready* state and assigns the ready region as the next in line for execution. If it fails to find a ready region, it sets the rank's to the *waiting* state and returns.

If the rank is in *waiting* state, it means that one or more regions have pending MPI requests to complete before becoming ready again **(22)**. If this is the case, the rank yields execution by calling its ULT's yield method. The rank will resume execution later when one or

[7] If this is a previous-iteration dependency we relax this requirement by one step.

more of its regions become ready. On the other hand, if the rank is in the *ready* state, it will not yield execution. Regardless of the case, the rank returns from its call to *Mate_GetNextRegionID()* with the region identifier to execute next **(23)**.

The translated MATE application uses the region identifier *regionId* to determine the next region to execute **(24)**. If this identifier corresponds to the *root* region (*regionId* == *ROOT_REGION*) then the rank exits the graph block *while* loop and continues to execute non-annotated code **(4)**. Otherwise, it switches to the corresponding region's *case* statement **(25)**. During the execution of a graph block region, the MATE runtime system will accept calls to *Mate_Isend* and *Mate_Irecv*. These functions will create new MPI requests **(27)** and append them to the currently executing region **(28)**. Calls to collective communication or barrier operations will execute as ordinary MPI code. However, since these operations require process-wide synchronization, they may induce a deadlock if the dependency graph is not carefully specified.

After a region has finished executing, the rank evaluates its *for* loop conditional **(29)**. If this condition evaluates to *false*, meaning that the loop must finish, it calls *Mate_Disable Region()* **(30)**. This function decreases the number of active regions in the rank by one and disables the current region. If the conditional evaluates to *true*, meaning that the region is still active, then it performs no action and iterates back to evaluate the *while* loop conditional **(15)**. In case this is not a *for* loop graph block, the rank will call *Mate_DisableRegion* unconditionally **(30)**.

4.4 Communication Backend

Our current implementation of MATE supports a subset of the operations specified in the MPI interface (See Appendix E for a list of currently supported MPI functions). MATE provides partial support for MPI two-sided operations (limited to *Send/Recv* and *Isend/Irecv*) and integrates them into the model for computation/communication overlap. We also provide support for contiguous and non-contiguous datatypes. On the other hand, MATE does not currently

support dynamic process creation, one-sided communication, nor user-defined MPI groups and communicators besides *MPI_COMM_WORLD*. However, our model does not rule out their implementation. Next, we describe the implementation of point-to-point, barrier, and collective communication operations in MATE.

4.4.1 Point-To-Point Communication

The main way in which two ranks exchange data in a conventional MPI application is by performing a point-to-point message exchange operation. The sender rank initiates a message by executing *MPI_Isend()*[8]. Similarly, the receiver rank will make a reciprocal call to *MPI_Irecv()* to announce its readiness to receive the message. This protocol requires that both ranks agree on the exchange of the message.

Based on the arguments passed in the function call, both *MPI_Isend* and *MPI_Irecv* will generate a new MPI *request* which stores all the information required by MPI to perform the exchange. Fig. 4.11 shows the fields that make up an MPI request. The *Count* field indicates how many elements (not bytes) to exchange. The *Datatype* field indicates the data type of the element, expressed in MPI format (*e.g.* MPI_INT, MPI_DOUBLE) and not native size format (*e.g.* *sizeof(double)*) to guarantee compatibility across different platforms. The *Buffer Pointer* field indicates either the source buffer (in case of the sender rank) or destination buffer (in case of the receiver rank) pointer. The *Communicator* field indicates which MPI subgroup of ranks to use in the exchange (typically, *MPI_COMM_WORLD*, which includes all ranks in the execution). The *Source* field indicates the rank of the process sending the message. MPI does not require this field as a function call argument in *MPI_Isend* since MPI fills it with the sender's rank. Similarly for the *Destination* field which indicates the receiver rank and is not required when issuing an *MPI_Irecv* request. Finally, the *Tag* field is a user-defined numerical identifier.

The only conditions required for MPI to complete a message exchange is that the *Source*,

[8] Also with *MPI_Send()*, which is equivalent to *MPI_Isend()* followed by an immediate call to *MPI_Wait()*.

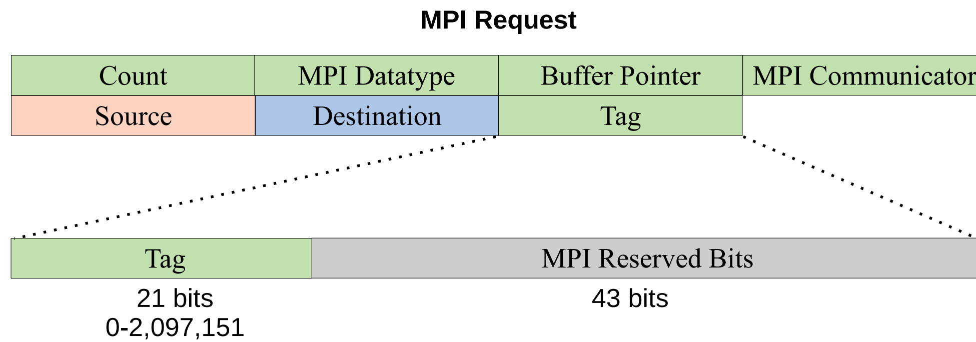

Figure 4.11: Structure of an MPI request.

Destination, and *Tag* fields coincide[9]in both send and receive requests. The tag field allows a programmer to individually identify different messages exchanged between the same pair of ranks. Although the *MPI_Irecv* and *MPI_Isend* function calls accept an 8-byte (64 bits) integer argument for the tag field, each particular MPI library implementation restricts its actual value range. Indeed, MPI libraries reserve part of the tag to store other information (possibly source and destination rank ids) to save on the size of the metadata portion of the packets sent through the network. In all of our experiments, we used the Cray-MPICH 7.7.0 library which reserves 43 of the 64 bits in the tag field. In this case, a programmer can specify a tag value ranging from 0 to 2,097,151.

One of the first and most difficult challenges we faced in developing our model was to find a way to identify local ranks individually in the exchange of MPI messages. Since we use MPI as our backend communication library, we can only use the *source* and *destination* fields to identify which MATE processes are involved in the exchange. However, these fields only specify MPI processes, and not local MATE ranks that arise from overdecomposition.

The simplest way to overcome this problem is to ‘wrap’ the message payload with an additional MATE metadata header containing the local identifier of the sender and receiver ranks. The receiver MATE process would then ‘unwrap’ the header and copy the data payload onto the

[9]A notable exception is the use of wildcards (*e.g.*, MPI_ANY_SOURCE, MPI_ANY_TAG) which are not currently supported in MATE.

actual rank's buffer. We immediately disregarded this solution due to its many drawbacks. First, the sender process needs to create a new internal buffer to store the payload plus the metadata before issuing the *MPI_Isend* request. Not only this required additional per-message memory allocations but also an additional *memcpy* of the message's payload. Second, we would have to design complex MATE-to-MPI bookkeeping to keep track of local request completion.

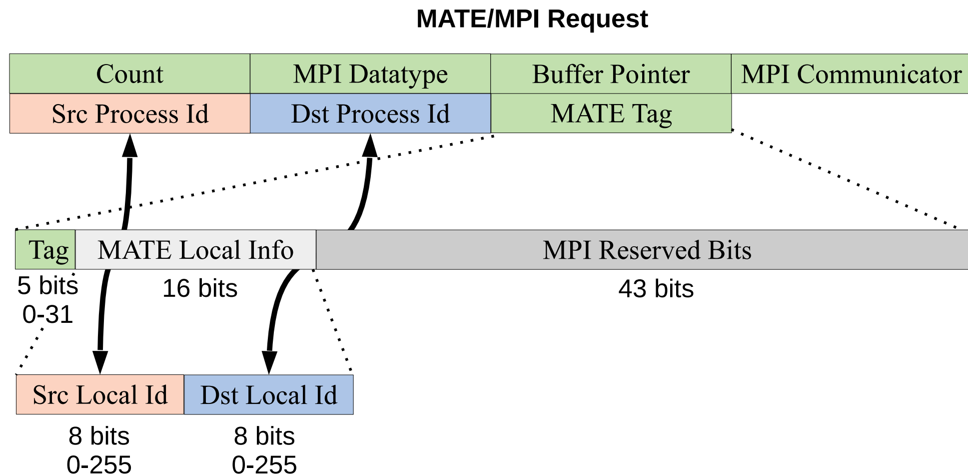

Figure 4.12: Structure of a MATE/MPI request, including local rank identifiers in the *tag* field.

We came to a much more efficient solution after realizing we could 'steal' bits from the *tag* field to identify the local identifiers of the sender and receiver ranks. Fig. 4.12 shows the format of a MATE/MPI message request. We reserved 16 out of the 21 bits in the user-definable portion of the tag field to indicate the local sender/receiver ids, using 8 bits for each one. Thus, a full MATE rank id is the concatenation of the MPI (process) rank and the local rank within the process. Since MPI will complete the exchange when the source, destination, and tag fields coincide, this approach entirely delegates MPI the responsibility of completing the request without the need for metadata headers or additional *memcpy*s.

The MATE/MPI request format we designed restricts the actual range of values of the *tag* field to 5 bits. This new range means that MATE applications over Cray-MPICH can only differentiate among 32 different messages exchange by the same sender/receiver pair. This restriction has not impacted any of our test cases, but it might be a limitation in a large application. In such

cases, it may be necessary to define additional communicators. as we only needed to 'reduce' the larger-than-32 tag identifiers in their original codes to smaller numbers. Additionally, using 8 bits for identifying local rank ids means that we currently support 256 local ranks per MATE process. This maximum will allow, for example, instantiating MATE process with eight MATE workers and with eight ranks per core each. We did not exceed this limit in our experiments.

4.4.2 Barriers

MATE provides two barrier mechanisms, *Mate_GlobalBarrier()* and *Mate_LocalBarrier()* for ranks to synchronize with each other during execution. *Mate_GlobalBarrier()* has the same semantics as *MPI_Barrier(MPI_COMM_WORLD)* in that all ranks in execution will wait for each other at that point. In fact, our translator automatically renames every instance of *MPI_Barrier* to *Mate_GlobalBarrier*. The second mechanism, *Mate_LocalBarrier()* synchronizes all the local ranks within the scope of a single MATE process. MATE ranks may call these mechanisms at any point of execution, even when executing a region within a graph block.

Both barrier mechanisms are aliases to a common *Mate_Barrier(bool isGlobal)* function. That is, they perform the same operations but only differ in the value of the *isGlobal* argument they pass. Fig. 4.13 shows the steps that a rank follows in executing *Mate_Barrier()*.

1. Obtain the rank's own pointer by calling *getCurrentRank()*.
2. Increase the MATEProcess counter *barrier* counter by one.
3. Evaluate whether the *barrier* counter equals the number of local ranks. If this is *false*, it means that not all local ranks have arrived at the barrier yet. In that case, the rank sets its *isBarrier* flag to *true* so that it will not be scheduled for execution again until all the other ranks have met the barrier, and then yields execution. If *counter* equals the number of local ranks, it means that all other ranks have arrived at the barrier. Therefore, the current rank becomes the 'root rank' of the barrier and executes the rest of the steps.

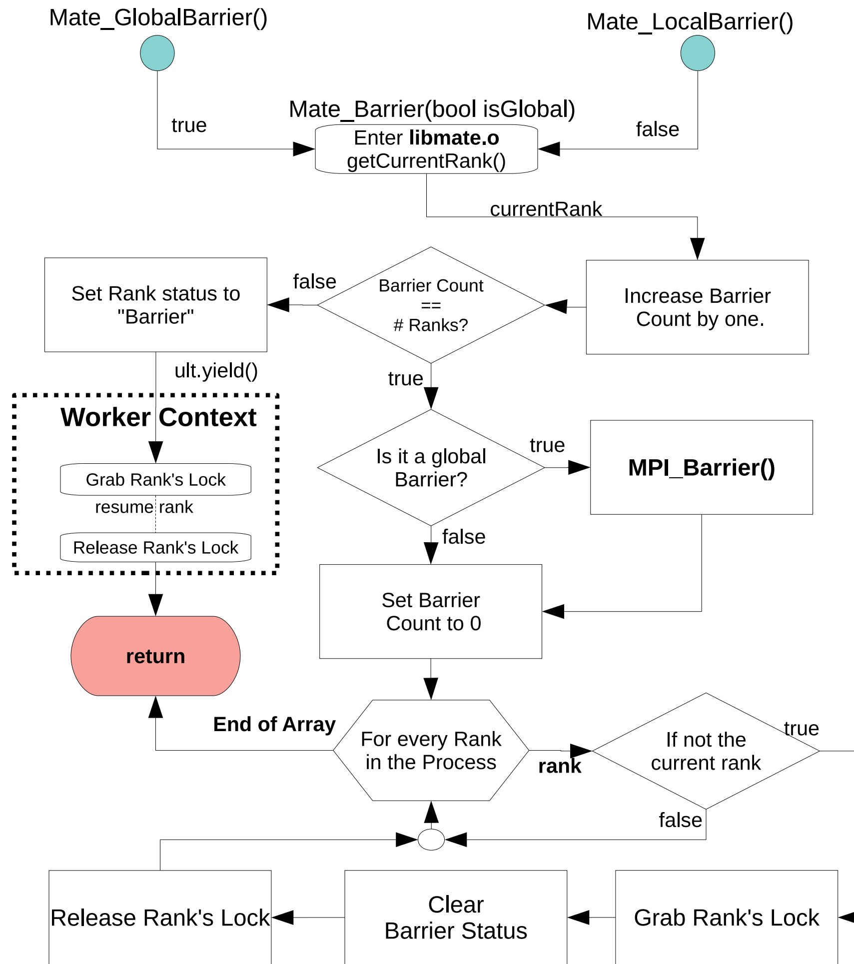

Figure 4.13: Flowchart of MATE barrier mechanisms.

4. In case the *isGlobal* flag is *true*, then the root rank calls *MPI_Barrier()* to synchronize all ranks across MATE processes. Otherwise, it omits this step.
5. The root rank re-initializes the *barrier* counter to zero for the next time a barrier executes.
6. The root rank grabs the lock of every other rank in the local process. Taking ownership of every rank is an important step to avoid concurrency problems with ranks that may still be in execution by other MATE workers. After obtaining the locks, it will set their *barrier* flag to false and release their lock. The root rank then returns from the call.
7. All non-root ranks resume execution and return from the call as well.

4.4.3 Collective Communication

MATE supports several MPI collective communication functions, including *Mate_Bcast, Mate_Reduce, Mate_Scatter, Mate_Gather, Mate_Allgather, Mate_Allgatherv*, and *Mate_Allreduce*. In our test applications, collective communication did not contribute a significant cost. Therefore, this support is designed with simplicity in mind rather than efficiency. Integration of collective operations into the MATE model remains a future work. We designed a general strategy for collective communication that meets our goals. Fig. 4.14 shows the steps of our algorithm.

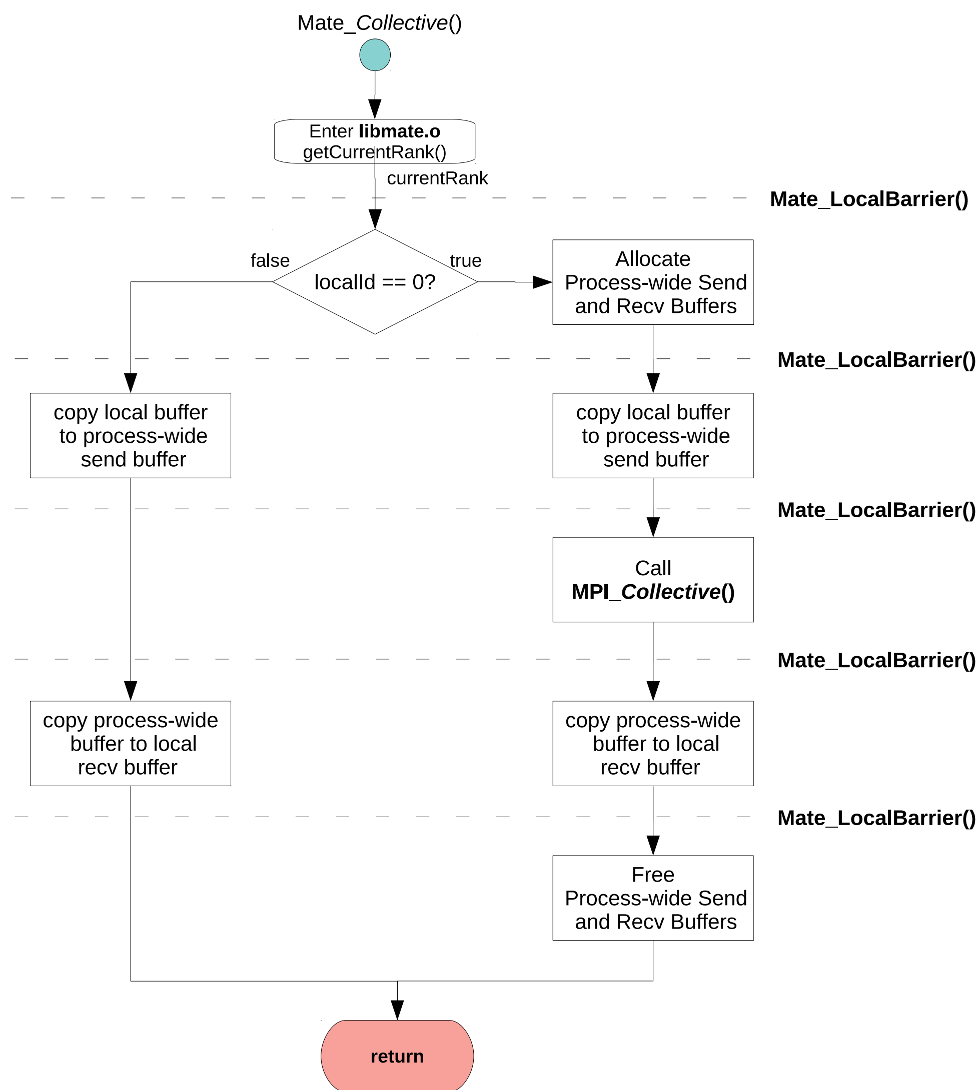

Figure 4.14: Flowchart of MATE collecive communication operations.

1. The rank determines whether it is the *root* rank (*i.e.*, it's local rank is 0).
2. The *root* rank allocates process-wide send and receive buffers for the collective operations (although most collective operations require only one buffer).
3. Every rank copies their local send buffer onto the process-wide send buffer.
4. The *root* rank performs the MPI equivalent of the collective in representation of the whole MATE process using the process-wide recv and send buffers.
5. Every rank copies their part of the process-wide buffer onto their local receive buffer.
6. The *root* rank deallocates the process-wide send and receive buffers.
7. Every rank returns from the call.

Acknowledgements

This chapter is, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

This chapter is also, in part, a reprint of the material contained in the article: "*Toucan - A Translator for Communication Tolerant MPI Applications*", by Sergio M. Martin, Marsha J. Berger, and Scott B. Baden, which appears in the Proceedings of 1st International Parallel and Distributed Processing Symposium (IPDPS 2017), Orlando, FL, USA, June 2017. This dissertation's author was the primary investigator and author of this paper.

Chapter 5

Test Case I: Jacobi3D

5.1 Overview

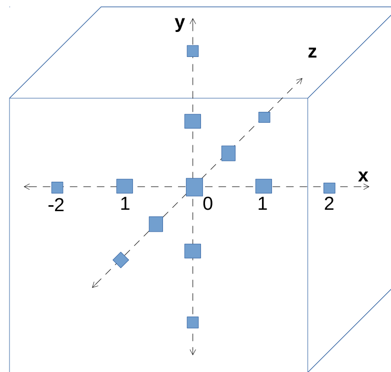

Figure 5.1: 13-Point Stencil on a three-dimensional grid.

Our first test case is *Jacobi3D*, a solver for the 3D Poisson equation subject to Dirichlet boundary conditions which uses a finite-different method. The computation applies Jacobi updates over all the elements of the grid using a 13-point central difference [119] stencil that updates each element with the average of its a central point and four nearby points per axis in a 2-point deep straight line, as illustrated in Fig. 5.1. This test case is a proxy for single level structured grid applications. Although structured and multi-grid solver are more complex [2], Jacobi3D represents a typical *hotspot* (*i.e.* time-consuming section) of a single mesh sweep.

5.1.1 Computation

The iteration logic of Jacobi3D's solver is similar to that of the pseudocode presented in Section 3.1.1 (Fig. 3.3), but extended to three dimensions. This code solves the equation $\Delta U = f$ equation in a box with $f = 0$ on $\partial\Omega$, the boundary of the box. The computational kernel of this code appears in Fig. 5.2[1]. Ranks define a boundary box that defines the starting and ending (x,y,z) indexes they use to iterate over all the elements of a 3D subgrid.

```
for (int z = start.z; z < end.z; z++)
for (int y = start.y; y < end.y; y++)
for (int x = start.x; x < end.x; x++)
{
 // C0: Coefficient for the central element
 Unew[x,y,z]  = C0 *  Uold[x,y,z];
 // C1: Coefficient for 1-deep nearby elements
 Unew[x,y,z] += C1 * (Uold[x+1,y,z] + Uold[x-1,y,z] + Uold[x,y,z];
                      Uold[x,y+1,z] + Uold[x,y-1,z] +
                      Uold[x,y,z-1] + Uold[x,y,z+1]);
 // C2: Coefficient for 2-deep nearby elements
 Unew[x,y,z] += C2 * (Uold[x+2,y,z] + Uold[x-2,y,z] +
                      Uold[x,y+2,z] + Uold[x,y-2,z] +
                      Uold[x,y,z-2] + Uold[x,y,z+2]);
}
```

Figure 5.2: Pseudo-code of the solver kernel of Jacobi3D.

The solver uses two grids: the current iteration's grid (Unew), and the previous iteration's grid (Uold). Each stencil update to the current grid element (Unew[x,y,z]) is a function of the values of the previous iteration's grid (Uold) elements, starting from the central point (Uold[x,y,z]), multiplied by the central element coefficient (*C0*). Added to this update, it uses the sum of the values of the neighboring points in the stencil at a Manhattan distance of one (*e.g.,* Uold[x+1,y,z] and Uold[x-1,y,z]), weighted by the 1-deep coefficient (*C1*) plus the sum of the points at a Manhattan distance of two (*e.g.,* Uold[x+2,y,z] and Uold[x-2,y,z]) weighted by *C2*.

We employ the cache-blocking and node-mapping optimizations described in Appendix B to maximize the performance of the solver and use the computational and memory resources to the best of their capacities. Having an optimal baseline is essential not only to represent

[1] In the original code from Fig. 3.3, this part is contained in *ApplyStencil()*.

highly optimized real-world applications but also to highlight the benefits of communication-reducing models. Any communication overheads that our model can reduce or hide will have more substantial effects in the absolute performance of the application if the base algorithm yields a maximum efficiency during its computational phase.

5.1.2 Verification

Our example verificaties the result's correctness with the routine shown in Fig. 5.3. The code calculates the local L2 residual of its subgrid, *i.e.,* the sum across all the elements in the subgrid of the squared difference between their calculated solution and their theoretical solution. In our example, the calculated solution is the value of *Uold* subgrid after executing the iterative solver, and the expected solution is a zero-valued grid.

```
1 double res;
2 double GlobalResidual = 0.0;
3 double LocalResidual  = 0.0;
4 for (int z = start.z; z < end.z; z++)
5  for (int y = start.y; y < end.y; y++)
6   for (int x = start.x; x < end.x; x++)
7    LocalResidual += Uold[x,y,z] * Uold[x,y,z];
8 MPI_Reduce (&LocalResidual, &res, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
9 GlobalResidual = sqrt(res/(N*N*N));
```

Figure 5.3: Verification code for Jacobi3D.

The solver computes the local contribution to the residual across all ranks and uses *MPI_Reduce* to sum these values, storing the result in a temporary variable (*res*). The global L2 residual is computed by taking the square root of the average residual across all the elements in the grid (res/N^3). To ensure that all our experiments produce a correct result, we compare the value of the global L2 residual and verify that they are equal.

5.2 Strong Scaling Studies

To evaluate the performance of Jacobi3D, we conducted a strong scaling study on our two computational testbeds (appendix A). In a strong scaling study, we keep the grid size constant while increasing the number of cores/nodes. The purpose of this study is to evaluate the effectiveness of MATE under varying amounts of communication overheads, which grow with the number of cores.

On a small scale (*e.g.,* 64 nodes), the cost of computation is higher than the cost of communication since the amount of work overwhelms communication requirements. However, as we scale up the number of cores in execution, the size of per-core subgrid decreases. The effect of decreasing per-core subgrid sizes is threefold. First, the amount of computational work that each core needs to perform decreases relative to the cost of communication. Second, the cost of network latency is intensified at larger scales since the average distance between any two nodes also increases.

We designed our study so that, at the largest scale, the size per subgrid variable requires approximately 1% of the total node memory in each computational testbed. This size is representative of a structured adaptive mesh solver for solving combustion problems [122] that employ up to a hundred variables. We compare the performance of six variants of the solver:

1. *Basic MPI* establishes the performance baseline upon which we compare the benefit from our other variants. Although this variant implements all the optimizations mentioned earlier, it does not attempt to overlap communication with computation.

2. *Over-MPI* employs overdecomposition as the only mechanism for hiding the cost of network communication and does not use MATE's hierarchical decomposition. This variant uses the identical code to *Basic-MPI*. To enable overdecomposition, we translate and run this variant using MATE's runtime system.

3. The *Olap-MPI* variant improves the baseline code from *Basic-MPI* with a split-phase

strategy that divides the rank's grid into smaller 3D tiles, each computed separately. This subdivision produces a similar effect as overdecomposition in MATE, as each tile requires separate communication and computation operations.

Fig. 5.4 shows the simplified pseudo-code of the manually overlapping variant. This variant divides each iteration into three distinct phases. First, it traverses over the 3D distribution of $nTilesX \times nTilesY \times nTilesZ$ tiles to initialize the receive requests for the boundaries of every tile (tx,ty,tz) (*lines 3-4*). Second, it performs the same traversal again to compute, initiate packing, and send requests for each tile (*lines 6-11*). The effect of this phase is to initialize send operations as soon as each tile's computation has finished, overlapping computation with computation. Lastly, it waits for the receive requests to finish and unpacks their buffers (*lines 13-17*).

```
1  for (int i = 0; i < Iterations; i++)
2  {
3   for (tx = 1:nTilesX, ty = 1:nTilesY, tz = 1:nTilesZ)
4    // Issue MPI_Irecvs for tile(tx,ty,tz);
5
6   for (tx = 1:nTilesX, ty = 1:nTilesY, tz = 1:nTilesZ)
7   {
8    // Compute tile(tx,ty,tz);
9    // Pack tile(tx,ty,tz) boundaries;
10   // Issue MPI_Isends for tile(tx,ty,tz);
11  }
12
13  for (tx = 1:nTilesX, ty = 1:nTilesY, tz = 1:nTilesZ)
14  {
15   // MPI_Waitall(requests for tile(tx,ty,tz));
16   // Unpack tile(tx,ty,tz) neighbor boundaries;
17  }
18 }
```

Figure 5.4: Pseudo-code of the manually overlapping variant of Jacobi3D.

4. The *Toucan* variant employs hierarchical overdecomposition but annotates the code with the fixed 3-region syntax prescribed in the *Toucan* Model [67], instead of MATE's generalized annotation model. The result is that the Toucan variant requires process-wide barriers to prevent data hazards among overdecomposed ranks whereas the MATE avoids this problem by utilizing inter-rank dependencies.

5. *MPI+OpenMP* combines MPI and OpenMP to provide a 2-level decomposition (similar to MATE's) in which multiple OpenMP threads share the work assigned to a single process (as described in Fig. 2.12), and thus do not explicitly move data on-node. This variant, however, does not employ overdecomposition nor any mechanism for overlapping communication with computation.

6. *MATE* applies all the optimizations of the MATE model as described in chapter 3. We use the same graph configuration as the one described in Fig. 3.16, including inter-rank dependencies among ranks located in the same MATE process. We employ overdecomposition in all our experiments to hide the cost of inter-node communication.

5.2.1 Cori Phase I (Haswell)

We ran our six variants over a range of 128 to 1024 nodes (4096 to 32768 Haswell cores), doubling the number of nodes at each scaling step. We performed 200 solver iterations on a grid of $n = 4096^3$ cells. We used 4096 cells per side since this number is divisible by 32, the per-side distribution of processes used in the *Basic-MPI* variant on 32768 cores. At 512 nodes, the grid consumes 1.56% of the total system's memory.

We ran the MPI variants using one MPI process per core, and an on-node rank distribution of $4 \times 4 \times 2$. The highest performing configuration for the *MPI+OpenMP* variant was 4 MPI processes $\times$ 8 OpenMP threads on each node with a local thread distribution of $1 \times 4 \times 2$ within each MPI process. For the *MATE* variant, we determined that the best configuration uses 4 MATE processes $\times$ 8 threads per node, with 32 MATE ranks per process with a local rank distribution of $1 \times 8 \times 4$. This configuration corresponds to an overdecomposition factor[2] of 4.

Fig. 5.5 shows the results of our study, plotting performance (TFlop/s). We performed three runs of each (variant, node count) combination and found less than $\leq$1% variation in wallclock time between runs. We report the best run for each combination. Results show that

[2]We define *overdecomposition factor* as the ratio of MATE ranks per core.

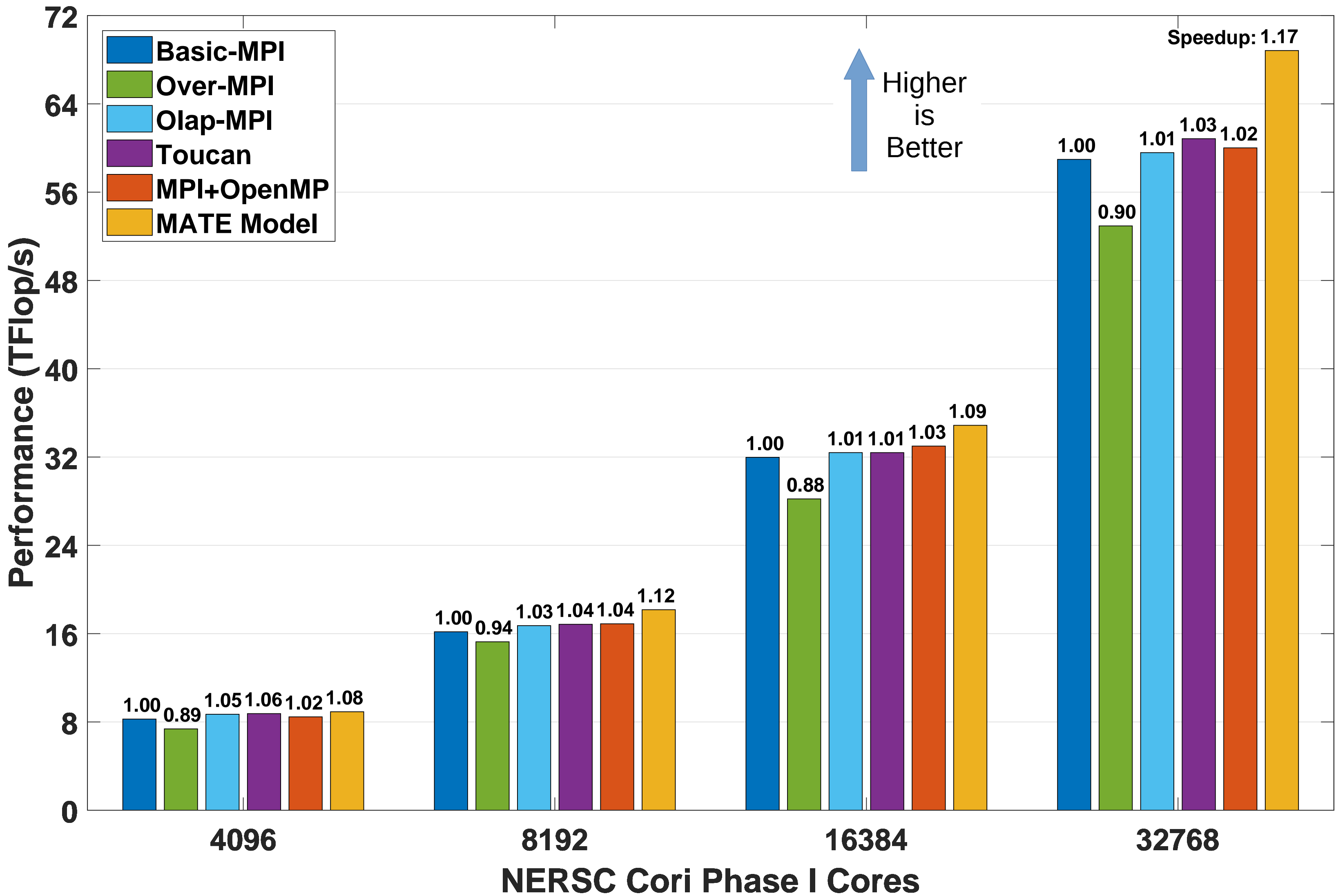

Figure 5.5: Strong Scaling results for Jacobi3D on 4k to 32k Cori Phase I cores. The number above each bar represents the total speedup compared to *Basic-MPI*.

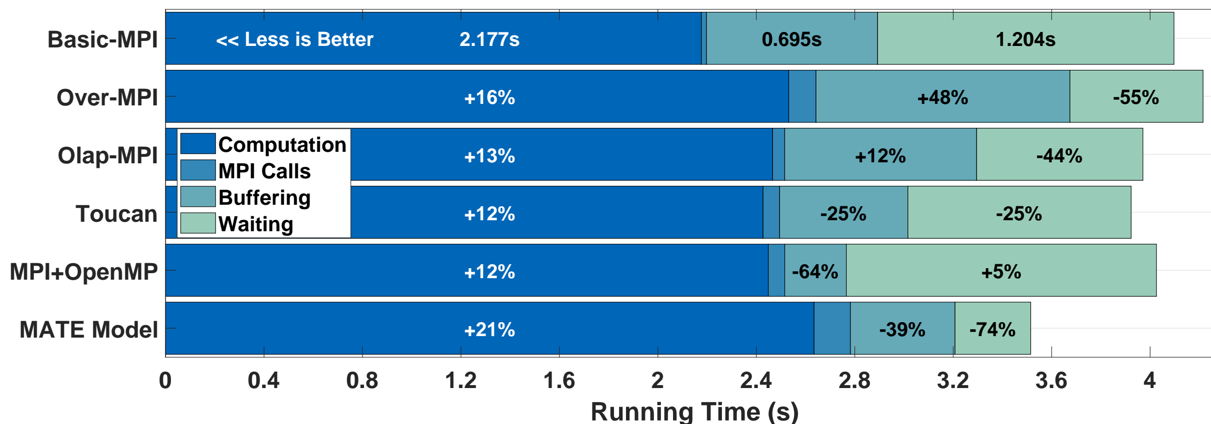

Figure 5.6: Time spent on different phases of our solver on 32k Cori Phase I cores.

MATE obtained outperformed all the other variants at all scales, yielding a 1.17x speedup at 32768 cores, compared to Basic-MPI. We observed similar results also at smaller scales.

To evaluate the particular effects on performance of each variant, we analyzed the time spent by each variant on the different operations of the solver on 1024 Cori Phase I nodes (32768 cores). Fig. 5.6 shows these results where: *Computation* represents the solver kernel time only; *MPI Calls* is the time spent issuing *MPI_Irecv/MPI_Isend* requests; *Buffering* is the time spent packing and unpacking message buffers and *Waiting* is the time spent waiting on network communication to complete. We report the wallclock time (seconds) taken per operation for the *Basic-MPI* variant, and the % of time change of each operation for the other variants.

Results show that MATE was able to reduce a large amount of the network communication (74%) on 32768 cores, compared with Basic-MPI. We observed similar communication reductions at smaller scales. We also see that, while the overdecomposition only (*Over-MPI*) variant reduces 55% of network communication time, its benefits diminish due to a notable increase (48%) in buffering costs and thus fails to produce any speedup. On the other hand, *Olap-MPI* reduced the communication time without a significant impact in its packing costs.

All the variants suffered an increase in computation time over the baseline. We hypothesize this increase comes from a loss in cache efficiency due to lack of prefetching. In the

Basic-MPI variant, the side effect of buffer unpacking is to prefetch grid data before executing the kernel, which will help speed it up. In turn, the kernel smoother will improve cache locality of buffer packing. To test this hypothesis, we developed a variant of Basic-MPI in which we remove its buffering operations (this variant does not produce a correct result).

We used the *Performance Application Programming Interface* (PAPI) [86] to activate hardware counters in the Haswell processor during the main loop of the solver. Table 5.1 compares the computation time of the Basic-MPI variant (with and without buffering operations), along with their absolute number of L2 cache misses and miss rate on a 100-iteration 128-node Cori Haswell I run. We also include results for the MPI+OpenMP and MATE variants.

The table shows that, by removing the buffering operation from Basic-MPI, we observe an increase in computation time (1.12x) that can be attributed to higher cache miss counts (1.10x). A slightly smaller increase in L2 cache miss count (1.09x) can be observed on MPI+OpenMP, which correlates with its slightly lesser impact on computational time (1.08x). The MATE variant suffers the highest L2 miss count (1.26x), which explains why it achieves the worse computation time of all (1.17x). We attribute this effect to a disruption of cache locality from the use of overdecomposition, where ranks switch more often, reloading their data back into cache lines and producing an excess in cache misses.

These results support our hypothesis that increased cache misses produced the slower computation times. Notably, the excess cache miss count did not come from an increased cache miss ratio, as all variants achieved relatively similar results, but from an increase in the total

Table 5.1: Comparison of computation times and L2 cache misses on 2k cores.

Variant	Compute Time	L2-D Misses	L2-D Accesses	Miss Rate
Basic-MPI	3.95s (1.00x)	2.98×10^{11} (1.00x)	8.37×10^{11}	35.6%
Basic-MPI (No Buf.)	4.35s (1.10x)	3.33×10^{11} (1.12x)	8.68×10^{11}	38.3%
MPI+OpenMP	4.28s (1.08x)	3.25×10^{11} (1.09x)	10.57×10^{11}	30.7%
MATE	4.65s (1.17x)	3.76×10^{11} (1.26x)	11.34×10^{11}	33.1%

number of L2 accesses.

Threading variants suffer from an additional cost of MPI call overheads. In MATE, workers serialize the injection of MPI messages due to threading concurrency limitations in the Cray-MPI library (Appendix C), producing periods of busy-waiting that extend the rank's occupancy in the core. The *MPI+OpenMP* variant suffers from this limitation as well as it also relies on multi-threading but to a lesser extent since it does not employ overdecomposition and thus exchanges fewer messages than MATE.

We observed that buffer packing cost in *MPI+OpenMP* shrinks by a more significant amount than in MATE: 64% vs. 39%. However, as noted above, this variant cannot to reduce communication costs as the overlapping variants did and so obtained little improvement in performance. These results demonstrate that MATE can improve performance by hiding communication, even though there is a loss of cache locality in computation and message serialization.

To gain insight into why the MATE variant was able to hide a large portion of the cost of network communication, we plotted the activity of 8 cores during execution of a short (30 iteration) run on 1024 nodes (32768 Haswell cores) of the Cori Phase I platform. Fig. 5.7 shows core usage timelines of the (*top*) Basic MPI and (*bottom*) MATE variants. These timelines show how cores fluctuate between busy time[3] (dark blue) and idle time (white).

Basic-MPI assigns every MPI rank to its MPI process, mapped to a single core. This variant cannot perform any useful work while ranks are waiting, and thus it suffers from the full cost of communication. On the other hand, the MATE variant keeps cores busy by executing ranks while others communicate. Since a MATE process manages a pool of 64 ranks running on 8 threads, it maximizes opportunities for overlap by allowing ranks to execute on any core. Fig. 5.8 shows half a second of execution of local rank 0, exclusively. We can see how the rank migrates (dashed line) across all the threads in the MATE process, starting execution as soon as it becomes ready.

[3]This includes time spent in computation, packing, and unpacking operations.

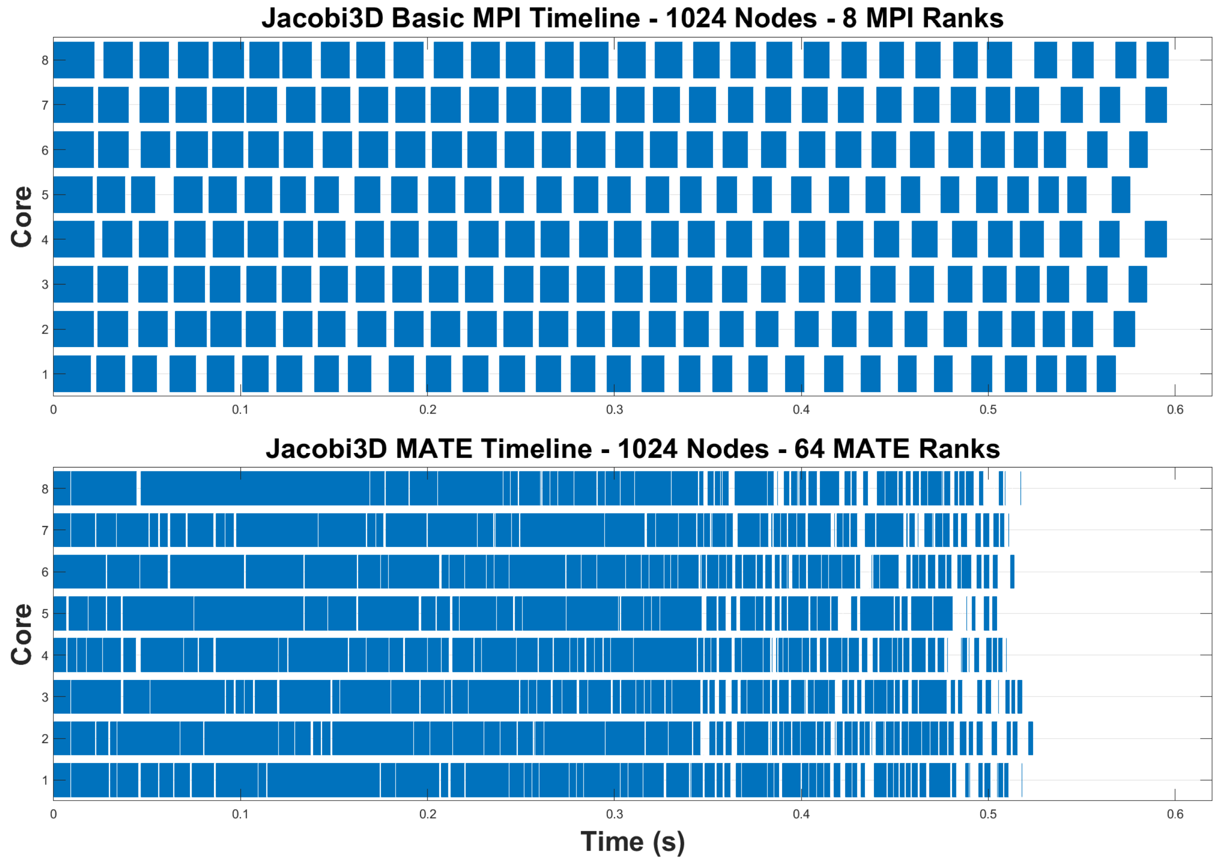

Figure 5.7: Core Timelines. (Top) Basic MPI (8 Ranks), (Bottom) MATE (64 Ranks).

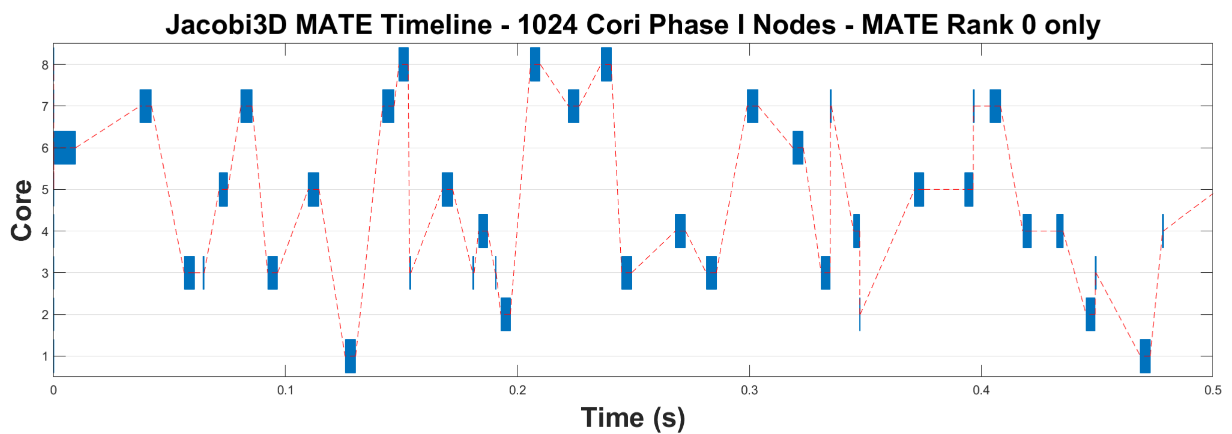

Figure 5.8: Timeline of local rank 0 transitioning across the 8 cores in the MATE process.

5.2.2 Cori Phase II (KNL)

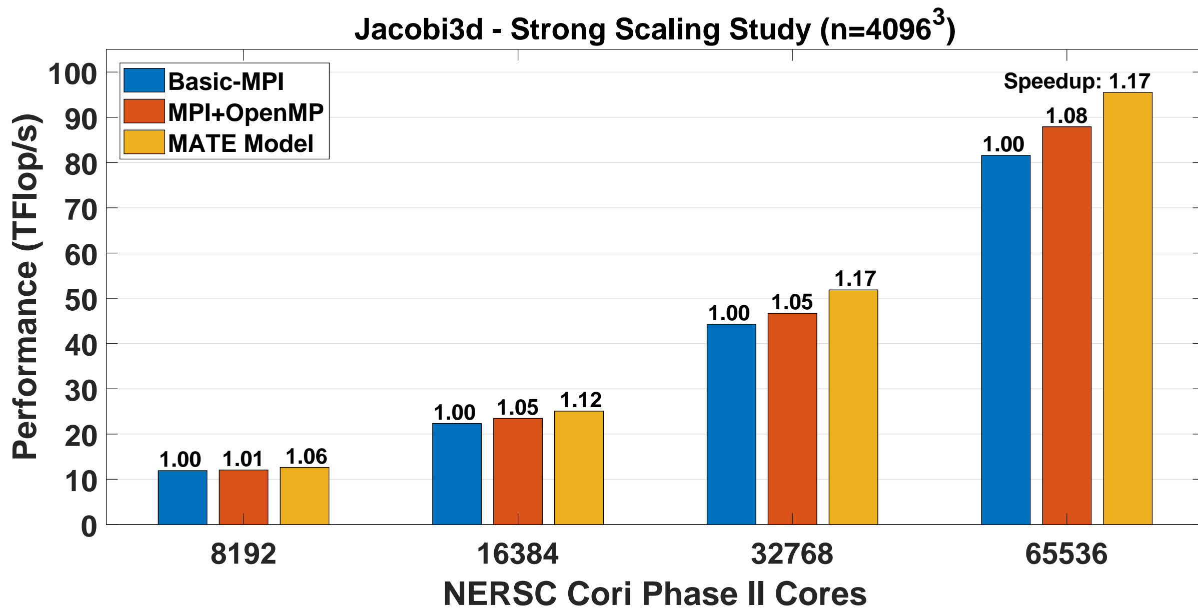

Figure 5.9: Strong Scaling results for Jacobi3D on 8k to 64k Cori Phase II cores.

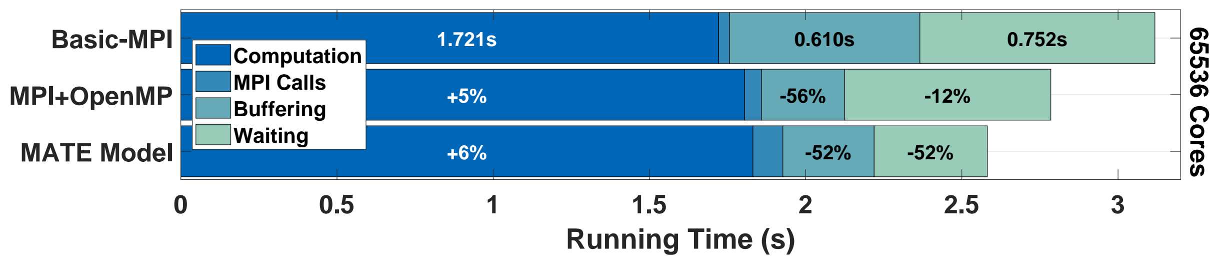

Figure 5.10: Time spent on different phases our solver on 64k Cori Phase II cores.

For the Cori Phase II platform, we ran three variants: *Basic-MPI*, *MPI+OpenMP*, and *MATE*. The purpose of this experiment is to demonstrate that our model can benefit performance on different computing platforms and, therefore, we donot include as many variants as on Cori PhaseI. Besides MPI and MATE, we also include MPI+OpenMP to give a comparison point for intra-node data motion reduction potential.

We performed a strong scaling study over a range of 128 to 1024 nodes (8192 to 65536 KNL cores), duplicating the number of nodes at each scaling step. We performed 200 solver iterations on a grid of $n = 4096^3$ cells. As we did for Cori Phase I, we used 4096 cells per side

since this number is divisible by 64, the per-side distribution of processes used in the *Basic-MPI* variant on 32768 cores. On 1024 nodes, the grid occupies 1.04% of the total system's memory.

We run the *Basic-MPI* variant using one MPI process per core and an on-node rank distribution of $4 \times 4 \times 4$. We determined that the best performing configuration for the *MPI+OpenMP* variant was 8 MPI processes $\times$ 8 OpenMP threads on each node with a local thread distribution of $1 \times 4 \times 2$ on each MPI process. For the *MATE* variant, we determined that the best configuration uses 16 MATE processes $\times$ 4 threads per node, with 16 MATE ranks per process with a local rank distribution of $1 \times 4 \times 4$, corresponding to an overdecomposition factor of 4.

Fig. 5.9 shows the results of our study. We performed five runs of each (variant, node count) combination and found less than $\leq 1\%$ variation in wallclock time between runs. We report the best run for each combination. Fig. 5.10 shows the per-operation running time breakdown for the largest run on Cori Phase II (65536 cores).

Table 5.2: Ratio of computation, buffering, and communication costs on 1024 nodes of Cori Phase I (32k cores) vs Cori Phase II (64k cores) for the Basic-MPI variant.

Basic-MPI			
System	Compute %	Buffering %	Network %
Cori Phase I	53%	17%	30%
Cori Phase II	55%	21%	24%

Table 5.3: Ratio of computation, buffering, and communication costs on 1024 nodes of Cori Phase I (32k cores) vs Cori Phase II (64k cores) for the MATE variant.

MATE			
System	Compute %	Buffering %	Network %
Cori Phase I	79%	12%	9%
Cori Phase II	75%	11%	14%

Table 5.2 shows a comparison of the running times of Basic-MPI on 1024 nodes of both Cori Phase I and Phase II systems. In this comparison, we solve the same problem ($n = 4096^3$, 200 iterations) in both systems and use the same number of nodes. We observe that both

systems have a similar computation ratio ($53 \sim 55\%$), although they differ in their sources of communication cost. The Phase I system suffers from a higher communication cost (30%) than Phase II (24%), which can be explained by their significant difference in global peak bisection bandwidth (5.65 Tb/s, in Phase I vs. 45Tb/s in Phase II) in their interconnects. On the other hand, Phase II suffers from a higher (21%) component of buffering, compared to Phase I (17%). This difference can be explained by the fact that Phase II exchanges more internal boundaries in each node contains an Intel '*Knights Landing*' processor with 64 cores vs. two 16-core Intel '*Haswell*' processors in Phase I.

Table 5.3 shows a similar comparison for the MATE variant. As we observed before, MATE achieved better performance in both systems compared to Basic-MPI. On both systems, MATE achieves a much higher ($75 \sim 79\%$) computation ratio (*i.e.*, effective core usage), reducing the buffering cost ratio to ($11 \sim 12\%$), even on Cori Phase II, which contains a higher internal-node boundary than Phase I. MATE also reduced the network communication ratio to a single digit in Phase I, demonstrating that MATE's mechanisms can significantly reduce this cost, even in interconnects with a with relatively low peak bandwidth.

5.3 Summary

Our results show that MATE was able to reduce a significant amount of the intra-node and network communication overheads of a single-level structured grid method. At the largest scale experiment on Cori Phase I (32768 cores), MATE was able to reduce 38% of the intra-node data motion and 75% of the network communication costs. These savings yielded a total 1.17x speedup, compared to Basic-MPI and a 1.15x speedup, compared to MPI+OpenMP. We observed similar benefits from MATE at smaller scale experiments as well.

We have also shown that MATE was effective in reducing a large portion of the cost of communication on Cori Phase II. On this system, we observed a reduction of 52% in both

buffering and network communication costs, yielding a speedup of 1.17x, compared to Basic-MPI, and 1.08x speedup, compared to MPI+OpenMP.

These results demonstrate that, by reordering communication and computation, MATE obtains a better performance than a hand-overlapped MPI variant, even though there is a loss of cache locality in computation and message serialization. Additionally, our results with Cori Phase I serve to show that MATE can significantly reduce the cost of network communication in low-bandwidth interconnects, while our results with Cori Phase II show that MATE can reduce the cost of intra-node data motion in highly-threaded processors, such as Intel 'Knights-Landing'.

Acknowledgements

This chapter is, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

Chapter 6

Test Case II: Cannon's Algorithm

6.1 Overview

Linear algebra algorithms play an important role in scientific computing as they are part of a wide range of applications. High-performance scientific distributed math libraries such as ScaLAPACK [24] and PETSc [14], contain highly optimized implementations of these algorithms.

Here we focus on dense matrix multiplication which requires $O(n^3)$ arithmetic operations[1]. Since well-optimized algorithms load $O(n^2)$ elements from memory, they tend to be computationally-bound.

For our next study, we use *Cannon's* algorithm [19] (*Cannon2D*), a parallel algorithm for computing the product of two dense square matrices $C = A \times B$. We introduce the baseline algorithm and a manually overlapping variant and describe our results using the MATE model. Finally, we evaluate performance on our two computational testbeds.

[1] Although algorithms with lower complexity have been proposed [41, 80].

6.2 Code Variants

To evaluate the benefits of applying MATE to Cannon2D, we test three variants:

1. *Basic-MPI* is the baseline for measuring the communication-reduction potential of the other variants. This variant implements Cannon's algorithm but makes no effort in reducing the costs of communication through overlap.

2. *Olap-MPI* is a latency-tolerant variant of the baseline MPI algorithm that employs a hand-coded communication/computation overlap strategy to reduce the cost of network communication.

3. The *MATE* variant employs all the mechanisms described in chapter 3 to reduce the cost of network and intra-node communication.

6.2.1 Base MPI Algorithm

Cannon's algorithm computes the product of two square $N \times N$ matrices. The baseline MPI implementation (*Basic-MPI*) divides the input A and B matrices into smaller square matrices and distributes them among a square number of ranks across the system. The algorithm computes $C = A \times B$ in a series of p steps, where p^2 is the number of ranks. The algorithm requires that p divides N evenly. Each rank owns a square sub-block of C and also holds local sub-blocks of A and B. At each step, ranks shift sub-blocks A and B along rows and columns of a 2D processor geometry and compute a partial matrix product to update the local portion of C ($C \mathrel{+}= A \times B$). Fig. 6.1 shows how the baseline algorithm performs submatrix shifts.

Fig. 6.2 shows the pseudocode of the baseline MPI algorithm. The code obtains the rank identifier *myRank* and the total number of ranks *nRanks* (*lines 3-4*) and calculates the number of ranks per side as $p = \sqrt{nRanks}$ (*line 6*). Next, it calculates the position of its submatrices along

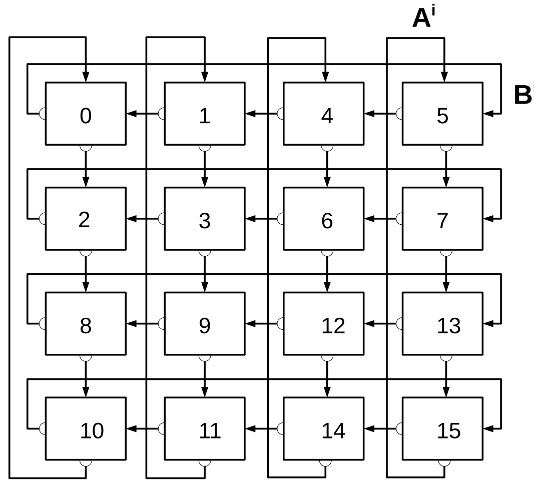

Figure 6.1: Baseline Cannon2D algorithm where ranks shift the A and B submatrices along rows and columns of the processor geometry, in a ring topology.

the *Y* axis by dividing its rank identifier by the number of ranks per side (*p*) (*line 7*). Ranks calculate their position on the *X* axis using the remainder function (*line 8)*.

We use a mapping optimization similar to the one described in Appendix B.3. This optimization reduces the volume of network communication by assigning a rectangular rank-to-core mapping[2] maximizing locality among the cores of a node and optimizing row and column shifts, as opposed to linear mapping which only benefits from row shifts. This mapping is crucial for optimizing the communication performance of the baseline algorithm.

Each rank sends its current step's *A* submatrix to the nearest neighbor located in the row below within the same column, and communication wraps around to the top row if the current rank is in the bottom row. Similarly, each rank sends its *B* submatrix to its left neighbor and, if in the leftmost column, communication wraps around to the rightmost rank. The *modulo* operation[3] is used to determine the id of its (left and up) neighbors (*lines 10-13*).

Ranks determine the number of elements per side of their submatrices (*n*) by dividing the number of global elements per side (*N*) by the number of ranks per side (*p*) (*line 15*). The rank's

[2]To simplify our description of the code, we do not include this optimization in Fig. 6.2.
[3]A better approach is to use row/column specific MPI groups to define the geometry. Our current implementation of MATE, however, does not yet support this feature. Nevertheless, this choice of implementation does not affect performance, and is purely a programming issue.

```
1  Cannon2D(N)
2  {
3   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
4   MPI_Comm_size(MPI_COMM_WORLD, &nRanks);
5
6   int p = sqrt(nRanks); // Uses a square 2D decomposition
7   myRow = myRank / p;
8   myCol = myRank % p;
9
10  int rA = mod(myCol + myRow, p); // Rank to receive A^i from
11  int rB = mod(myRow + myCol, p); // Rank to receive B^i from
12  int sA = mod(myCol - myRow, p); // Rank to send A^(i-1) to
13  int sB = mod(myRow - myCol, p); // Rank to send B^(i-1) to
14
15  int n = N/p; // Elements per submatrix side
16  initialize(A^0, B^0, C, n*n);
17
18  for(int i = 1; i <= p; i++)
19  {
20   MPI_Isend(n*n, A^(i-1) → sA);  MPI_Irecv(n*n, A^i ← rA);
21   MPI_Isend(n*n, B^(i-1) → sB);  MPI_Irecv(n*n, B^i ← rB);
22   MPI_Waitall(requests);
23   dgemm(C += A^i × B^i);
24  }
25
26  verify(C);
27 }
```

Figure 6.2: MPI pseudo-code of Cannon2D's solver.

A^0 and B^0 submatrices (where 0 indicates the first iteration) are initialized using a predetermined formula that provide a fast way to verify the correctness of the result, while the C is initialized to zero (*line 16*).

Within the outer loop (*lines 18-23*), each rank initiates submatrix shifts, sending n^2 elements that contain the current values of its A^i and B^i matrices (*i.e:* A^{i-1}, B^{i-1}) and receiving the new values (*i.e:* A^i and B^i) from its neighbors, and waiting until the exchanges finish (*line 22*). Once the incoming submatrices arrive, the local C submatrix is updated with the product of the local $A^i \times B^i$ submatrices via *dgemm*[4], a heavily optimized double-precision single-core dense matrix multiplication library function. The complete product of the rank's C is obtained after performing p steps; that is, after the A and B submatrices have fully circulated around their respective communication rings.

[4]We use the implementation of *dgemm* provided in the Intel's *Math Kernel Library* [93].

To verify that the algorithm produces a correct result, we use a fast verification algorithm (*line 26*). This algorithm requires input matrices A and B for which the product can be readily computed in closed form. These matrices are described by the formula Eq. 6.1, and the product is calculated by the formula in Eq. 6.2.

$$A_{i,j} = B_{i,j} = \frac{1}{i+j} \quad \forall \quad i = 1..n,\ j = 1..n \tag{6.1}$$

$$C_{i,j} = \sum_{k=1}^{n} \frac{1}{(i+k)\times(j+k)} \quad \forall \quad i = 1..n,\ j = 1..n \tag{6.2}$$

If at least one value in the matrix C differs from the value calculated by this formula by more than 10^{-12}, the program will output an error message and exit. Since every rank can perform this calculation for their particular submatrices, verification can be carried on in parallel. However, since this operation also has a $O(n^3)$ complexity, it may still take a long time for large matrices. We performed this verification once for all our variants at all scales to make sure the optimizations we applied did not affect the final result.

6.2.2 Overlapping MPI Algorithm

The communication/computation overlapping variant (*Olap-MPI*) re-structures the main solver to pipeline the work, computing the current partial product of C while advancing the communication for the next step. Fig. 6.3 shows how a rank sends the current A^i and B^i submatrices to its down and left neighbors, while receiving the next iteration's A^{i+1} and B^{i+1} from its up-/right neighbors. While these exchanges take place, the partial product of C is updated with the result of $A^i \times B^i$. Once the exchanges finish, the rank swaps the pointers of the current and next iteration's submatrices for the next iteration.

Fig. 6.4 shows the pseudo-code for *Olap-MPI*. Each rank receives the values of A^1 and B^1 submatrices from its neighbors (*lines 4-6*) and will not move on until the exchange has

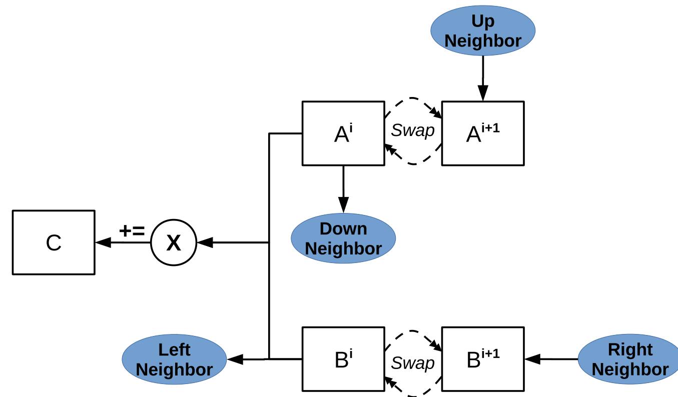

Figure 6.3: A rank achieves overlap by receiving A^{i+1} and B^{i+1} submatrices for the next step while updating the value of C with $A^i \times B^i$ in the current step.

```
for(int i = 1; i <= p; i++)
{
 if (i == 1)
 {    MPI_Isend(n*n, A^0 → sA);   MPI_Irecv(n*n, A^1 ← rA);
      MPI_Isend(n*n, B^0 → sB);   MPI_Irecv(n*n, B^1 ← rB);
      MPI_Waitall(requests);     }

 if (i < p)
 {    MPI_Isend(n*n, A^i → sA);   MPI_Irecv(n*n, A^(i+1) ← rA);
      MPI_Isend(n*n, B^i → sB);   MPI_Irecv(n*n, B^(i+1) ← rB);    }

 dgemm(C  +=  A^i × B^i);
 if (i < p) MPI_Waitall(requests);
 swap(&A^i ↔ &A^(i+1));   swap(&B^i ↔ &B^(i+1));
}
```

Figure 6.4: MPI pseudo-code of overlapping Cannon2D's solver.

finished. This operation is only performed during the first iteration to obtain the initial values for the A and B submatrices. Next, the code initiates communication requests to obtain the values of the next step's A^{i+1} and B^{i+1} submatrices (*lines 9-10*) and updates the current value of C (*line 12*), overlapping communication and computation. Once it finishes updating C, it waits for the completion of the communication requests (*line 13*). After the communication exchange has finished, the current and next iteration pointers to A and B are swapped (*line 14*). Since the rank already obtained its A^i and B^i submatrices, it requires no communication in the last iteration ($i == p$).

6.2.3 MATE Variant

The *MATE* variant introduces a hierarchical approach by applying Cannon's algorithm at two levels at every iteration, once across MATE processes and once among the local ranks within each process. Each process divides the input matrices into p_0 level 0 submatrices, and then subdivides these submatrices into p_1 level 1 submatrices, where p_0 is the number of processes per side and p_1 is the number of local ranks per side. Each local rank calculates a level 1 matrix multiplication, then rotates shared pointers among local neighboring ranks residing in the same level 0 submatrix, avoiding intra-process data movement. This step is repeated p_1 times until the current iteration's value of C at a process-level completes.

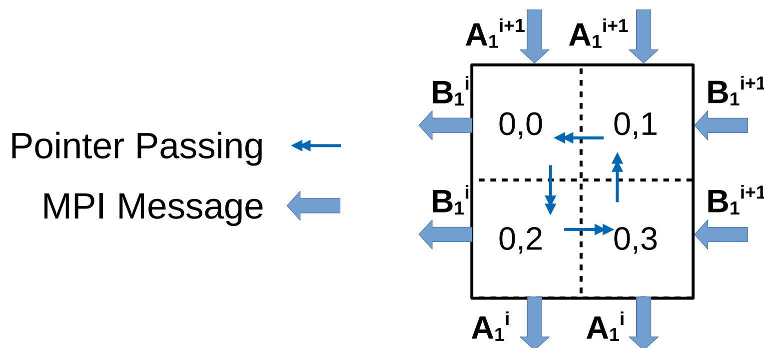

Figure 6.5: Communication in the MATE variant of the Cannon's solver using a hierarchical decomposition. We simplified this figure to show MPI messages only across boundary ranks. However, every rank exchanges MPI messages with neighboring processes.

For each iteration i, every level 1 rank sends its part of the current iteration's A_1^i and B_1^i submatrices, while receiving the next iteration's level 1 A_1^{i+1} and B_1^{i+1} submatrices from neighboring processes, as shown in Fig.6.5. This approach uses a similar communication/computation overlapping strategy as *Olap-MPI* as it receives the matrices for the next iteration while computing the current iteration.

Fig. 6.6 shows the pseudo-code of our MATE implementation. Ranks obtain their process and local identifier and counts at *lines 3-6*. Next, they determine the row (*pRow*) and column (*pCol*) position of their containing MATE process's submatrix in the 2D process topology (*lines*

```
1  Cannon2D_MATE(N)
2  {
3   Mate_local_rank_id(&myLocalId);
4   Mate_global_process_id(&myProcessId);
5   Mate_local_rank_count(&localCount);
6   Mate_global_process_count(&processCount);
7
8   int p_0 = sqrt(nRanks); // Uses a square 2D decomposition
9   pRow = myProcessId / p_0; // MATE Process' Row
10  pCol = myProcessId % p_0; // MATE Process' Column
11  int p_1 = sqrt(p_0); // Uses a square 2D local decomposition
12  y = myLocalId / p_1; // Local Rank's Row
13  x = myLocalId % p_1; // Local Rank's Column
14
15  rA = mod(x + y, p_0); // Remote rank to receive A_1^{i+1} from
16  rB = mod(y + x, p_0); // Remote rank to receive B_1^{i+1} from
17  sA = mod(y - x, p_0); // Remote rank to send A_1^i to
18  sB = mod(x - y, p_0); // Remote rank to send B_1^i to
19
20  for (int i = 0; i < p_1; i++) Mate_AddLocalNeighbor(y*p_1 + i); // Neighbor in same row
21  for (int i = 0; i < p_1; i++) Mate_AddLocalNeighbor(p_1*i + z); // Neighbor in same col
22
23  n = N/p_0;  // Elements per process-wide submatrix side
24  nl = n/p_1; // Elements per local sub-submatrix side
25  k = (x+y)%p_1; // Initial submatrix pointer position for local multiplication
26  if (myLocalId == 0) initialize(A_0^0, B_0^0, C_{0}, n); // Initializing Level 0 Submatrix
27  Mate_LocalBcast(&A); Mate_LocalBcast(&B); Mate_LocalBcast(&C);
28
29  #pragma mate graph
30  for(int i = 0; i < p_0; i++)
31  {
32   #pragma mate region(communicate) depends (update*@)
33   {
34    MPI_Isend(nl*nl, A_1^i → sA); MPI_Irecv(nl*nl, A_1^{i+1} ← rA);
35    MPI_Isend(nl*nl, B_1^i → sB); MPI_Irecv(nl*nl, B_1^{i+1} ← rB);
36   }
37
38   #pragma mate region(compute) depends (update*@)
39   for(int j = 0; j < p_1; j++)
40   {
41     dgemm(C_1  += A_1^i × B_1^i);
42     k++; if (k == p_1) k = 0; // Rotating Local Submatrix Pointer Positions
43   }
44
45   #pragma mate region(update) depends (compute@, communicate)
46   swap(&A_1^i ↔ &A_1^{i+1});  swap(&B_1^i ↔ &B_1^{i+1}); // Local Pointer swap
47  }
48
49  verify(C);
50 }
```

Figure 6.6: Pseudo-code of Cannon's algorithm enhanced with MATE annotations and calls.

8-10). Since local ranks apply Cannon's algorithm at a local level as well, they also calculate the number of local ranks per side (p_1) and the row (y) and column (x) position of their local submatrix (*lines 11-13*) and determine the ids of their send/receive remote neighbors for their A_1 and B_1 submatrices (sA, rA, sB, rB, respectively)

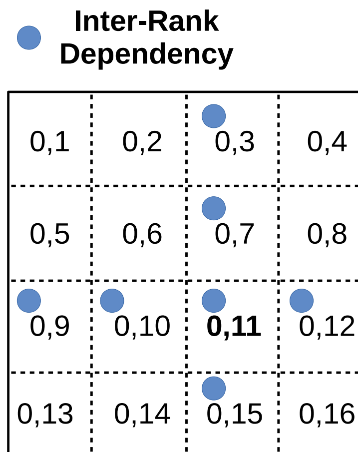

Figure 6.7: Rank (0,11) declares inter-rank dependencies only along its same column/row.

Each ranks declares inter-rank dependencies with all the local ranks in the same row or column (*lines 20-21*). These dependencies guarantee that a local rank will have all the necessary local submatrices ready for access when performing the current step's update. Local ranks that do not reside in the same row or column need not synchronize since they will not exchange submatrix pointers among each other. Fig. 6.7 shows an example of the inter-ranks dependencies declared by rank 11, belonging to a process 0 which contains 16 ranks. Rank 11 only waits for ranks 3, 7, 11, 15, 9, 10, 12 (and itself) to finish updating their pointers in the current iteration before initiating communication and computation operations in the next iteration.

Lines 23 and 24 compute the number of elements per side for the level 0 and level 1 submatrices and establish the k starting row/column in the level 1 Cannon's algorithm execution (*line 25*). Only local rank 0 allocates and initializes the process-wide A, B, and C submatrices and then distributes their pointers to the other local ranks (*line 26*). Each local rank computes only their part of the process-wide C submatrix.

The main loop of the solver (*lines 29-47*) defines three regions. In the *communicate* region (*lines 32-36*), ranks initiate the exchange of their part of A and B with remote neighbors. Each remote message contains nl^2 elements, corresponding to the rank's level 1 section of the process-wide submatrix.

The *compute* region (*lines 38-43*), updates the C submatrix. This update applies Cannon's algorithm at a local level by updating the value of $C_{x,y}$, p_1 times. At the first step, each rank calculates the value of the product of $A_{(x,k)} \times B_{(k,y)}$ (*lines 41*). Then, ranks increment the value of k, setting it back to *0* if it reaches p_1. Every time ranks change the value of k, they change their local values of the A and B pointers to those owned by neighbor local ranks in the same row (for A) and column (for B). At the end of this procedure, C contains the product of $A_0 \times B_0$ consistent with the conventional Cannon's algorithm operation.

The *update* region (*lines 45-46*), updates the local pointers of A and B to contain the values of the incoming submatrices for the next iteration. This region executes after completing the exchange of remote submatrices, and all the local ranks in the same row and column (including itself) have finished computing the current iteration.

6.3 Size Scaling Study

We performed a size scaling study to show that MATE's realizes communication and computation overlap over a wide range of matrix sizes. We evaluate the performance of our three variants on increasing values of N while keeping the number of cores fixed. For this experiment, we used 128 Cori Phase I nodes (4096 Haswell cores) and matrix sizes starting from $N = 12288$ to $N = 40960$. We employed a 4×4 rank-node mapping for all our variants. We run the *Basic-MPI* and *Olap-MPI* variants using one MPI process per core. For the *MATE* variant, we determined that the best configuration uses 8 MATE processes $\times$ 4 threads per node, with 64 MATE ranks per process. This configuration represents an overdecomposition factor of

4. We performed three, 40 iteration-runs of the main solver to reduce noise and variation in our results.

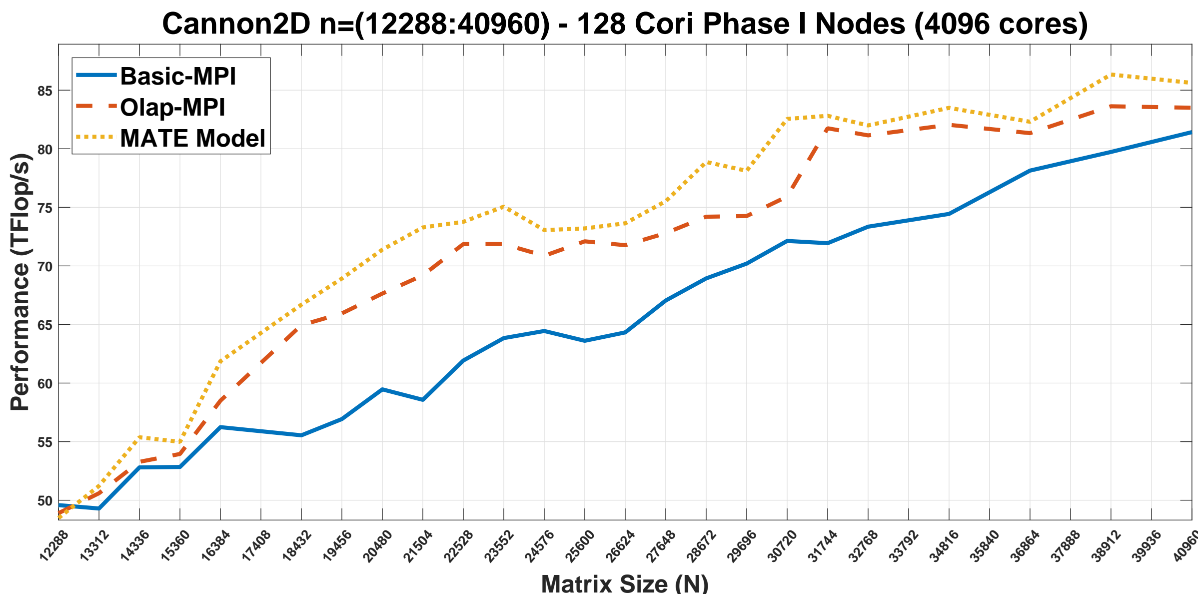

Figure 6.8: Matrix Size Scaling of our three Cannon2D variants on 4k Cori Phase I nodes.

Fig. 6.8 shows the results of our study. The *MATE* variant consistently outperforms the *Olap-MPI* variant. We attribute this result to to two factors. First, *MATE* does not perform communication among ranks in the same process, but instead exchanges pointers which reduce, in part, the cost of in-node communication. Second, besides the overlapping strategy employed by Olap-MPI, *MATE* also employs overdecomposition which increases potential for communication with computation overlap.

We also observe that the computational performance of all variants increases with the size of the matrices. This increase is to be expected since, as we scale the size of the matrix, each core has more floating point operations to perform (this scales with $O(n^3)$) than data to communicate (this scales with $O(n^2)$). The relative cost of communication, then, decreases linearly with increasing matrix sizes. This relation explains why the potential benefit of overlapping variants also grow smaller as we approach the largest matrix sizes.

Both communication-reducing variants achieve their peak speedups, compared with *Basic-*

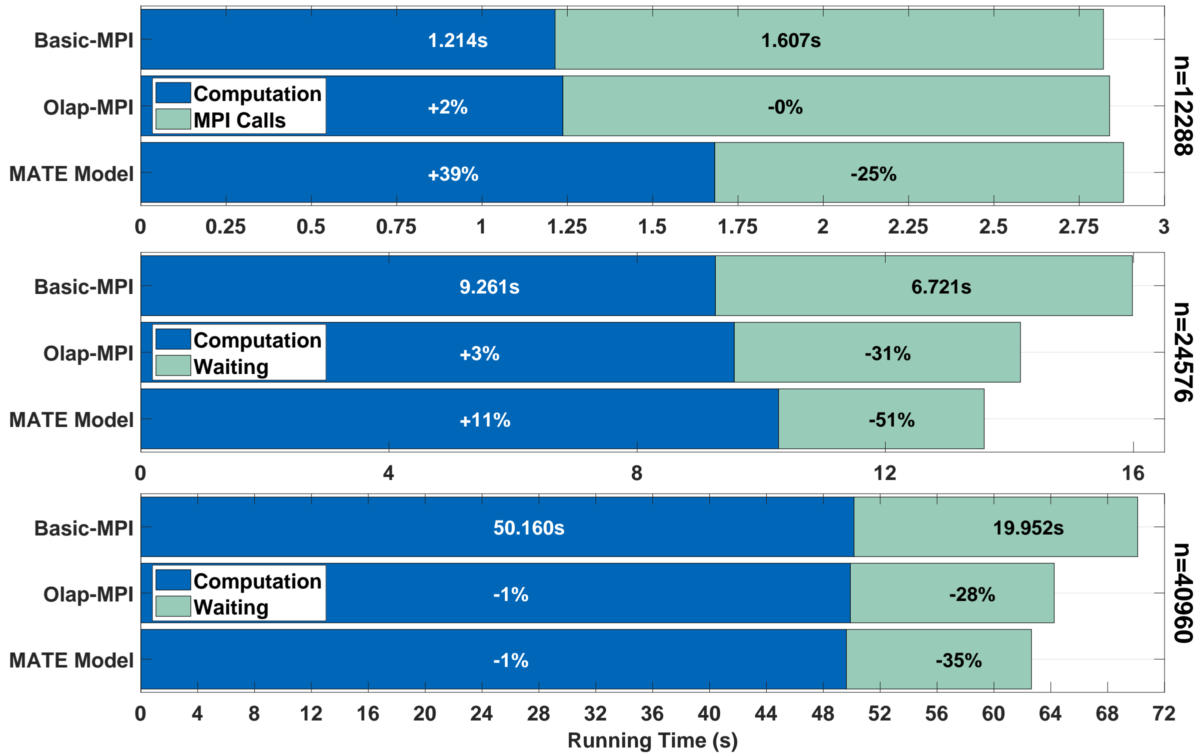

Figure 6.9: Execution breakdown for Cannon2D on 4096 Haswell cores with matrix sizes (top) n=12288, (middle) n=24576, and (bottom) n=40960.

MPI, at matrix sizes in the of the middle range of N. On the other hand, their performance approaches that of *Basic-MPI* for small and large matrices. To investigate the cause of this phenomenon, we measured the wallclock time spent on computation and communication operations at $n = 12288, 24576, 40960$ for *Basic-MPI* variant, and show the % of time change of each operation for the other variants. Fig. 6.9 plots the results.

For $n = 12288$ elements per side, overlapping variants fail to improve the performance of the baseline algorithm. First, the *Olap-MPI* variant has a similar performance to that of the baseline algorithm, which indicates that the overlapping approach does not affect very small matrices. Second, although the *MATE* variant reduces the cost of communication by a quarter, it also introduces additional overhead that offsets the benefit. This overhead comes from issuing 4-times the number of messages since we use an overdecomposition factor of 4. This overhead does not affect the running time for $N = 24576, 40960$ matrices, since its impact is relatively

insignificant.

For $N = 40960$ elements per side, MATE achieves a 35% reduction in network communication with no loss in computational performance. However, we can see that, for the reasons we exposed above, the component of computation in the baseline is higher at this scale. Therefore, communication reduction efforts have less of an impact on performance than in lower ranges.

The middle scale, $N = 24576$, corresponds to a point at which the communication cost is significant enough that its reduction yields a reward, while MATE's mechanisms have a less impactful (11%) effect on computational performance. We can conclude that communication/-computation overlapping strategies are most efficient within a range of matrix sizes large enough not to be dominated by the fixed costs of communication, and small enough not do be dominated by computation costs.

6.4 Weak Scaling Studies

We performed a set of weak scaling experiments to verify that *MATE* reduces the cost of communication at different scales and on different platforms. In a weak scaling study, we keep the amount of work per core fixed as we increase the number of nodes. We have chosen middle-sized matrices as initial inputs in our study to showcase the potential of overlapping approaches[5].

We increase the size of the input matrices as we scale up the number of cores using the formula: $n_{(2p)} \simeq \sqrt[3]{2}n_{(p)}$, where n is the linear dimension of the matrix, and p^2 is the number of cores. We round this value to the nearest square number that is also divisible by the number of cores on the node to distribute submatrices evenly across all ranks.

[5]We do not employ a strong scaling study since the per-core computational decreases by 8x every time we double the number of cores, which brings matrix sizes to either extremes of the range in which we have determined that overlapping strategies have little effect.

6.4.1 Cori Phase I (Haswell)

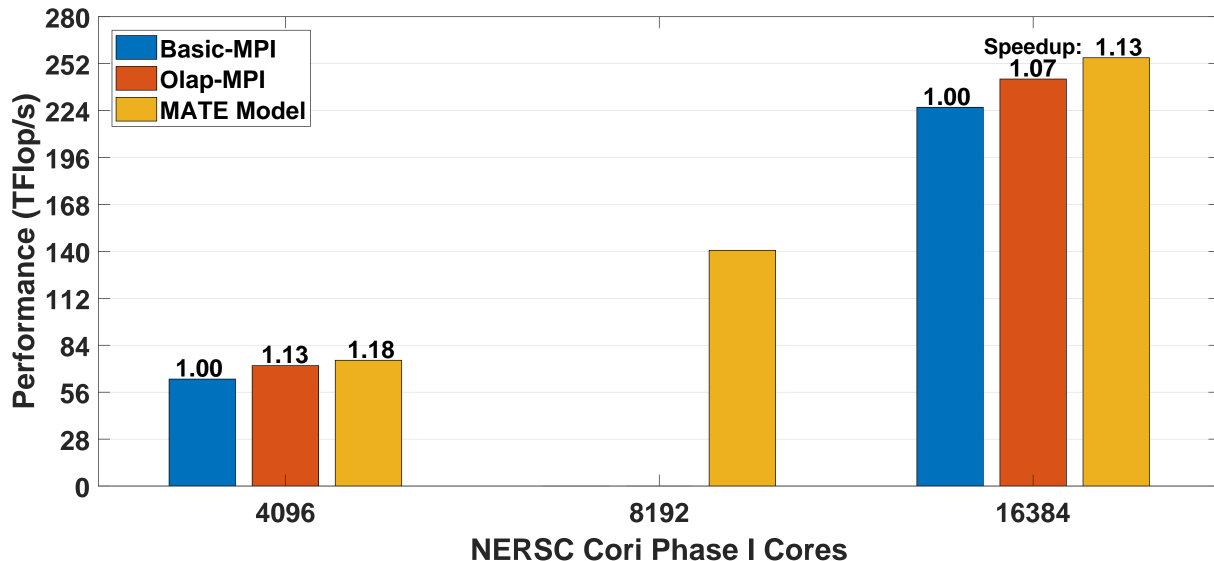

Figure 6.10: Weak Scaling results for Cannon2D on 4k to 16k Cori Phase I cores.

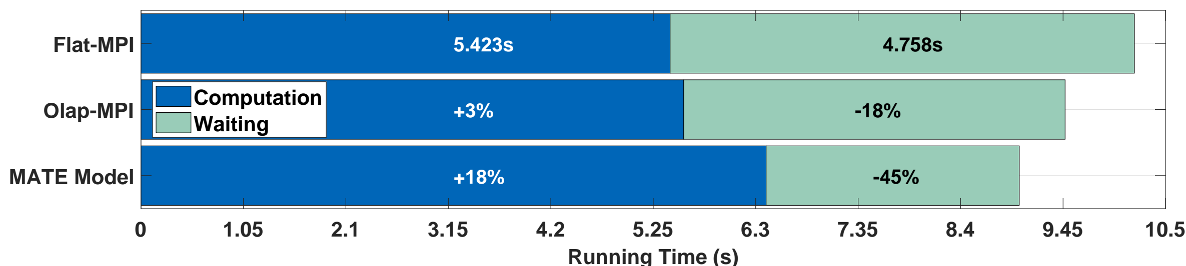

Figure 6.11: Execution breakdown on 16k Cori Phase I cores.

For Cori Phase I, we use an initial matrix size $N = 24576$ and run the multiplication kernel 40 times to reduce system noise. We use the same testbed configuration as in our matrix size scaling experiments. Since Cannon's algorithm requires that the number of ranks be a perfect square, we tested the MPI variants on 4096 and 16384 cores. Running on 8192 cores is not possible with MPI variants since this number is not a perfect square. MATE, on the other hand, is able to run in 8192 nodes as it provides the flexibility of using a different overdecomposition factor as on the other experiments. We use 2 MATE ranks per core on 8192 nodes, and 4 ranks per core on 4096 and 16384 cores. Fig. 6.10 shows the results of our study.

Fig. 6.11 shows the execution breakdown at 512 Cori Phase I nodes. The *Olap-MPI*

variant was able to reduce 18% of the cost of communication while suffering almost no computation time overheads, and thus obtaining a total 1.07x speedup. On the other hand, the *MATE* variant was able to reduce 45%, of the communication cost a net improvement of 27% over the manual overlapping variant. MATE, however, suffered from a 18% increase in the computation time. Despite this added cost, MATE was able to obtain a higher speedup of 1.13x, compared to *Basic-MPI*.

6.4.2 Cori Phase II (KNL)

For Cori Phase II, we conducted a weak scaling study on 4096, 16384, and 65536 cores using an initial matrix size $n = 24576$ and 20 multiplication iterations per run to reduce system noise. We ran the *Basic-MPI* and *Olap-MPI* variants using one MPI process per core and a square rank-node mapping of 8×8. The best configuration for MATE uses 4 processes $\times$ 8 threads per node, with 64 MATE ranks per process and a rank-node mapping of 4×4. This configuration corresponds to an overdecomposition factor of 4.

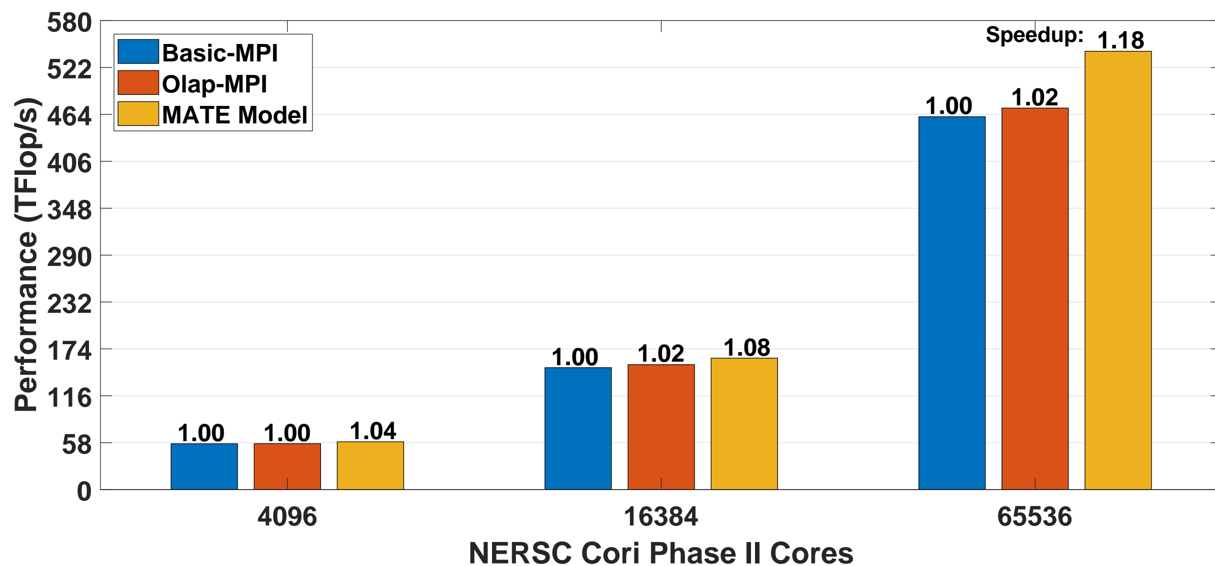

Figure 6.12: Weak Scaling results for Cannon2D on 64, 256, and 1024 Cori Phase II nodes.

Fig. 6.12 shows the results of our study. We can see that the manually overlapping

variant could not improve the performance of the baseline algorithm, while MATE was able to significantly overperform the baseline algorithm on 1024 nodes. To explain the performance disparity between the Olap-MPI and MATE variants, we measured the wallclock time spent in communication and computation operations for all variants on 1024 nodes. Fig. 6.13 shows the result of this analysis. The *Olap-MPI* variant was able to reduce 11% of the cost of communication. However, it also suffered from a 6% computation time increase, offsetting its benefits and obtaining only a 2% improvement in running time. MATE was able to reduce a significant part (48%) of the communication cost, while suffering a 14% computation time increase, obtaining a total 1.18x speedup.

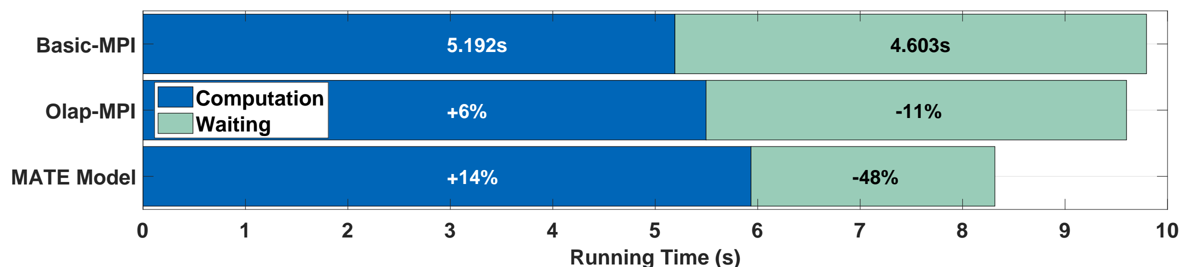

Figure 6.13: Execution breakdown on 64k Cori Phase II cores.

Table 6.1: Ratio of computation and communication on 64, 256, and 1024 nodes for the Basic-MPI and MATE variants.

	Basic-MPI		MATE	
Scale	Compute %	Network Comm. %	Compute %	Network Comm.%
64 Nodes	69%	31%	71%	29%
256 Nodes	65%	35%	72%	28%
1024 Nodes	53%	47%	71%	29%

We also observe that the improvements for the MATE variant increase as we doubled the number of nodes, starting at 4% on 64 nodes, 8% on 256 nodes, and then 18% on 1024 nodes. To understand this effect, Table 6.1 shows the communication and computation ratios of the Basic-MPI and MATE variants on 64, 256, and 1024 nodes. We can see that the MATE variant

achieves a constant ratio of $\sim 71\%$ computation across all scales. If we interpret this ratio as MATE's upper bound for core usage, we see that its speedup is related to the difference of this bound and the computation ratio of the Basic-MPI variant.

At smaller scales (64 and 256 nodes), the Basic-MPI variant achieves a relatively good computation ratio (69% and 65%, respectively), which offers little improvement potential towards MATE's upper bound. On 1024 nodes, however, Basic-MPI's computation time decreases to 53%, allowing MATE to obtain a more significant improvement. We attribute this decrease to a communication cost jump that occurs between 256 and 1024 nodes. Since each cabinet in Cori Phase II contains 192 nodes (appendix A), it is more likely that, on 1024 nodes, many more messages are crossing cabinet boundaries through optical links, causing a noticeable increase in network communication costs.

6.5 Summary

We have shown that the MATE model can improve the performance of dense matrix multiplication on our two supercomputing testbeds. Our matrix size scaling experiment shows that overlapping strategies can obtain speedups over a broad range of matrix sizes, only failing to obtain speedups over the baseline algorithm at small matrix scales, in which the fixed cost of communication dominates the running time, and on large matrix sizes, where the computation costs are the primary cost. Furthermore, this experiment shows that MATE outperforms the manually-overlapping variant at every matrix size.

Through our weak scaling studies, we determined that MATE can reduce almost half of the communication costs at large scales and obtain significant speedups. On 16k cores of the Cori Phase I testbed, MATE was able to reduce 45% of communication costs, yielding a 13% improvement over the baseline algorithm. On 64k cores of the Cori Phase II testbed, MATE reduced 48% of communication costs, yielding a 18% improvement over the baseline algorithm.

Finally, we have shown that overdecomposition enables restrictive algorithms that require a number of ranks that is a perfect square, to run with a non-square number of cores, which is not possible with traditional MPI approaches without underusing the system's resources.

Acknowledgements

This chapter is, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

Chapter 7

Test Case III: Cloverleaf3D

7.1 Overview

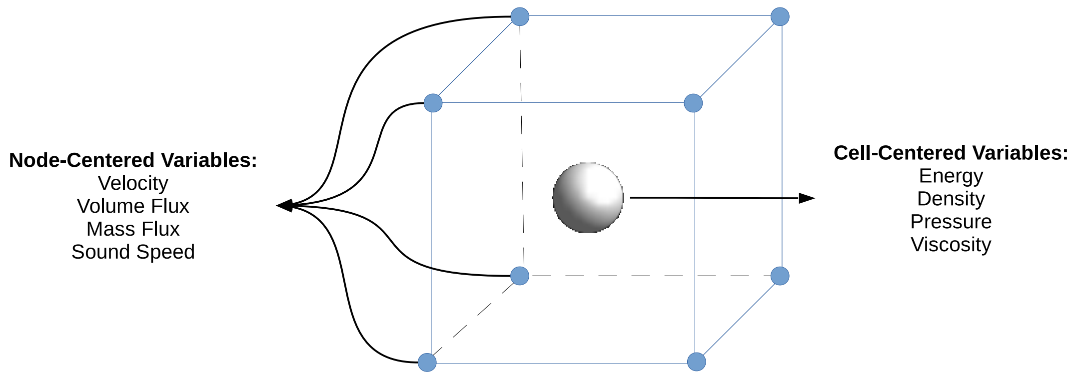

Figure 7.1: CloverLeaf3D's staggered grid with cell and node centric variables.

Cloverleaf3D [96] is the 3D implementation of *Cloverleaf* [64], a Lagrangian-Eulerian hydrodynamics benchmark. Cloverleaf3D solves the compressible Euler equations with a second-order degree of accuracy using an explicit finite-volume method. The solver uses a 3-dimensional staggered grid that divides the space into a set of cells of equal volume and stores multiple field variables at a cell and node-levels, as shown in Fig. 7.1. CloverLeaf3D grids store 14 fields in total: *cell density*, *cell energy*, *cell pressure*, *cell viscosity*, *speed of sound*, *node velocities* (x, y and z), *volume flux* (x, y and z), and *mass flux* (x, y and z).

Given the complexity of this code, it is not reasonable to recode it to employ an overlapping strategy and therefore represents an important test case for MATE. This case shows that MATE can be gainfully applied in applications where it is believed that communication/-computation overlapping will benefit performance at large scales but the user has no means of implementing this strategy themselves.

7.2 Code Variants

7.2.1 Base MPI Algorithm

Cloverleaf3D breaks down computation into six different kernels that sweep over the entire grid and update one or multiple grid field variables, and each kernel uses a different stencil. The *IdealGas* kernel updates the pressure and sound speed of the cell; the *Viscosity* kernel updates the viscosity in each cell's volume; the *PdV* kernel calculates the integral of the pressure with respect to the cell's volume differential and updates the cell's energy and density; the *Flux* kernel calculates the change in volume flux, and; the *Advection* kernel restores the cells to their original position and calculates the mass differential. Cloverleaf3D advances the solution iteratively until it reaches a user-defined end time or desired convergence.

Fig 7.2 shows the stencils and grid fields that each of the kernels read and update during their execution. In between kernel executions, Cloverleaf3D ranks communicate the updated boundary information (*update_halo()*) to their neighbor grids. In total, the solver executes six different kernel calls and boundary exchanges per iteration[1]. Cloverleaf exchanges boundary cells among all the neighbors reachable in 3 Manhattan distances on the 3D grid. In total, each cell exchanges data among its 26 immediate neighbors (six faces, twelve edges, and eight corners).

To exchange boundary information, Cloverleaf3D uses MPI Fortran bindings which have the same input arguments as the C++ bindings. Fig. 7.3 shows how *update_halo()* exchanges

[1]Cloverleaf3D executes some of these kernels repeatedly, performing more than six exchanges per iteration.

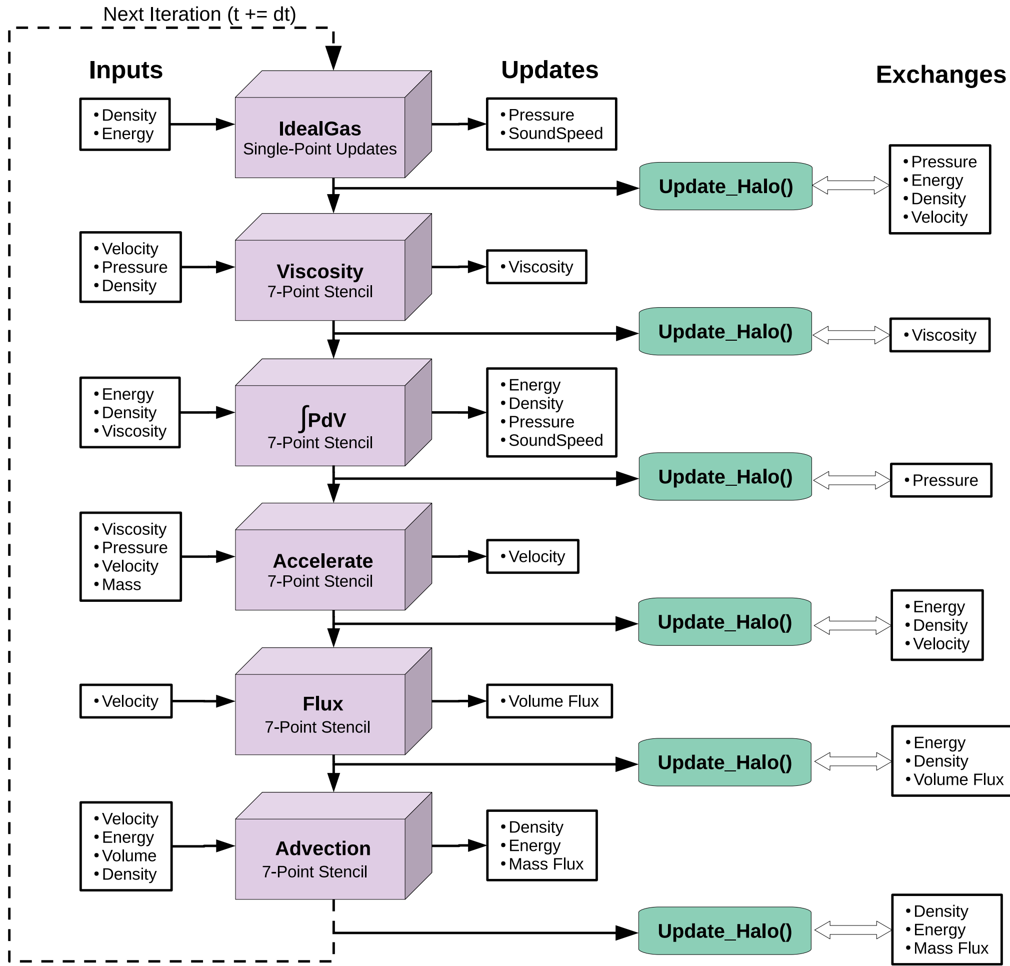

Figure 7.2: CloverLeaf's main solver's kernel and exchange operations.

boundary data in a 3-stage process, allowing ranks to reach their 26 neighbors across 3 dimensions. First, each subgrid communicates boundary faces across the x-axis. After this stage completes, boundary faces across the y-axis are exchanged. Since this second stage includes ghost cells of the x-faces, the exchange of y-faces will include x-edges originally belonging to its neighbors in the x-axis. Finally, ranks exchange faces across the z-axis, which will include the edges in the y-axis from the second step, and the x-corners from the first step. At the end of this procedure, all ranks receive information from neighbors up to a 3-deep Manhattan distance.

Fig. 7.4 shows the pseudocode for *update_halo()*[2]. This routine receives one argument,

[2]Although Cloverleaf is programmed in Fortran, we use a C-like pseudocode to simplify the explanation.

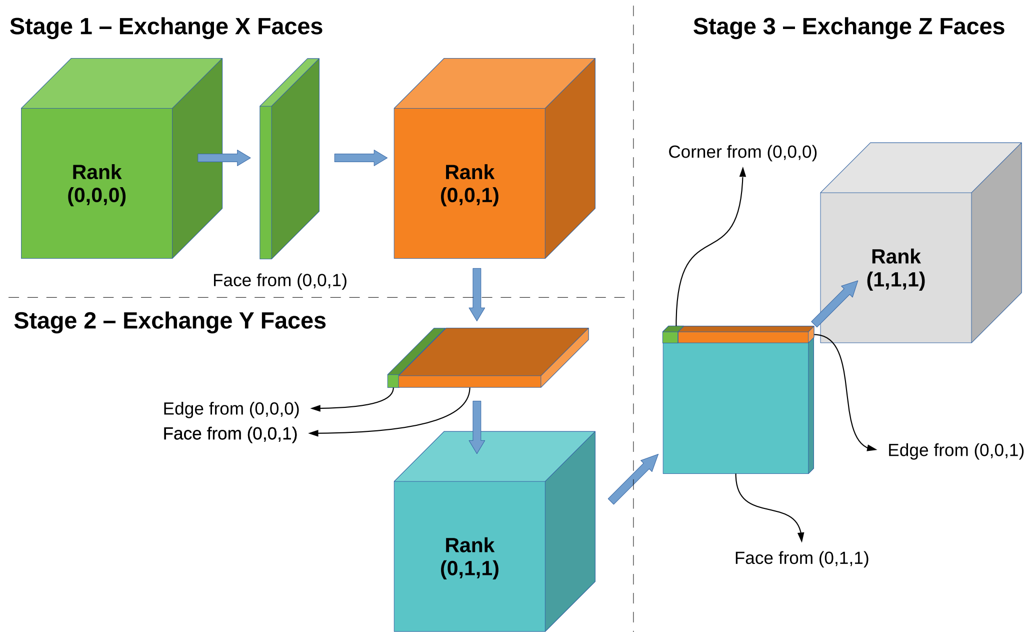

Figure 7.3: CloverLeaf3D's *update_halo()* 3-stage boundary exchange procedure.

fields (*line 1*), an array of boolean values that indicate which of the 14 fields of the grid will be exchanged in this operation, as indicated in Fig. 7.2. Then, *update_halo()* invokes the *exchange_faces()* routine three times, one per each dimension in a $x \rightarrow y \rightarrow z$ order (*lines 3-5*), as described in the previous paragraph.

Fig. 7.5 shows the pseudocode of *exchange_faces()*. This routine exchanges boundary face data with two neighboring ranks (N_{d-1} and N_{d+1}) in the specified d dimension (*e.g*, N_{x-1} and N_{x+1} are the neighbors in the x direction). Note that *exchange_faces()* will only exchange faces if the given rank subgrid does not abut the physical boundary along the specified d dimension. We use the *isBoundary()* routine to determine whether a given face abuts a physical boundary.

While executing *exchange_faces()*, a rank first initializes two receive requests, one for each neighbor (*lines 3-4*), and then packs the two send buffers with boundary face information (*lines 5-6*). The *pack* routine stores the grid's (non-contiguous) fields specified in the *fields* argument into a contiguous buffer. Next, *exchange_faces()* initializes the send requests (*lines*

7-8) and waits until the exchanges finish (*line 9*). After the exchange completes, the rank calls *unpack* (*lines 10-11*) which moves the incoming contiguous data into the corresponding grid fields.

```
1  update_halo(fields)
2  {
3    exchange_faces(x, fields);
4    exchange_faces(y, fields);
5    exchange_faces(z, fields);
6  }
```

Figure 7.4: MPI pseudo-code of Cloverleaf3D's *update_halo()* routine (simplified).

```
1  exchange_faces(d, fields)
2  {
3    if (!isBoundary(N_{d-1})) MPI_Irecv(recvBuffer_{d-1} ← N_{d-1});
4    if (!isBoundary(N_{d+1})) MPI_Irecv(recvBuffer_{d+1} ← N_{d+1});
5    if (!isBoundary(N_{d-1})) pack(Grid(fields) → sendBuffer_{d-1});
6    if (!isBoundary(N_{d+1})) pack(Grid(fields) → sendBuffer_{d+1});
7    if (!isBoundary(N_{d-1})) MPI_Isend(recvBuffer_{d-1} → N_{d-1});
8    if (!isBoundary(N_{d+1})) MPI_Isend(recvBuffer_{d+1} → N_{d+1});
9    MPI_Waitall();
10   if (!isBoundary(N_{d-1})) unpack(Grid(fields) ← recvBuffer_{d-1});
11   if (!isBoundary(N_{d+1})) unpack(Grid(fields) ← recvBuffer_{d+1});
12 }
```

Figure 7.5: MPI pseudo-code of Cloverleaf3D's *exchange_faces()* routine (simplified).

7.2.2 MATE Variant

Although Cloverleaf3D was written in Fortran, we were able to apply the MATE model by manually introducing calls to the MATE runtime, instead of using an annotation-guided translation. To this end, we built a Fortran interface to MATE's API (appendix E.2) that allow us to achieve the effect of translation. To simplify our explanation, we will explain our transformations using annotations as if we operated on C/C++ source code.

We implemented a 2-level hierarchical decomposition in Cloverfield3D that allows local ranks to exchange boundary information without the need for MPI messages. However, unlike our Jacobi3D test case (Chapter 5), we did not implement a dotted line strategy, *i.e.,* a single allocation per MATE process and a logical division of work among local ranks. Instead, we

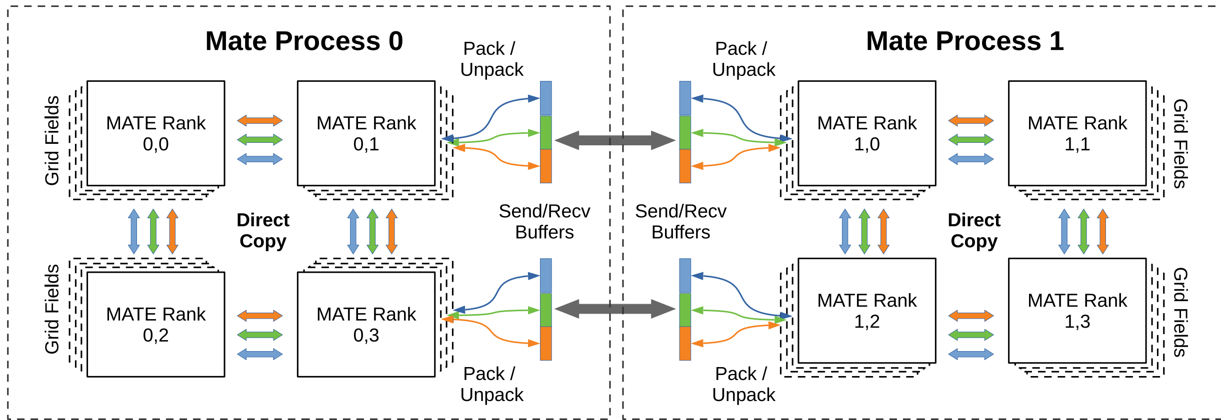

Figure 7.6: MATE hierarchical variant of Cloverleaf3D's halo exchange, represented in 2D for simplicity. Local ranks exchange boundary field information by direct copy while inter-process communication requires buffer packing/unpacking.

opted to allocate separate subgrids for each rank, and then broadcast their pointers locally for direct read/write access. The reasons for this decision are twofold:

1. The nature of pointer and array accessing in Fortran hinders the implementation of a dotted line strategy. To support this feature, we would need to introduce significant changes to each of the six solver kernels.

2. It is uncertain whether allocating process-wide grids would preserve cache-locality, given that the grid stores 14 different fields. It is possible that traversing the grid would cause cache-misses at every row, a problem that using a separate subgrid allocation per rank would otherwise avoid.

Despite using separate subgrid allocations, local ranks in our MATE variant avoid most of the cost of intra-node data motion by copying non-contiguous boundary information directly via *memcopy*, without the need to serialize data and exchange messages. Fig. 7.6 shows how MATE ranks exchange boundary information both locally, and across MATE processes. Although this approach does not avoid explicit intra-node copying entirely, it reduces the data motion costs considerably and vastly simplifies the implementation of the MATE model on Fortran applications.

```
exchange_faces(d, fields)
{
 #pragma mate graph
 {
  #pragma mate region(receive) depends (unpack*)
  {
   if (!isBoundary(N_{d-1}) && !isMATELocal(N_{d-1})) MPI_Irecv(recvBuffer_{d-1} ← N_{d-1});
   if (!isBoundary(N_{d+1}) && !isMATELocal(N_{d+1})) MPI_Irecv(recvBuffer_{d+1} ← N_{d+1});
  }

  #pragma mate region(pack) depends (unpack*, localExchange*@, send*)
  {
   if (!isBoundary(N_{d-1}) && !isMATELocal(N_{d-1})) pack(Grid(fields) → sendBuffer_{d-1});
   if (!isBoundary(N_{d+1}) && !isMATELocal(N_{d+1})) pack(Grid(fields) → sendBuffer_{d+1});
  }

  #pragma mate region(send) depends (pack)
  {
   if (!isBoundary(N_{d-1}) && !isMATELocal(N_{d-1})) MPI_Isend(recvBuffer_{d-1} → N_{d-1});
   if (!isBoundary(N_{d+1}) && !isMATELocal(N_{d+1})) MPI_Isend(recvBuffer_{d+1} → N_{d+1});
  }

  #pragma mate region(localExchange) depends (pack@)
  {
   if (!isBoundary(N_{d-1}) && isMATELocal(N_{d-1})) localExchange(Grid(fields) → N_{d-1});
   if (!isBoundary(N_{d+1}) && isMATELocal(N_{d+1})) localExchange(Grid(fields) → N_{d+1});
  }

  #pragma mate region(unpack) depends (receive)
  {
   if (!isBoundary(N_{d-1}) && !isMATELocal(N_{d-1})) unpack(Grid(fields) ← recvBuffer_{d-1});
   if (!isBoundary(N_{d+1}) && !isMATELocal(N_{d+1})) unpack(Grid(fields) ← recvBuffer_{d+1});
  }
}
```

Figure 7.7: MATE's pseudo-code of Cloverleaf3D's *exchange_faces()* routine (simplified).

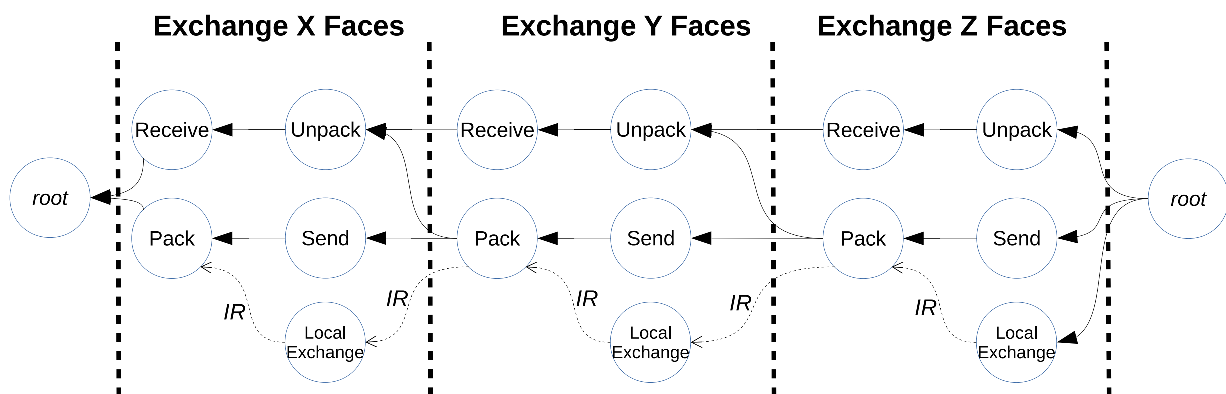

Figure 7.8: Graph derived from MATE annotations on the *exchange_faces()* routine.

In the MATE variant, ranks declare up to seven inter-rank dependencies: itself, and up to two local ranks per axis (if they exist in the same MATE process). We use the same logic as in Fig. 3.15, making calls to the *Mate_AddLocalNeighbor()* routine. Fig. 7.7 shows the annotated version of *exchange_faces()* we used in Cloverleaf3D's MATE variant. We declare a MATE graph with five regions: *receive, pack, send, localExchange,* and *unpack*. Fig. 7.8 shows the dependency graph described by the region annotations and their dependencies.

The *receive* region (*lines 7-8*) exchanges requests for boundary data from neighboring ranks, only if they reside in a different MATE process. That is, if the current rank's subgrid belongs in the MATE process' boundary but not the physical boundary. The *isMATELocal()* call verifies rank locality conditions and returns true if the neighbor rank belongs to the same MATE process. The *receive* region depends on the execution in the previous iteration (indicated by the '*' modifier) of the *unpack* region. This dependency guarantees that no incoming messages will overwrite the remote receive buffers before the ranks have unpacked them into grid fields.

The *pack* region (*lines 13-14*), copies boundary data from the subset of fields, indicated by the *fields* argument, into a contiguous send buffer. Similar to the *receive* region, the *pack* region only fills a send buffer if the neighboring rank belongs in a different MATE process. This region depends on the execution of the *unpack* region in the previous iteration. This dependency ensures that the incoming data from the previous iteration's face(s) are unpacked onto the grid's fields before sending the face(s) for the current iteration. This region also depends on the previous execution of the *localExchange* region by its neighboring local ranks (and by itself), to guarantee all previous local face(s) have arrived before sending the current face. Finally, there is a dependence on the previous iteration of the *send* region. This dependency guarantees that the send buffers used in the previous faces have been released from the previous send operations.

The *send* region (*lines 19-20*) produces send requests for boundary data only if the neighbor resides in a different MATE process. This region depends only on the *pack* region in the current iteration, ensuring that send buffers are ready and packed before issuing the send requests.

The *localExchange* region (*lines 25-26*) copies boundary data directly to their local neighbor's fields without packing, unpacking, or messaging. We developed *localExchange()* by combining the logic from the original pack and unpack procedures while removing their buffering component. In this way, field information is copied into the corresponding grid positions on the receiving local neighbor rank. This region depends on the execution of the current face's *pack* region from neighboring local ranks (and itself). This dependency acts as a local barrier that defers boundary exchange until all the neighbors have reached the current face.

Finally, the *unpack* region (*lines 31-32*) will copy the contiguous boundary information received from remote ranks into the corresponding fields of the rank's grid. This region only depends on the *receive* region to guarantee that data has arrived in the receive buffers before unpacking them into the grid. Note that, if a MATE rank does abut a boundary in the MATE process in the current dimension, then the *receive*, *pack*, *send*, and *unpack* regions will not perform any actions.

7.3 Strong Scaling Studies

7.3.1 Cori Phase I (Haswell)

We ran the *Basic-MPI* and *MATE* variants on 128, 256, and 512 nodes (4096, 8192, and 16384 Haswell cores) of the Cori Phase I system, with a 1024^3 element grid. We ran the MPI variant using one MPI process per core, using the 3D mapping geometry provided in the original source code. For the *MATE* variant, we determined that the best configuration uses 16 MATE processes $\times$ 2 threads per node, with 4 MATE ranks per process. This configuration corresponds to an overdecomposition factor of 2. We report performance in terms of the number of cells processed per second (inverse of the *time per cell* metric reported by the source code). To verify correctness, we check that the remaining *solution energy* is equal for both the baseline and MATE variants.

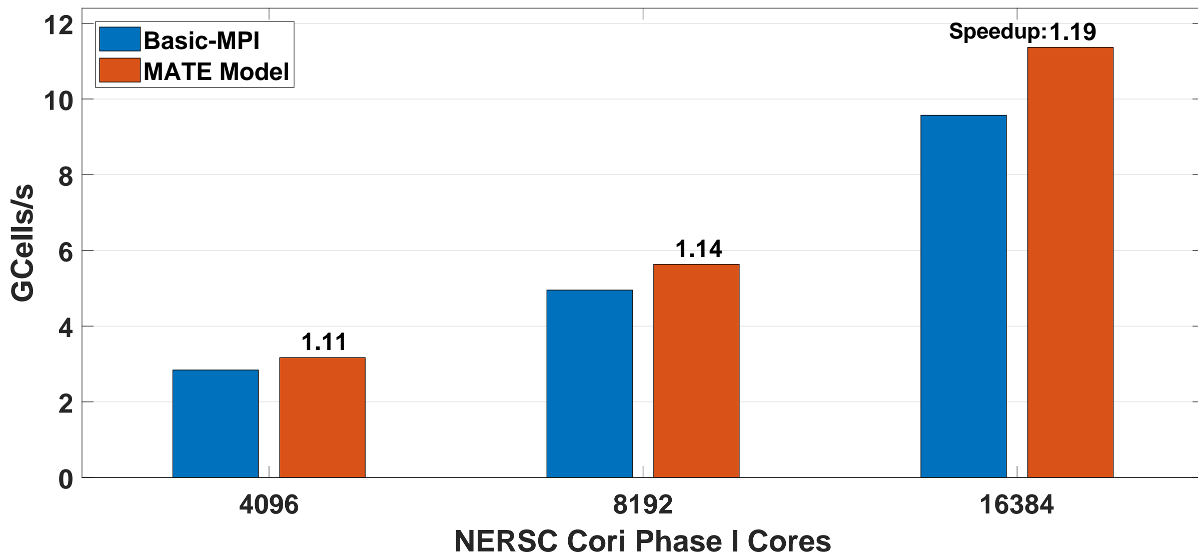

Figure 7.9: Strong Scaling results for Cloverleaf3D on 4k to 16k Cori Phase I cores.

Fig. 7.9 shows the results of our strong scaling study. MATE was able to outperform the baseline MPI algorithm at all scales. Furthermore, we observe increasing speedups as we double the number of nodes, from 1.11x on 4k cores to 1.14x on 8k cores, and 1.19x on 16k cores.

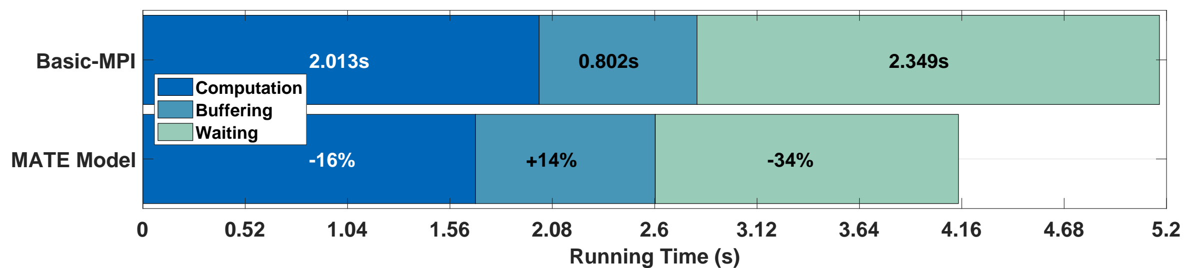

Figure 7.10: Execution breakdown on 16k Cori Phase I cores.

Fig. 7.10 shows the wallclock time spent by computation, buffering, and communication operations for the Basic-MPI variant on 16k Cori Phase I cores, and how much these times differ for the MATE variant. The *MATE* variant was able to hide 34% of the communication cost and achieved a 1.19x speedup compared to the baseline variant. However, it also suffered from a 14% increase in the cost of buffer serialization. We attribute this increase to the use of overdecomposition since, even if it does implement a local exchange mechanism, our variant does not entirely eliminate the cost of additional intra-node data motion introduced by overdecomposition. We

also observe a 16% decrease in computation time. It is possible that the reduced per-rank subgrid size that comes from overdecomposition produced a cache-blocking effect, similar to that described in chapter B, that decreased the number of cache fails during kernel executions.

7.3.2 Cori Phase II (KNL)

For the Cori Phase II system, we ran our two variants on 128, 256, and 512 nodes (8192, 16384, and 32768 KNL cores), keeping the grid size fixed at 1280^3. We ran the MPI variant using one MPI process per core and, for the *MATE* variant, we determined that the best configuration uses 8 MATE processes $\times$ 8 threads per node, with 16 MATE ranks per process. This configuration represents an overdecomposition factor of 2.

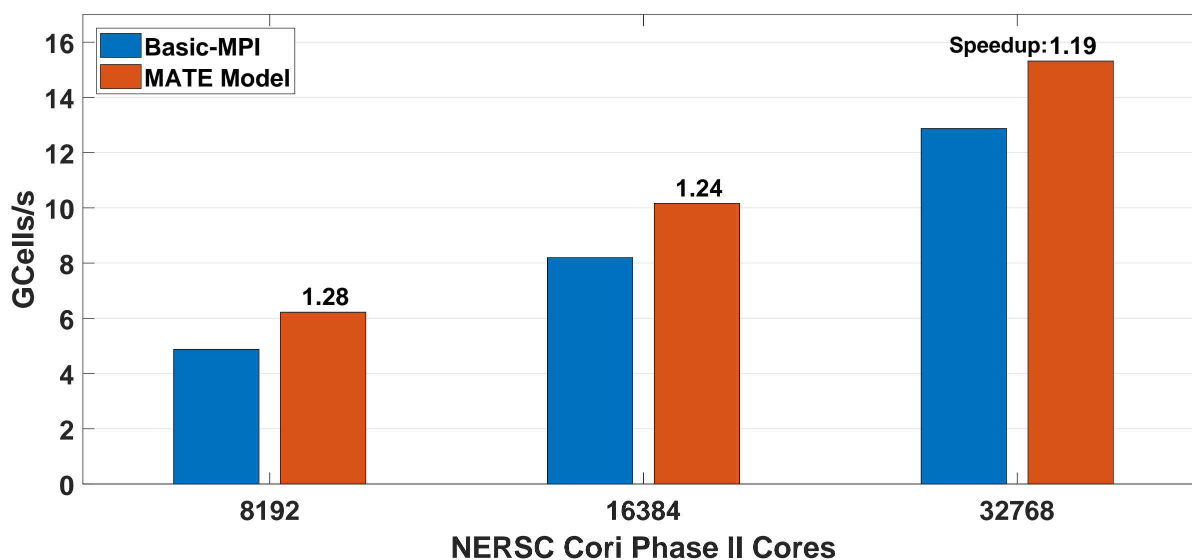

Figure 7.11: Strong Scaling results for Cloverleaf3D on 8k to 32k Cori Phase II cores.

Fig. 7.11 shows the results of our strong scaling study. MATE was able to outperform the baseline MPI algorithm at all scales. Contrary to our Phase I results, we observe decreasing speedups as we double the number of nodes, from 1.28x on 8k cores to 1.24x on 16k cores, and 1.19x on 32k cores. To understand this effect, we calculated the ratio of computation, buffering, and communication costs on the running of both variants at different scales.

Table 7.1: Ratio of computation, buffering, and network communication operations on 128, 256, and 512 nodes for the Basic-MPI and MATE variants on Cori Phase II.

	Basic-MPI			**MATE**		
Scale	Compute%	Buffer%	Network%	Compute%	Buffer%	Network%
128 Nodes	46%	16%	31%	62%	16%	22%
256 Nodes	40%	16%	44%	56%	16%	28%
512 Nodes	34%	16%	50%	42%	24%	34%

Table 7.1 shows that MATE achieves a 62% effective core usage on 8k cores, only dropping to 56% on 16k cores, which explains the slight loss of speedup. However, the main difference occurs at 512 nodes, where MATE suffers a higher buffering cost (24%) than in all the other runs (16%). This increase in intra-node data motion accounts for the higher loss of speedup on 32k cores (1.19x) when compared to the speedup at 16k cores (1.24x). To figure out the cause, we measured wallclock time spent by computation, buffering, and communication operations for the Basic-MPI variant on 512 Cori Phase II nodes, and how much they differ for the MATE variant as shown in Fig. 7.12.

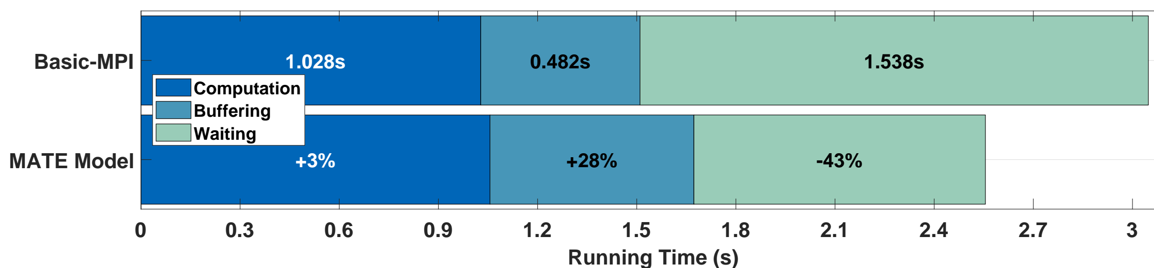

Figure 7.12: Execution breakdown at 512 Cori Phase II nodes.

Although MATE was able to 43% of the communication cost, we also observe a 28% increase in packing time. This could be explained as a consequence of excessive buffering serialization costs at small per-core subgrid sizes which is exacerbated by overdecomposition. These costs are much higher than those in Cori Phase I, which contains half the number of cores per node (and therefore fewer boundaries to exchange), where MATE only suffers an 14% increase in packing time.

7.4 Summary

We have shown that the MATE model can improve the performance by overlapping communication and computation and improving locality of a finite volume solver that uses a multivariable staggered grid at large scales and on multiple supercomputing platforms. Our results in all three platforms show that MATE was able to reduce a significant amount of communication overheads obtaining performance speedups of 1.19x at the largest scales on both testbeds. These results also show that MATE can improve the performance of complex applications, when a manual re-factoring is difficult or impractical.

Even if it does not entirely remove the cost of intra-node communication, MATE enables the use of efficient overdecomposition that reduces the total cost of communication significantly by overlapping it with computation. Furthermore, inter-rank dependencies have once again proven to be essential in enabling a MATE application to execute with an efficient synchronization and communication logic between local ranks.

Acknowledgements

This chapter is, in part, a reprint of the material contained in the article: "*MATE, a Unified Model for Communication-Tolerant Scientific Applications*", by Sergio M. Martin and Scott B. Baden, which appears in the Proceedings of 31st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2018), Salt Lake City, UT, USA, October 2018. This dissertation's author was the primary investigator and author of this paper.

Chapter 8

Load Balancing

8.1 Overview

Workload distribution represents an essential aspect of the performance of scientific applications. In a properly balanced application, every core in the system performs an equal amount of computation before reaching the communication phase. However, for some applications, especially those with irregular domain decompositions, achieving proper workload balance could be infeasible or could negatively affect other aspects of application performance. These applications suffer from *Load Imbalance*.

Load imbalance occurs when an application's computational workload decomposes unevenly among its ranks (cores), delegating more computational work to some ranks than others. This phenomenon will –assuming homogeneous hardware– cause some ranks to finish their part of the workload significantly before others, delaying the exchange of messages, and wasting computational resources due to extended wait times.

Fig. 8.1 shows the effect of load imbalance on an application programmed under the *Bulk-Synchronous* model. In this example, process *0* computes a smaller workload compared to that of process *k*. As a consequence, process *0* will finish its computation stage and issue

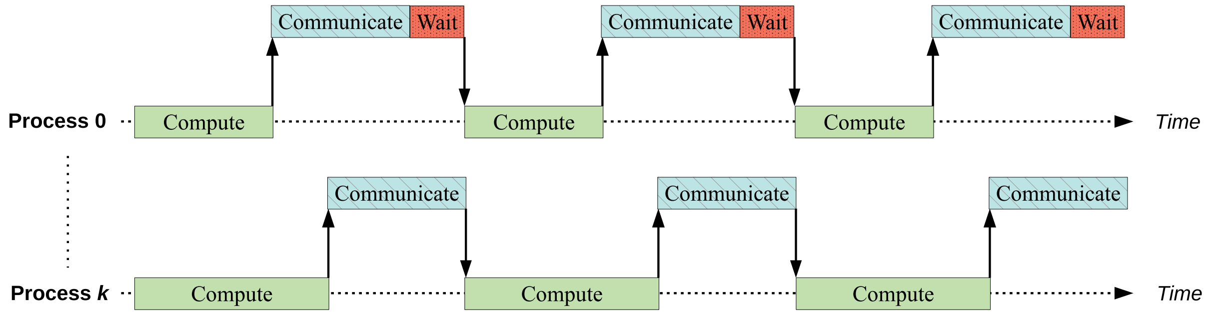

Figure 8.1: Execution timeline of an imbalanced application.

message exchange requests much before process k does. These requests cannot progress[1] until process k issues reciprocal requests. Therefore, process *0* will spend additional *idle* time (shown as red/wait segments) waiting for process k to complete its communication phase.

The additional waiting time that comes from load imbalance delays process *0* until process k finishes its computation phase. Therefore, in an imbalanced application, the time spent in the computation phase for *every* rank becomes equal to the cost of that with the maximal workload. Since this additional cost manifests itself during communication waits (*e.g.,* during *MPI_Waitall*), it could lead to the incorrect conclusion that irregular applications suffer from excessive network or intra-node communication overheads.

We can estimate the component of the total running time that comes from load imbalanced with the Formula in Eq. 8.1, where T_{Cmax} is maximum time spent in computation time by any given rank, T_{Cmean} is the mean computation time across all ranks, and T_{total} is the total running time of the application.

$$\%ImbalanceCost = \frac{T_{Cmax} - T_{Cmean}}{T_{total}} \tag{8.1}$$

We use the mean computation time –and not the minimum computation time– because, in a perfectly balanced application, all ranks would finish computation precisely at the mean time. Hence, we say that the difference between the maximum and mean computation times is the *cost*

[1]Although asynchronous *send* operations can begin during this waiting period, this progress does not help in reducing the total waiting time from *receive* operations.

of opportunity lost from load imbalance. Fig. 8.2 illustrates this cost by showing two variants of the same application: A and A'. Although variants have the same total workload, variant A has an irregular distribution of workload while the workload in variant A' is evenly distributed.

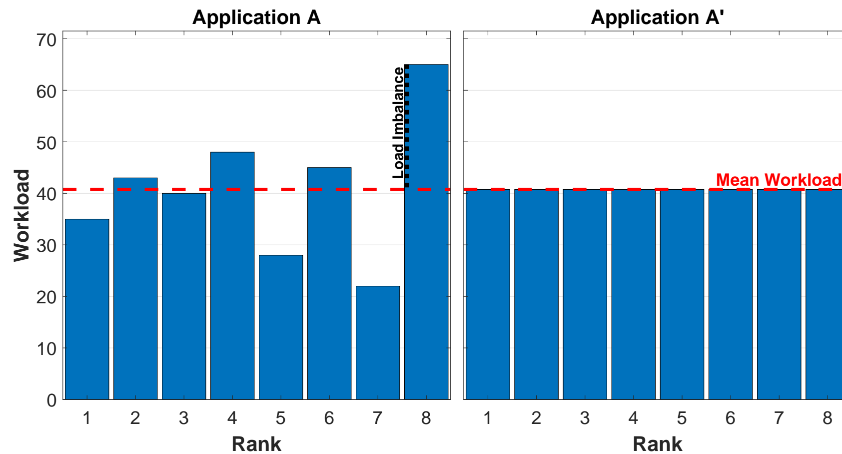

Figure 8.2: Workload distribution of applications A and A'.

Though steps can be taken to avoid load imbalance can certainly be caused by poor programming, some scientific applications motifs contain inherent irregularities that cannot be avoided. Such is the case of unstructured multi-grid (*e.g.*, [3][95] an aerodynamics design solver) or adaptive mesh refinement (*e.g., CASTRO* [5], an astrophysical hydrodynamics solver) applications.

In these applications, spatial locality represents a stronger factor for performance than attaining a perfect distribution of work. Therefore, they require a careful tuning between two parameters: (1) the spatial distance and connectivity between subgrids, and (2) workload distribution. In this section, we use one of such applications, *Mpix_flowCart*, to analyze the potential benefits of the MATE model in mitigating the effects of load imbalance while preserving rank locality.

8.2 Example: Mpix_flowCart

Mpix_flowCart is a high-fidelity inviscid analysis package for conceptual and aerodynamic design. This code is developed by scientists at the NASA Ames Research Center and the Courant Institute at New York University and has many users in the aerospace design industry.

Mpix_flowcart uses a Cartesian multi-level embedded boundary mesh with cut cells located where the mesh intersects the geometry (*e.g.,* a wing's surface). It partitions the mesh hierarchy into one subdomain per rank at run-time, using a space-filling curve to order the meshes.

Space-filling curves [47, 75] work by partitioning an irregular grid in way that maximizes both spatial locality and workload balance. Spatial locality ensures that ranks to be more likely to communicate with their immediate neighbors, reducing network communication. *Mpix_flowcart* employs a multigrid scheme that iterates over coarser meshes which are also ordered with a space-filling curve.

One of the lessons we learned with *mpix_flowcart* is that the cost of load imbalance can disguise itself as communication overhead. Indeed, our initial analysis showed that the *MPI_Waitall* operation consumed a significant part of the application's running time. However, from analyzing the behavior of individual ranks, we found that most of the time spent on *MPI_Waitall* came from ranks waiting for others to finish their computation phase.

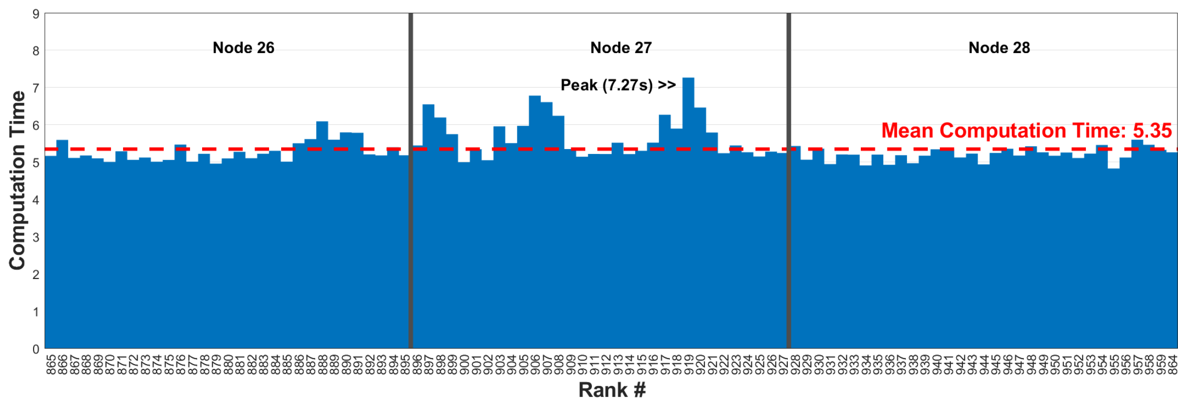

Figure 8.3: Computation time distribution of nodes 26, 27 and 28 on a 2048 Haswell core-run.

To visualize the extent of workload imbalance in mpix_flowcart, we performed a test run on 1024 Cori Phase I nodes (32768 cores) in which we stored the computation times of each rank. Fig. 8.3 shows the results of our study, zoomed in to nodes 26, 27, and 28. Each bar indicates the computation time for an individual rank. We observed that peak computation time occurs on 918 (7.27s), while the mean computation time across these three nodes is 5.35s and the total running time is 8.83s. Therefore, using our formula in Eq. 8.1 we were able to determine that its load imbalance amounts to $\sim 22\%$ among these three nodes. The observation that workload imbalance is a dominant component of the running time of *mpix_FlowCart* motivated us to ponder whether we can use MATE to improve its balance.

8.3 Rebalancing with MATE

We realized that the number of partitions (*i.e.,* number of ranks) that the *Mpix_FlowCart*'s partitioner creates is fixed by the number of cores in the baseline MPI execution. This configuration limits the granularity on which the space-filling curve can distribute work among cores, and thus its ability to achieve a better workload balance. Fig. 8.4 (*top*) shows the computation times for 32 cores in node 27, each executing a single rank. To overcome this limitation, we implemented a set of mechanisms provided by our MATE runtime system.

8.3.1 Mechanism I: Hierarchical Overdecomposition

First, we implemented overdecomposition to increase the number of partitions per core. Fig. 8.4 (*middle*) shows the effect of applying an overdecomposition factor of 4. This figure indicates the total computational time spent by each core (thick blue bars). In turn, each core executes four ranks (red thin bars) that take each roughly a fourth of the time to compute. We can see that, by using overdecomposition, *Mpix_FlowCart*'s space-filling curve has a higher degree of freedom in refining the partitions that represent a better fit for the input mesh. As a

consequence, we observe a slight smoothing effect on the redistribution of work. The busiest core is now 21 with 7.059s, yielding a 19% imbalance ratio among the cores in the node.

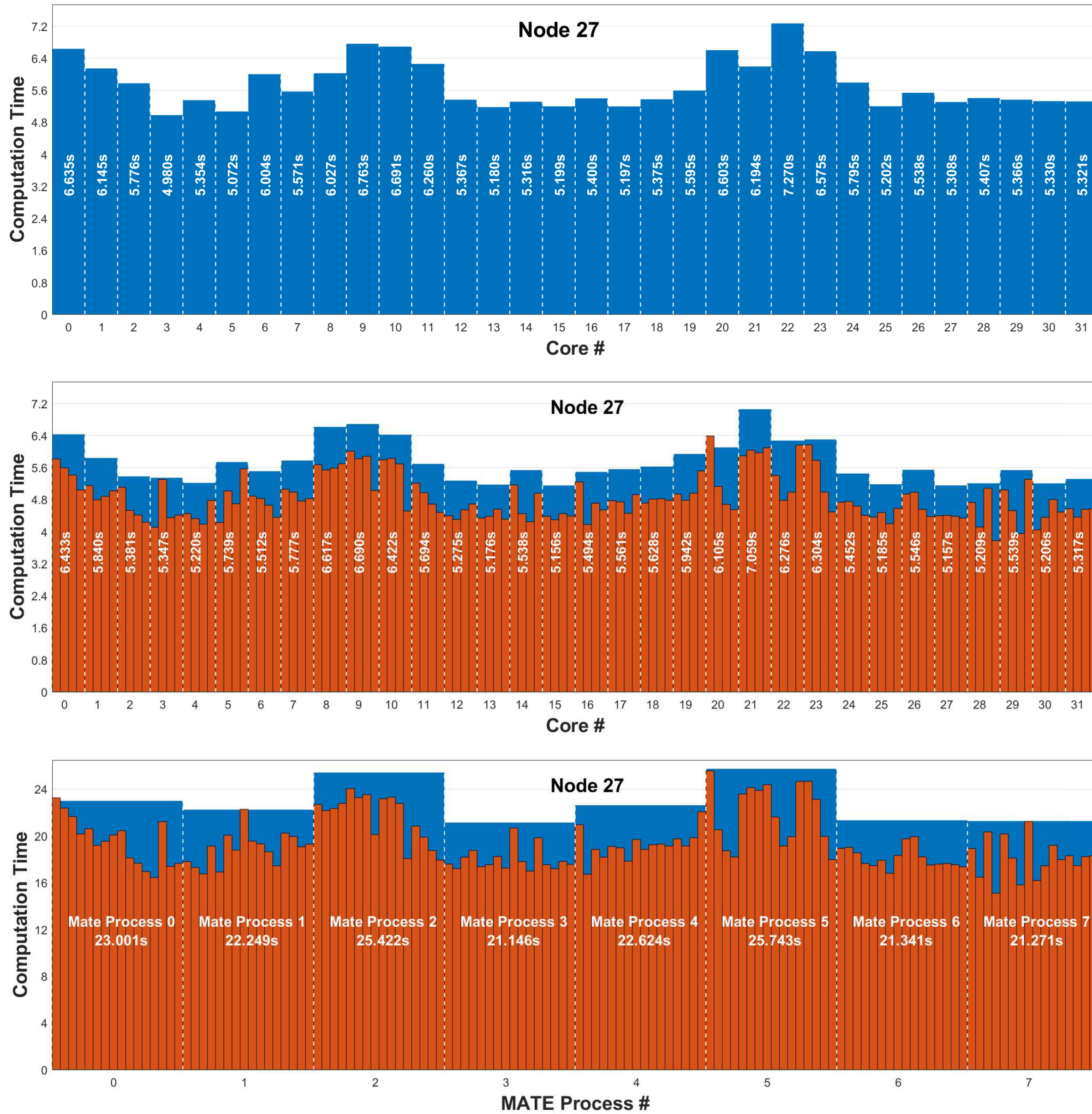

Figure 8.4: Computation time distribution of the 32 cores on node 27, *(top)* without overdecomposition, *(middle)* using an overdecomposition factor of 4, and *(bottom)* using a hierarchical decomposition with 4 workers per MATE process.

Second, we applied MATE's hierarchical decomposition. In a hierarchically decomposed execution, ranks are no longer bound to a single core nor do they execute out of private memory,

but are instead shared among all the cores in a MATE process (defined in chapter 4). This configuration smooths the computation times within the process. We illustrate this effect in Fig. 8.4 (*bottom*), where we execute 8 MATE processes per node, each spanning four cores. This effect allows us to consider workload distribution at a MATE process-level, instead of a core-level. In this case, the 16 ranks in MATE process 5 sums up to 25.743s out of a 22.849s per-process mean, yielding an 11% imbalance at a MATE-process level.

8.3.2 Mechanism II: Inter-Node Balancing

Although we can flatten the workload distribution of cores within a given node, we still face the challenge of improving *inter-node balance*, that is, balancing the workload among nodes. Inter-node imbalance occurs when the space filling curve assigns more work to one node than the average across all nodes, even if rank workload within each node is perfectly balanced. When this imbalance occurs, the ranks in one node finish later than in other nodes, incurring a cost that cannot be addressed by intra-node balancing only.

Fig. 8.5 shows the execution times of 24 MATE processes spanning three consecutive nodes (nodes 26 to 28). We can see a disparity between the total load of each node, with node 27 spending 183s in computation whereas the mean among the three nodes is 175s, thus causing a 4% imbalance.

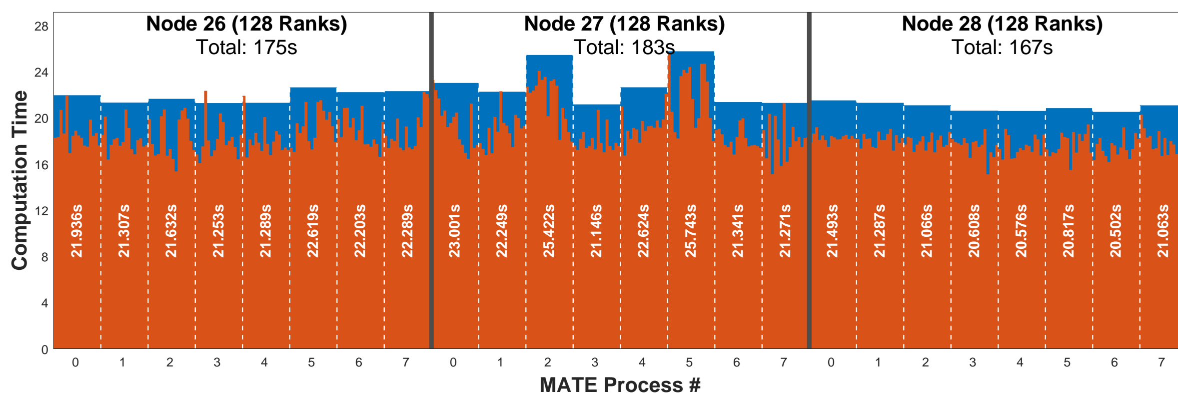

Figure 8.5: Workload imbalance across three nodes.

To solve this problem, we extended MATE's functionality to allow rank migration across node boundaries. That is, we can enable nodes with a higher workload to yield costly ranks to less occupied nodes. To enable this re-distribution, a MATE overdecomposed application instructs processes in one node to 'donate' ranks to processes in neighboring nodes while retaining enough ranks to keep cores busy[2]. To enable rank migration, we created the *Mate_TaskMapping()* MATE Runtime API call (see: Appendix E.2) that allows users to assign a custom number of MATE ranks per process.

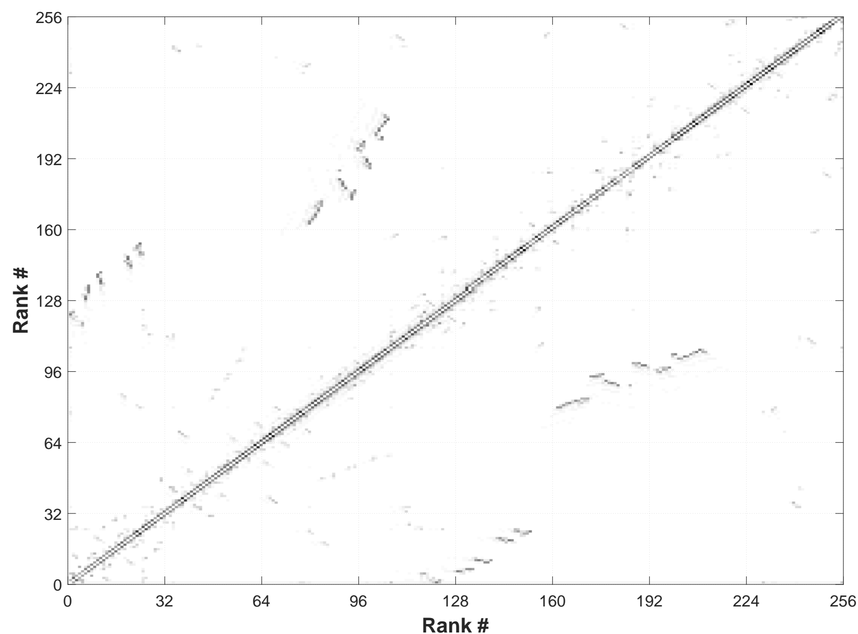

Figure 8.6: Rank-to-Rank communication distribution in mpix_flowCart. Darker spots indicate a higher communication volume; clearer spots indicate zero or little communication.

From our analysis of mpix_flowCart's, we learned that its space-filling curve optimizes locality to maximize communication among consecutive ranks while minimizing communication across non-consecutive ranks. Fig 8.6 illustrates mpix_flowCart's locality by showing the density of rank-rank communication in a 256 rank execution (each 32x32 square represents a node). We can see in the figure that communication volume is higher among neighboring ranks.

[2]Note that it is not possible to assign an uneven number of ranks per node in flat MPI variants, where ranks execute as processes mapped 1:1 to a core, since assigning fewer ranks to a node will cause some of its cores to remain idle, wasting computational resources.

Indeed, we observed that we cannot change the order or ranks across node boundaries without a costly impact in performance ($\sim 20\%$ slowdown, per our experiments), even if inter-node balancing improves.

To improve inter-node balancing, we faced the problem of defining the best distribution of ranks among nodes, without changing their ordering. We discovered that this problem is analogous to the well-defined *Painter's Partitioning Problem* [101] also known as the *Fair Workload Problem* [115], both of which have a polynomial $O(nlog\sum_{i=1}^{r} C_i)$ solution, where n is the number of nodes, r is the number of ranks, and C_i is the communication time of rank i. We adapted the solution provided in [101] to mpix_flowCart (the rationale and MATLAB code can be found in appendix D.1) by using the individual computation times of each rank as input. We obtained these values by running a short execution (10 iterations) of the solver[3].

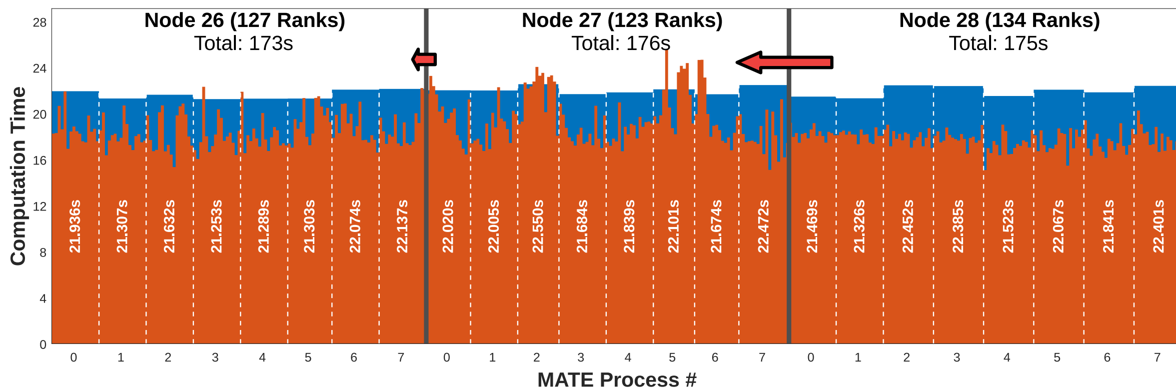

Figure 8.7: Workload distribution from 8.5 after inter-node rebalancing. Node 26 donated one rank to node 27 which, in turn, donated six ranks to Node 28, moving rank distribution (thick gray lines) to the left.

Fig. 8.7 shows the result of applying our solution. First, we can see that workload distribution at a node-level has improved slightly. Second, we can see that the number of ranks per MATE process is not uniform. Indeed, some processes contain fewer (down to 14, for node 27 / process 2) ranks, while others contain more (up to 17, for multiple processes) ranks.

[3]Other applications may provide workload metrics (*e.g.*, how many cells to compute per rank) without the need of sampling runs.

Furthermore, this redistribution produces a smoothing effect that reduces the maximum process load down to 22.5s (node 27, process 2).

8.3.3 Mechanism III: Intra-Node Balancing

We established that a balancing algorithm should not alter the ordering of ranks across nodes since that would increase the cost of inter-node communication. This constraint, however, does not hold in the case of intra-node balancing. Since intra-node communication is mostly homogeneous[4], we can freely change the order of ranks among MATE processes inside a node.

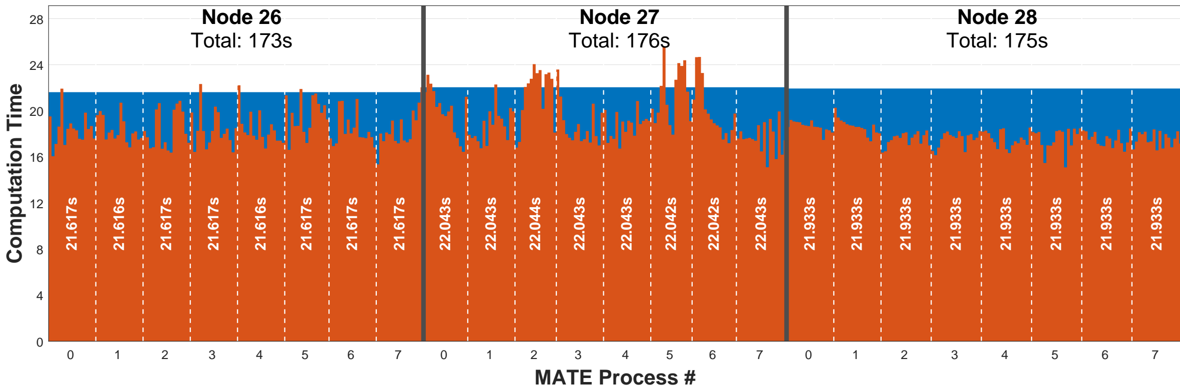

Figure 8.8: Computation time distributions of 16 MATE processes after rank re-shuffling.

Without the restriction of conserving rank ordering, we have are $n!$ possible combinations that, making an exhaustive search unfeasible. Nevertheless, we developed an algorithm (we detail its rationale and MATLAB code in Appendix D.2) that iteratively smoothens peak process workloads by re-distributing their ranks among the other processes in the node. Our algorithm has a $O(kn^2)$ complexity, where k is the number of processes, and n is the number of ranks per process. We have confirmed through experimental testing that our solution achieves a satisfactory balance[5] after 50-100 smoothing iterations, which require negligible time.

[4] Although re-shuffling the order of ranks can increase communication across NUMA domains and L2/L3 cache structures, we determined these factors do not hinder performance enough to enforce a consecutive ordering.

[5] Since there is no polynomial-time way to find an optimal solution, we cannot define a precise upper bound.

Fig. 8.8 shows the result of applying our intra-node balancing algorithm. We observe that the workload distribution at a node-level has mostly flattened. Although there is still a minimal inter-node imbalance, the algorithm eliminates computation time peaks among MATE processes within the node.

8.4 Experimental Results

To evaluate the effects of workload balancing through the mechanisms provided by the MATE model, we conducted experiments on the Cori Phase I and II testbeds. We used *mpix_flowCart* to model the flow for the *OneraM6* wing [79, 42], a well-known standard test case for aerodynamic simulation. Every experiment solves a 75Mcell mesh for 20 iterations and four multigrid levels with a CFL number of 1.2. We report performance in total running time (less is better). To verify correctness, we check that the remaining solution energy is equal for both the baseline and MATE variants.

We ran the baseline *Flat-MPI* and *MATE* variants on 64 nodes (2048 Haswell cores) of the Cori Phase I system. We ran the MPI variant using one MPI process per core and the *MATE* variant with 16 MATE processes $\times$ 2 threads per node, with 8 ranks per process. This configuration represents an overdecomposition factor of 4.

Fig. 8.9 shows the results of our study on Cori Phase I. The (dark blue) bars at the bottom indicate the computation time, the (light green) bars at the middle estimate the imbalance cost, and the (lighter green) bars at the top indicate actual communication time. The horizontal (dotted red) line represents the mean computation time across all ranks.

Our results show that, for the baseline variant (*top*), computation time comprises 60% of the total running time, while waiting periods accounted for the remaining 40% of the time. However, by zooming in, we observed that peak computation times occur on ranks 0 (7.28) and 918 (7.20s), while the mean computation time across all nodes is 5.34s. Therefore, using our

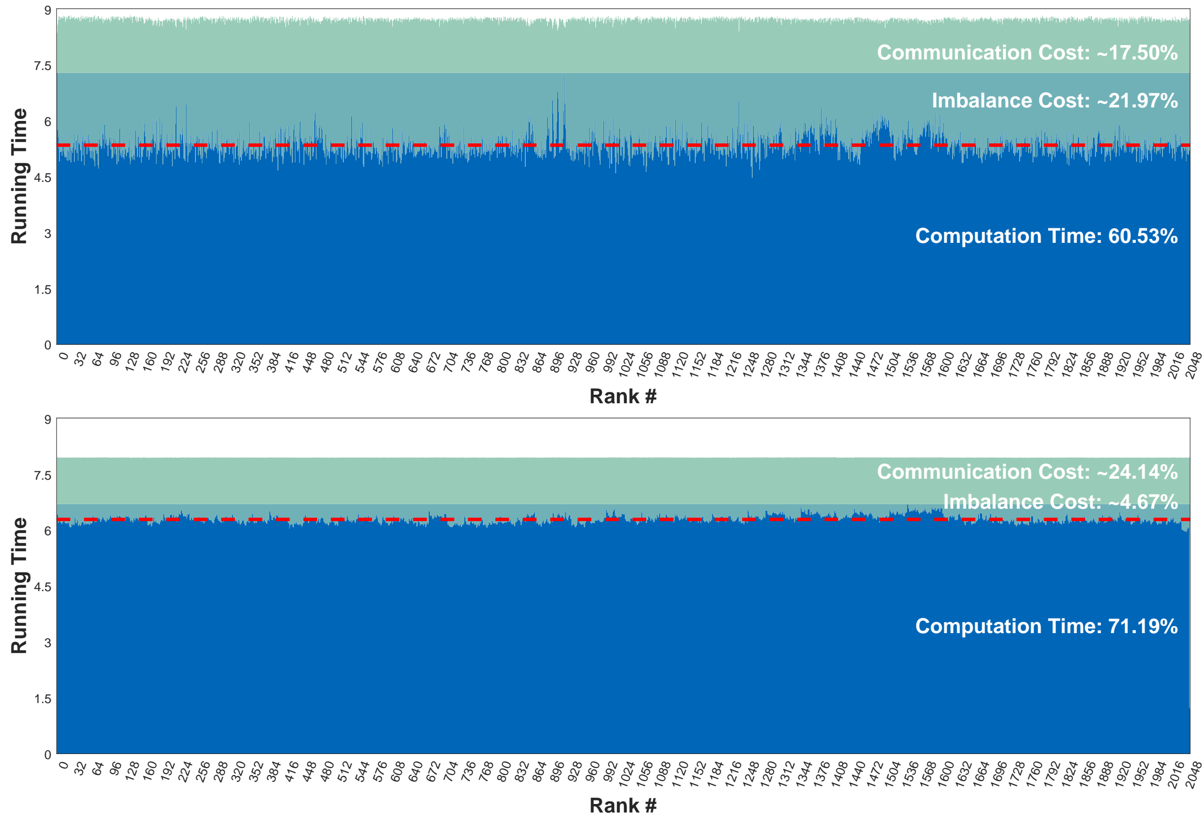

Figure 8.9: Running time breakdown of 2048 Cori Phase I cores for (*top*) the baseline variant, and (*bottom*) the MATE rebalanced variant.

formula in Eq. 8.1 we were able to determine that load imbalance was a more significant cost ($\sim 22\%$) than intra-node and network communication ($\sim 18\%$).

For the MATE variant (*bottom*), we observed a much smoother distribution of computation times with a maximum of 6.703s out of an average of 6.290s, producing a workload imbalance cost that comprises only 4.67% of the running time. On the other hand, we observe that the average computation time across all cores has increased by (from 5.34s, in the baseline variant to 6.290s). We attribute this excess cost to a loss of cache efficiency from overdecomposition. Although the ratio of communication increased from 17.5% to 24.1%, MATE did achieve a reduction of 13% in the absolute communication time when we adjust for the decrease in total running time. Overall, the MATE variant obtained a 1.11x speedup over the baseline variant.

8.5 Concurrency Limitations

Our re-balancing strategy provides the highest benefits for mpix_flowCart when we instantiate multiple ranks per core. Ideally, imbalance reduction would maximize when using a single MATE process that spans all the cores and ranks in a node or NUMA domain. However, we have found that we obtained the best results when using two threads per MATE process. Using a larger number of threads yields slower performance, even worse than the base variant.

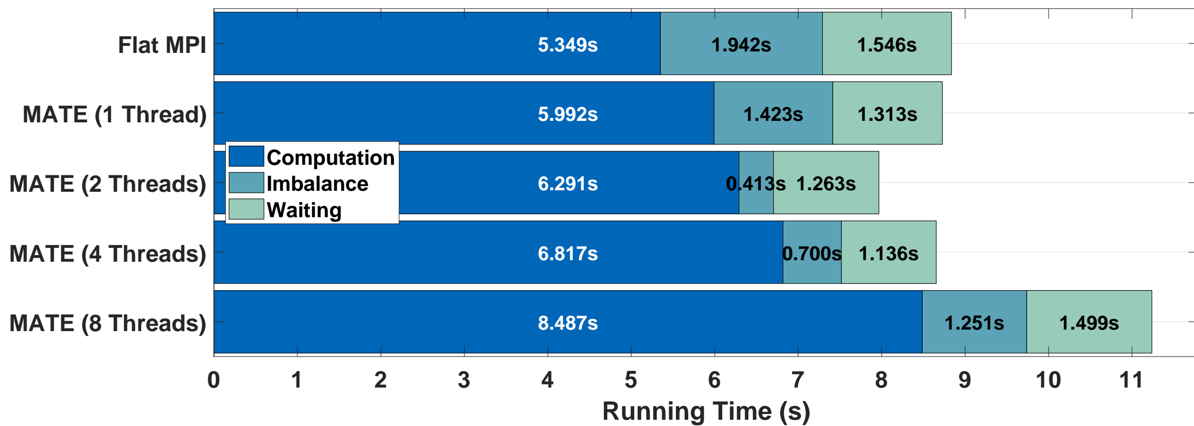

Figure 8.10: Execution breakdown for different multi-threading levels on Cori Phase I.

Fig. 8.10 shows our results for the Cori Phase I system. We employ the same configuration for the baseline variant, but run the MATE variant using different levels of multi-threading, from 1 thread per process to 8 threads per process. These results show that the MATE (1 Thread) variant achieves a small reduction of load imbalance thanks to the use of overdecomposition and inter-process balancing, but cannot benefit from shuffling ranks among different cores since each process runs a single thread.

The MATE (2 Threads) variant is the one that achieves the best results, reducing a large part of the load imbalance cost. However, it suffered from a larger cost computation. This trend in increasing costs of computation continues increasing towards the MATE (4 Threads) and MATE (8 Threads), with the latter suffering a noticeable slowdown.

Through the use of profiling tools, we ruled out the possibility of cache-locality loss

as the primary source of overheads, although they certainly contributed to a small degree. We traced the increased costs to thread stalls during the creation and completion test of MPI requests. Since we have not observed such a dramatic slowdowns in our other experiments, we focused *mpix_flowCart*'s communication patterns to determine what made them different.

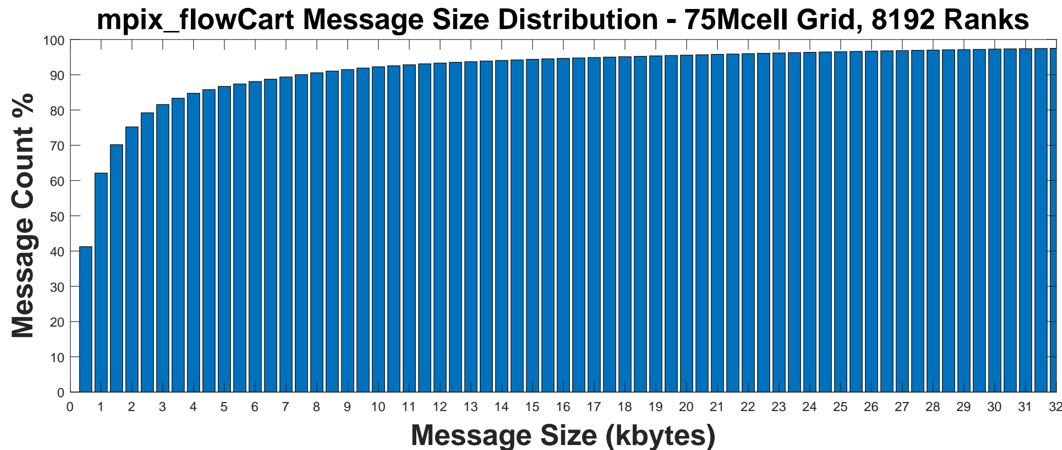

Figure 8.11: Message size histogram for *mpix_flowCart*.

We found that, unlike our other test cases, *mpix_flowCart* exchanges a large number of very small messages due to its use of coarsening grids. Fig. 8.11 shows the distribution of message sizes in a 64 node-run of *mpix_flowCart*, solving a 75Mcell using four multi-grid levels and an overdecomposition factor of 4. We observe that 90% of messages are smaller than 8kb, while 75% are below 4kb.

In C we describe the adverse effects that multi-threaded applications suffer when they that rely mostly on small messages. In *mpix_flowCart* case, 98% of the messages exchanged fall well beneath MPI's eager limit. As a consequence, threads within MATE will destructively compete for the use for access to the network, suffering noticeable stalls. For small number of threads (*e.g.,* 2), this overhead's impact is not enough to offset the benefits of re-balancing. However, we have seen that higher threading levels severely punish performance.

This limitation in the MPI library prevents us from attaining better results that may come from higher levels of multi-threading. Furthermore, these limitations also prevents us from

applying the full MATE model and enabling communication overlapping on *mpix_flowCart*. Future work may include adapting mpix_flowCart to use alternative communication libraries better suited for multi-threading.

8.6 Related Work

Prior work in workload balancing can be categorized into *static* and *dynamic* approaches. Static balancing requires pre-calculated information about either the application's behavior (*e.g.*, auto-tuning techniques [34]) or the network topology [90] to determine the best distribution of work and improve locality. Since static balancing defines these parameters at launch time, this approach works best in applications that do not change workload distribution or communication patterns during execution.

Dynamic balancing libraries, such as AMPI [12, 17], enable applications to migrate ranks across nodes during execution, based on core usage information generated during runtime. AMPI allows a programmer to instantiate overdecomposed MPI applications and specify a re-balancing policy that uses real-time metrics to determine when and how to migrate ranks. This approach typically provides better results in irregular applications whose workload change during runtime[6]. The downside of dynamic balancing is that migrating ranks has a large associated cost that is only justified in cases of severe load imbalance.

The novelty of our approach is that it implements elements from both static and dynamic re-balancing techniques. We employ pre-calculated computation times to determine the best distribution of ranks that minimizes inter and intra-node workload imbalance statically at launch time. On the other hand, a hierarchically overdecomposed MATE process dynamically assigns ranks to workers to maximize core usage opportunistically. Although our dynamic strategy is

[6]We have found through testing, however, that executing mpix_flowCart with AMPI ran much slower than the baseline MPI algorithm, even when not employing overdecomposition nor rank migration. This slowdown could be a result of an implementation issue, causing the current version of AMPI to perform poorly on our testing platforms or the Aries network.

restricted to the ranks inside each process, it has little to no associated cost. We have indeed achieved the best results for mpix_flowCart by combining static and dynamic mechanisms.

8.7 Summary

We have shown that MATE can improve the workload balance in mpix_flowCart. First, overdecomposition gives its space filling curve additional degrees of freedom to create smaller partitions that can migrate across different cores. Second, hierarchical decomposition allows us to smooth the workload of multiple cores within a MATE process. Finally, we modified our runtime system to allow a re-ordering of ranks across MATE processes.

Our experience shows that consecutive re-balancing improves workload balance at a node-level without increasing the volume of intra-node communication. Furthermore, we show that an optimal solution for this mapping can be found in polynomial time. We have also demonstrated that intra-node mapping can withstand rank reordering across MATE processes without a penalty to performance. We show that these mechanisms allow us to benefit from a combination of static and dynamic balancing effects.

Our results show that MATE was able to improve the performance of mpix_flowCart across multiple platforms, without the need for optimizing or altering its original space-filling curve. By applying MATE mechanisms, we obtained a 1.11x performance speedups on 64 Cori Haswell I nodes. We plan to explore a broader array of irregular applications so we can further generalize our approach.

Chapter 9

Conclusions and Future Work

9.1 Research Contributions

Traditional programming models and libraries, such as MPI, have been widely used to develop large-scale scientific applications for decades. However, these models have grown increasingly incapable of providing appropriate support for tolerating the ever-increasing costs of communication in both current Petascale and future Exascale supercomputers.

In this dissertation, we introduced MATE, a new programming model that supports the development of communication-tolerant scientific applications. MATE integrates multiple communication-reducing mechanisms into a single programming model, providing a benefit that is greater than the sum of the parts. Unlike prior work, our model reduces the two primary sources of communication cost –network and intra-node communication– simultaneously via a unified interface.

MATE introduces new mechanisms for developing communication-tolerant applications. First, it exposes a hierarchical decomposition and locality model that enables efficient overdecomposition, a technique that enhances communication/computation overlap, avoiding an increase in (and even reducing) explicit intra-node data motion. Second, it enables the programmer

to partially order computation via a dependency graph that exposes additional parallelism and supports efficient local synchronization. We have developed a programming framework comprised of a source-to-source translator, annotation syntax, and runtime system that supports the MATE model and requires only modest changes to existing MPI application source.

We showed that MATE realized a noticeable reduction in communication cost on large-scale experiments in three scientific applications. MATE's ability to hide network communication with the help of overdecomposition, while managing intra-node data motion costs, provides a novel way to manage the growth of communication costs on future Exascale systems. We showed that MATE improves the performance of a structured grid and a dense matrix multiplication algorithms. Additionally, in the case of Cloverleaf3D, we demonstrate that MATE can reduce the cost of communication of complex applications in which manual refactoring for overlap would be difficult or impractical.

Finally, we described how MATE could reduce load imbalance in an irregular multi-grid solver using overdecomposition and rank-reordering heuristics. Our approach combines the benefits of static and dynamic load balancing techniques incurring a reasonable amount of modifications to the source code.

9.2 Limitations and Future Work

9.2.1 Improve Thread Concurrency

As discussed in Appendix C, we are still facing concurrency issues in our MPI-based communication backend due to the use of multi-threading and overdecomposition. Some effort has been dedicated to avoiding thread serialization in the latest versions of Cray-MPICH. The *craympich-mt* flag permits an application to link to a variant of the Cray-MPICH that employs a per-object lock, instead of a process-wide lock, allowing the simultaneous access and creation of different MPI requests. However, we observed only a small reduction in the serialization costs

and no improvement in network communication times.

We are currently contemplating using UPC++ as communication backend instead of MPI which may provide better concurrency for MATE threads. Another possibility is to interpret MPI applications through a UPC++ backend to improve network saturation. However, early experiments with this approach indicate that emulating two-sided operations on top of a one-sided+RPC backend requires additional bookkeeping overheads that may offset any benefits.

9.2.2 UPC++ Integration

We are working on a prototype to integrate the UPC++ communication library into the MATE model. Our preliminary results show that a MATE annotated UPC++ program can produce the same results as the baseline program, indicating that it will be possible, as part of future developments, to build communication-tolerant UPC++ applications that take full advantage of the MATE model.

9.2.3 CUDA Integration

MATE could be used to manage dependencies in GPU kernel execution and data movement between host and device memories. This functionality could be particularly useful in heterogeneous applications, where both the CPU and GPU resources are employed together. Furthermore, a programmer can integrate these GPU operations with UPC++ and MPI communication operations to develop a distributed heterogeneous application entirely scheduled by MATE.

9.2.4 Global Variables Handling

An outstanding problem with thread-based approaches is that static and global variables become shared among threads or ranks living in the same process. In standard MPI libraries,

each rank executes in a separate memory space and has exclusive access of its global and static variables. However, in thread-based models, these variables are accessible by all the ranks in the process, causing incorrect behavior due to unintended data sharing. Prior work has explored automated solutions [120, 20] to solve this problem. Each of these solutions has their pros and cons. Thus, a general purpose solution may be unfeasible.

9.2.5 Lightweight Translation

We currently use the ROSE compiler framework to parse MATE annotations and perform the translation steps (see Chapter 4) required by the model. However, our annotation syntax is relatively simple, requiring only a handful of modifications to the source code. Therefore, a robust framework like ROSE may not be necessary. ROSE can take several minutes to translate the source code for some complex codes, such as mpix_flowcart, deterring developers from using our model.

We are currently considering develping a custom text-based parser. MATE annotations only require a stack-based parser that can identify basic blocks and pragma annotations. We expect that this approach will simplify MATE's installation process and will significantly reduce translation times.

9.2.6 Support Fortran Annotations

Although the MATE Runtime System's API provides bindings for function calls from C/C++/Fortran code, our current translator can only process user-defined directives from C/C++ source. This limitation is imposed by the ROSE compiler framework –ROSE support Fortran code–, nor our annotation model (which is language-netural), but instead from the lack of support in the MATE translator. To support Fortran/MATE applications, we can either adapt our existing translator to integrate pragma parsing from Fortran code.

9.2.7 Support for Collective Communication Overlap

Although MATE supports MPI collective communication, we did not dedicate effort into optimizing these operations for performance. Collective communication did not represent a significant portion of running time in any of our test cases. However, other applications may require these optimizations. Prior work has proposed sophisticated algorithms for collective communication that employ minimal communication volume and overheads in MPI applications [77, 4, 87, 32]. Integrating these mechanisms into our hierarchically decomposed model is one of the main steps to consider in the future development of MATE.

9.2.8 Automate Graph Generation

The current annotation model requires that the programmer make sure that MATE directives define a correct and efficient dependency graph, which otherwise may result in deadlocks or incorrect results. Prior research gives systematic ways (*e.g.*, the *MPI-CFG* method [81]) to extract the underlying graph of an MPI application. We hope to integrate this approach into the MATE framework as a means to building an initial MATE graph that maintains the semantics of the MPI program while giving an initial graph from which the developer can refine parallelism.

Appendix A

Experimental Environment

A.1 Hardware Configuration

Table A.1 shows a side by side description of our two computational testbeds: Cori Phase I, and Cori Phase II.

Table A.1: Side by side description of our three computational testbeds.

	Cori Phase I [107]	**Cori Phase II**
Computer		
+ Architecture	Cray XC40	Cray XC40
+ Location	NERSC	NERSC
+ Node Count	2,388	9,688
+ Nodes/Cabinet	192	192
+ Node Performance	1.2 TFlops	3.0 TFlops
Processor		
+ Family	Intel *‘Haswell’*	Intel *‘Knights Landing’*
+ Count	2	1

+ Cores/Processor	16	64
+ Hyperthreads	2	4
+ Vector Units	2x256-bit	2x512-bit
+ Frequency	2.3Ghz	1.4 Ghz
+ Core Performance	36.8 GFlops	44.8 GFlop/s
Memory		
+ Technology	DDR4	DDR4
+ Node DIMMs	2x4x16GB	1x6x16GB
+ Node Capacity	128GB	96GB
+ L1 Cache	64 Kb	64Kb
+ L2 Cache	256Kb	1Mb (four-shared)
+ L3 Cache	40Mb/Socket	16GB MCDRAM*
Interconnect		
+ Architecture	Cray Aries[106]	Cray Aries
+ Topology	Dragonfly	Dragonfly
+ Global Peak Bandwidth	5.625 TB/s	45.0 TB/s

*The KNL processor on Cori Phase II provides two types of computing modes: *cache* mode employs MCDRAM as a last-level cache structure shared by all cores; *flat* mode, MCDRAM is available as addressable memory. We have used the *cache* mode for all our experiments.

A.2 Software Configuration

We configured the compilation environment with the following list of modules:

- modules/3.2.10.6
- nsg/1.2.0
- intel/18.0.1.163
- craype-network-aries
- craype/2.5.14
- pmi/5.0.13
- atp/2.1.1
- PrgEnv-intel/6.0.4
- craype-haswell **(only on Cori Phase I)**
- craype-mic-knl **(only on Cori Phase II)**
- cray-mpich/7.7.0
- craype-hugepages2M
- gcc/7.3.0
- upcxx/2018.9.0
- altd/2.0

For compiling our test cases, we use the following base set of flags:

- UPCXX_THREADMODE=par (Only for UPC++ variants)
- UPCXX_CODEMODE=O3 (Only for UPC++ variants)
- CFLAGS = -O3 -qopt-report=5 (performs and reports on auto-vectorization optimizations)
- CFLAGS += -mkl=sequential (To use mkl dgemm in Cannon's algorithm)
- CFLAGS += -restrict -fno-alias -fp-model strict -fp-model source (only for Cloverleaf3D)
- CFLAGS += -prec-div -prec-sqrt (only for Cloverleaf3D)

Appendix B

Code Optimizations

B.1 Cache Blocking

The solver kernel in Fig. 5.2 traverses the subgrid across its contiguous dimension (x) entirely before continuing to the next column on its non-contiguous y dimension. Similarly, it traverses across its y dimension before continuing to the next z-plane. The side-effect of traversing the grid in this fashion is that cache structures will fill cache lines with data to be read/written once and then immediately evicted. These evictions represent a loss of opportunity since elements in the y and z dimensions (required by the stencil) contained in cache-lines previously loaded will be required later when traversing neighboring columns/planes, and thus need to be reloaded into cache every at every access, incurring frequent cache misses. The constant eviction and re-loading of data produces a detrimental effect on the performance of the solver.

One way to improve cache locality is to employ a blocking strategy across the non-contiguous x and y dimensions. Fig. B.1 shows the pseudo-code of the solver kernel from Fig. B.1 modified with two additional *for* loops that iterate over the subgrid in increases of size *BSIZE*. The inner y and z loops only iterate within the *BSIZE* range with start points defined by the outer loops.

```
#define BSIZE 64
for (int zb = start.z; zb < end.z; zb += BSIZE)
 for (int yb = start.y; yb < end.y; yb += BSIZE)
  for (int z = zb; z < zb + BSIZE; z++)
   for (int y = yb; y < yb + BSIZE; y++)
    for (int x = start.x; x < end.x; x++)
    {
     Un[x,y,z]  = C0 *  U[x,y,z];
     Un[x,y,z] += C1 * (U[x+1,y,z] + U[x-1,y,z] +
                        U[x,y+1,z] + U[x,y-1,z] +
                        U[x,y,z-1] + U[x,y,z+1]);
     Un[x,y,z] += C2 * (U[x+2,y,z] + U[x-2,y,z] +
                        U[x,y+2,z] + U[x,y-2,z] +
                        U[x,y,z-2] + U[x,y,z+2]);
    }
```

Figure B.1: Cache-blocking Jacobi3D Pseudo-code.

Blocking the outer *for* loops enables the solver to store and iterate over a relatively small box of subgrid elements that fits entirely in the L2/L3 cache structures. The effect is that the elements loaded in the box persist in cache during their traversal and, therefore, enable the solver to reuse them before eviction. This reduction in cache-misses vastly improves the computational performance of the solver. We define the value of *BSIZE* empirically, tuning it to the size of each processor's cache capacity. We have found that the optimal size for *BSIZE* is 64 in all of our computational testbeds.

B.2 Vectorization

We compiled our test cases using the *-O3* flag which instructs the Intel compiler to enable auto-vectorization. A vectorized application uses floating-point vector instructions that apply an operation on multiple contiguous data elements simultaneously, instead of one element at a time. To apply auto-vectorization, the compiler imposes two conditions:

1. In the case of a *for* loop, none of the elements of the vector may depend on the execution of either the previous or next iteration. If that were the case, the vector operation could incur a RAW or WAR data hazard. Since we use pointer accesses inside our solver, the compiler cannot possibly determine whether or not there will be a violation, and therefore desists

from converting the kernel's arithmetic operations into their vectorized variants. However, since we read and write from/to different arrays (U and Un, respectively), we know that no violations will occur and therefore can inform the compiler that no data hazards exists by adding the *#pragma ivdep* directive.

2. The initial address to the elements in the vector needs to be aligned to a 64 bit boundary. Although this is the default alignment in our system's *glibc malloc* routine, we need to indicate the compiler that this indeed the case by adding the *#pragma vector aligned* directive.

Fig. B.2 shows the pseudo-code of the solver kernel with added vectorization-enabling directives.

```
1  #define BSIZE 64
2  for (int zb = start.z; zb < end.z; zb += BSIZE)
3  for (int yb = start.y; yb < end.y; yb += BSIZE)
4  for (int z = zb; z < zb + BSIZE; z++)
5  for (int y = yb; y < yb + BSIZE; y++)
6  #pragma vector aligned
7  #pragma ivdep
8  for (int x = start.x; x < end.x; x++)
9  {
10  Un[x,y,z]  = C0 *  U[x,y,z];
11  Un[x,y,z] += C1 * (U[x+1,y,z] + U[x-1,y,z] +
12                     U[x,y+1,z] + U[x,y-1,z] +
13                     U[x,y,z-1] + U[x,y,z+1]);
14  Un[x,y,z] += C2 * (U[x+2,y,z] + U[x-2,y,z] +
15                     U[x,y+2,z] + U[x,y-2,z] +
16                     U[x,y,z-2] + U[x,y,z+2]);
17 }
```

Figure B.2: Vectorized and cache-blocking Jacobi3D Pseudo-code.

B.3 Cubic Mapping

By default, MPI identifies ranks within a node with a contiguous range of identifiers. For example, in a 32-core system, node *0* executes MPI ranks 0 to 31, and node *1* executes MPI ranks 32 to 63. In distributing the workload of an algorithm, we could use this default

mapping to assign subgrids in a node assigning them across the x dimension, and advancing to the next y-dimension column or z-dimension plane when reaching the end of the row. Fig. B.3 illustrates this strategy across a single z-plane. Each of the eight cubes in a given node represents a different rank and its subgrid, and each arrow represents a message exchange over the network interconnect.

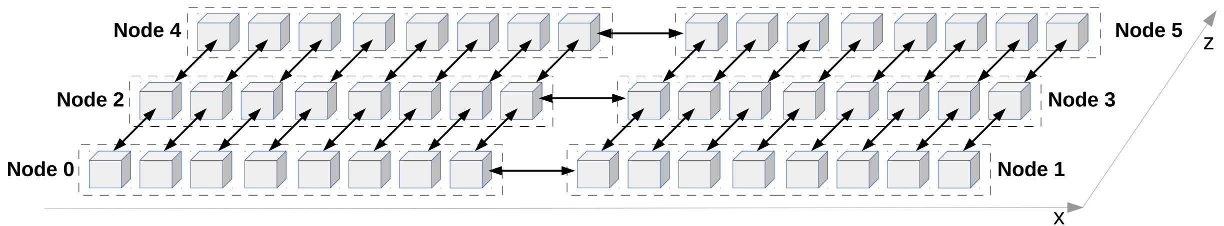

Figure B.3: Linear rank mapping across nodes.

The problem with a linear workload distribution is that processes located in the same node will only have local neighbors across the contiguous x dimension. Such a distribution implies that neighbors in the y and z dimensions will forcibly reside in a different node. Fig. B.3 shows that, with this distribution, each rank has six faces with only two of them neighboring with node co-located processes. As a consequence, every process in the node requires exchanging messages with different nodes on four of its faces. Therefore, this strategy maximizes the overhead of network inter-node communication.

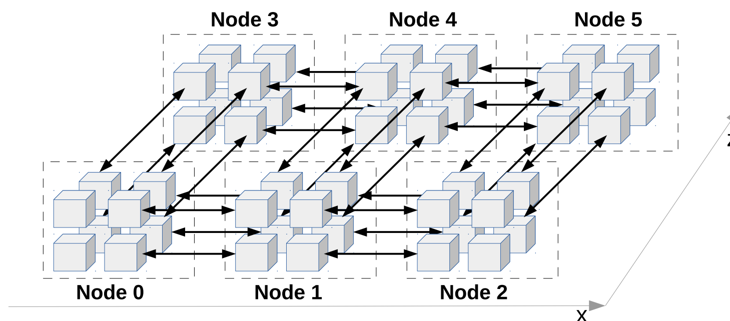

Figure B.4: Cubic rank mapping across nodes in Jacobi3D.

To reduce the volume of inter-node communication, we implement a cubic (or rectangular, in the case of Cannon's algorithm) mapping strategy that maximizes rank locality. We

allocate subgrids across the ranks of a node so that the configuration forms a 3D box that maximizes the number of sides faces neighboring local processes. These faces will communicate boundary data internally, reducing intra-node communication.

Fig. B.4 shows the effect of applying a cubic mapping. This mapping reduces the number of outgoing faces in each rank to three, instead of four. In general, larger boxes (*i.e.* the number of cores in a node) reduce even more of the volume of inter-node communication. For example, in a 4x4x4 grouping, ranks located in the center have all their faces neighboring node-local ranks.

Appendix C

MPI Concurrency Limitation

One of the main features of the MATE model is its ability to benefit from multi-threading to maximize opportunities for overlapping communication with computation. Ideally, a MATE application would instantiate a minimal number of MATE processes to span all the cores in a node or NUMA domain. However, there are certain limitations to the number of threads and ranks a single MATE process can span. One of them is cache efficiency; constantly migrating ranks may cause a loss in the locality of data. However, we have found that the major liming factors in taking advantage of multi-threading lays in the implementation of the MPI communication backend. In this appendix, we explore two main problems: *thread serialization*, and *bandwidth saturation*.

C.1 Problem 1: Thread Serialization

The first problem is that the current implementation of the Cray-MPICH library (and others available in NERSC supercomputers) are unable to handle concurrent communication operations efficiently. Conventional MPI libraries allocate a single process-wide set of structures to manage send/recv requests and monitor their progress. While this configuration works well for flat MPI applications (*i.e,* those running a single thread per process, one process per

core), it does not allow multiple threads to perform communication operations simultaneously. Instead, threads need to serialize their execution when creating or polling for send/recv requests. Fig. C.1 shows multiple levels that a programmer can specify during initialization (via: *MPI_Init_thread(level)*) to enable multi-threading in an MPI application:

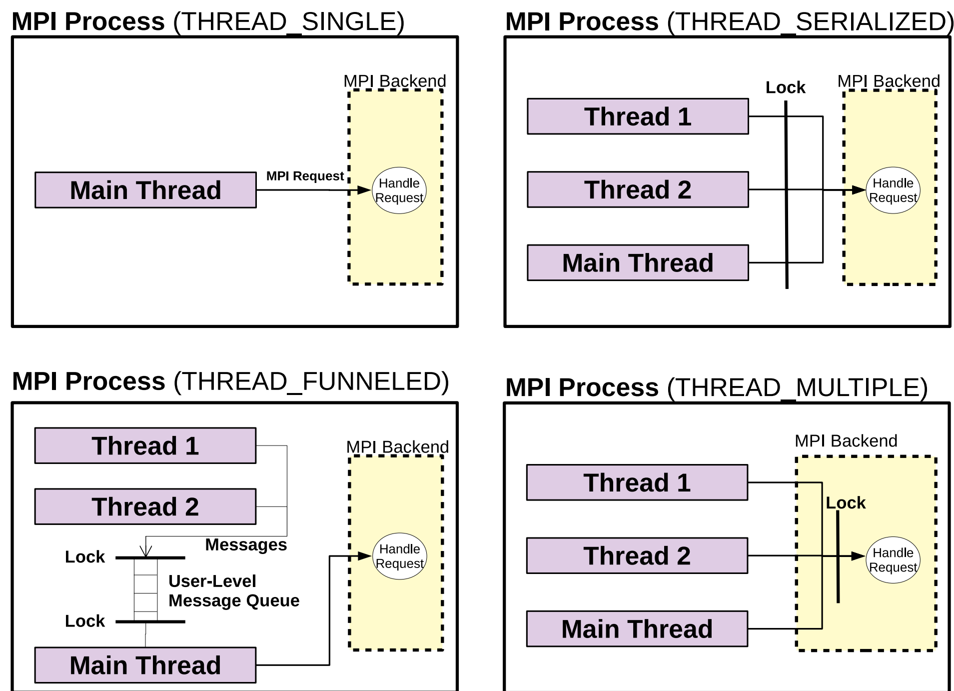

Figure C.1: Different level of multi-threading in MPI.

- *MPI_THREAD_SINGLE* instructs the MPI backend that there will only be a single thread in execution in the current process. Providing this level equates to a normal MPI_Init() initialization.
- *MPI_THREAD_FUNNELED* instructs the MPI backend that, although multiple threads may be created during runtime, only the *main* thread will perform MPI operations..
- *MPI_THREAD_SERIALIZED* instructs the MPI backend that multiple threads will be created, all of them will perform concurrent MPI operations. This mode, however, requires that only a single thread can operate on the MPI library at a time. Therefore, it requires the use of a mutual exclusion mechanism (*e.g., mutex*) on the application's side to prevent

concurrent access. Performing concurrent accesses may result in unpredictable/erroneous behavior.

- *MPI_THREAD_MULTIPLE* instructs the MPI backend that multiple threads will be created, all of them will perform concurrent MPI operations. Although this mode allows applications threads to access the MPI backend simultaneously, current implementations of MPI implement a process-wide lock[1] in the library's side to prevent concurrent access to communication structures. This mode is similar in practice as *MPI_THREAD_ SERIALIZED*, except that it does not require the use of mutual exclusion locks on the application side.

We can see that, although MPI libraries support multi-threading applications, they still enforce serialized access to communication resources, causing significant computational overheads. The adverse effect of serialization in multi-threaded MATE applications is threefold. First, it reduces core usage efficiency since cores are stalled for longer times just waiting to gain access to MPI's process-wide lock, whereas they could be performing useful computation instead. These stalls occur each time they need to issue a new send/recv operation. Second, serialization forces us to limit the number of threads and ranks per process which in turn, reduces opportunities for communication/computation overlap. Third, the additional per-message waiting time causes an increase in their latency, spreading the delay across both sender and receiver ranks. This overhead especially affects overdecomposed MATE applications since they require issuing a larger number of MPI messages.

C.2 Problem 2: Bandwidth Saturation

The second problem with MPI's concurrency limitation is that it does not provide a sufficient number of channels for cores to saturate the bandwidth of the *Network Interface Card*

[1] Various efforts are underway to provide smarter implementations that do not rely on process-wide locks.

(NIC). The Cray XC Series (Aries) networks, which are employed in our three testbeds (see chapter: A), provides two main data transfer protocols [6]: the *Block Transfer Engine* (BTE) protocol, and the *Fast Memory Access* protocol.

BTE uses an asynchronous protocol for transfering data between local and remote memory. A communication library accesses BTE channels by pushing a *put/get* communication request onto the NIC's BTE queue, specifying the size and address of the local/remote buffers. After submitting the request, execution control returns immediately to the calling application. The NIC will poll the BTE queue to feed its four concurrent channels, which are shared among all the processes in the node. Due to its asynchronous operation, BTE does not require CPU intervention, thus providing a high-latency/high-bandwidth data transfers, which is better suited to large messages.

The FMA protocol, on the other hand, is mediated entirely by the CPU through I/O operations into the NIC. These operations start immediately, yielding a very low per-message latency. However, since the CPU is in charge of actively moving data from RAM to the NIC, the FMA protocol occupies the core and has a lower bandwidth than the BTE protocol. For this reason, the FMA is better suited for small messages. Fig. C.2 shows how the MPI backend automatically selects which channel to use when sending a message, based on a message-size threshold, called the *eager limit* (by default set to around *30kb*, although users can define this value through an environment variable)[2]

When the message size is lower than the eager limit threshold, MPI employs an *eager* communication strategy. This strategy uses the FMA protocol to copy the data to the remote process, even if no reciprocal *recv* request has arrived yet. The receiving process will store the incoming data in an intermediate buffer until the *recv* request is posted. Eager communication reduces per-message latency at the cost of low bandwidth and a potential extra copy at the receiving end. This cost is affordable for small messages, but can be punishing for larger messages.

[2]This is a representation of how the current version of Cray-MPICH works under the Cray Aries network. Other MPI libraries may use different thresholds and configurations on different networks.

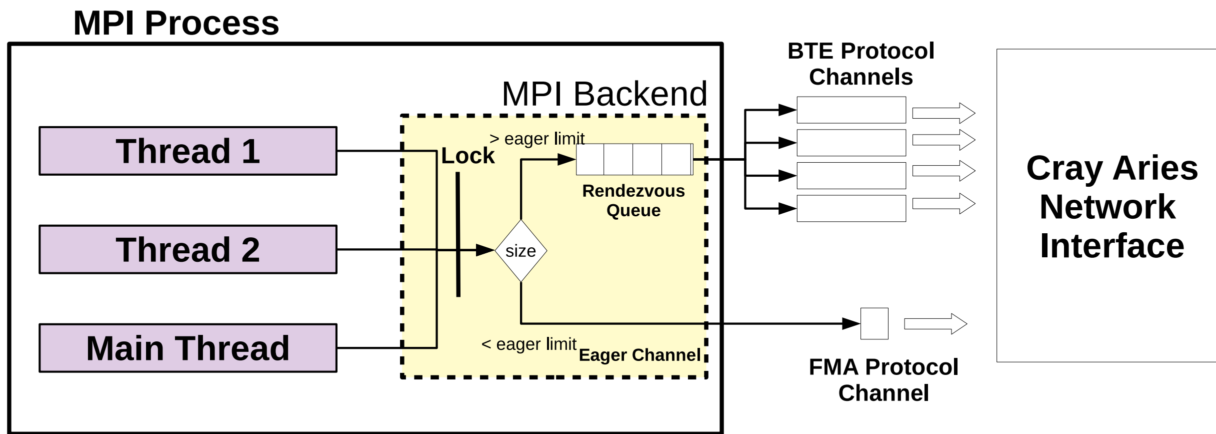

Figure C.2: Rendezvous (BTE protocol) and Eager (FMA protocol) strategies employed by the MPI library.

In the case of larger messages, MPI uses a *rendesvous* strategy. This strategy queues *send* requests and holds them at the source until a reciprocal *recv* request has been posted. At the destination, MPI pushes the message into the NIC's BTE protocol queue (which will occupy one of the four channels) as soon as it becomes ready. This approach has a higher latency due to waiting for reciprocal requests and the use of the high-latency BTE protocol. However, the higher bandwidth of this channel compensates for its latency cost with larger messages.

The fundamental characteristics of these protocols have different implications for multi-threaded applications. Since the NIC provides four BTE channels at each node, and application needs to perform at least four concurrent operations simultaneously to saturate the NIC's bandwidth capacity. FMA channels, however, are not as easy to saturate. Since MPI serializes access to its backend via mutual exclusion locks, only one thread can access the FMA channel at a time. For conventional MPI applications, where each core has its dedicated channel, this is not a problem since all of them (at most four) can potential supply the NIC. For MATE applications, however, only one core per process can feed the FMA channel at a time, failing to saturate the NIC's bandwidth. The consequence of serializing FMA operations is hugely punishing, especially for applications exchanging mostly small messages, such as *mpix_flowCart* (chapter 8).

C.3 Experimental Tests

To verify the adverse effects of concurrency in MATE, we developed a benchmark based on our *Jacobi3D* test case (see chapter: 5). In this benchmark, we added a *multiplier* argument (m) to split boundary exchange messages into multiple, smaller messages. For example, by specifying $m = 2$, each rank will send twice as many messages of half the size each to their neighbors. Defining $m = 1$ will result in a base case execution where each message contains a full boundary. Regardless of the value of the multiplier argument, the benchmark will produce the same correct result as the base case.

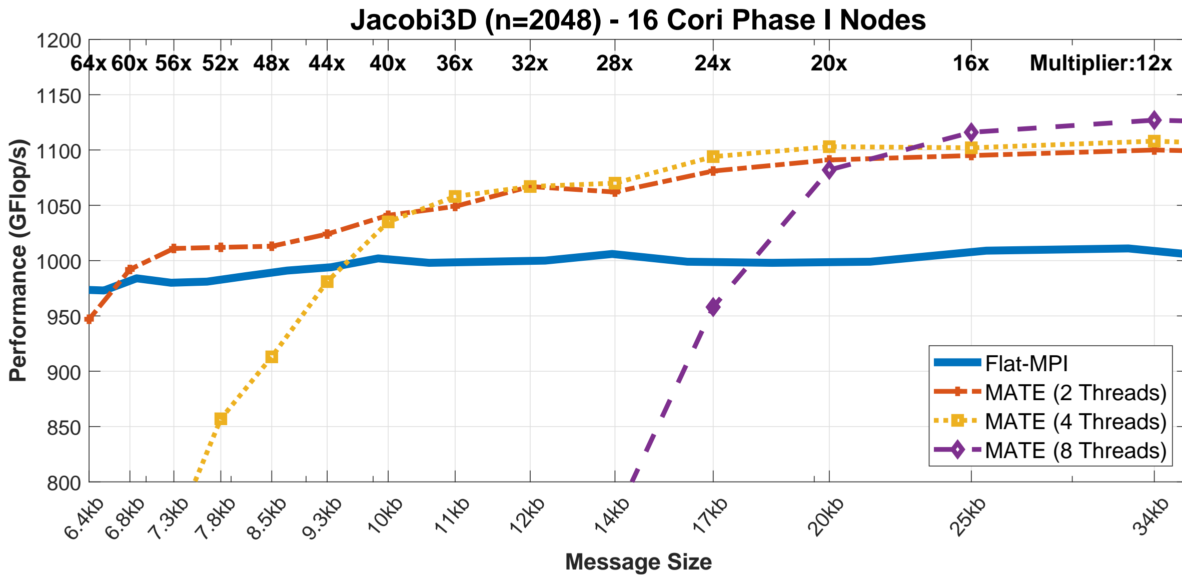

Figure C.3: Performance of the baseline Flat-MPI compared to multi-threaded MATE variants, as message sizes decrease and total message count increase.

The goal of this experiment is to measure the performance degradation in both MPI and multiple levels of threading in MATE. We ran our tests on 16 nodes (512 cores) of the Cori Phase I platform to solve a 3D grid of $n = 2048$ elements per side. Fig. C.3 shows the best performance of Flat-MPI and MATE (2, 4, and 8 threads), using a range of $m = 64$ (6.4kb) to $m = 12$ (34kb). Since we observed a stable performance of every variant between the range of $m = 1$ to $m = 12$, we omitted those results from the figure.

We see that the performance of *Flat-MPI* (32 processes per node) is stable throughout the ranges of m, with only a small degradation towards the large values of m. We attribute this effect to increased overheads in handling more numerous messages. On the other hand, the best performing variant of all, MATE (8 Threads), suffers from a quick degradation starting around $\sim 30kb$ and towards smaller sizes. This threshold coincides with the 28kb threshold in which we measured MPI switch from the rendezvous strategy (BTE protocol) to the eager strategy (FMA protocol).

MATE (8 Threads) provides more opportunities for communication/computation overlap than the other variants since it contains the highest worker and rank counts, and therefore reduces the amount of intra-node communication, improving performance. However, only four processes are created, suffering from a steeper degradation since eight core per process serialize their access to the FMA channel. As the number of messages increase, this variant is increasingly unable to saturate the bandwidth of the FMA channel fully. The serialization effect is less destructive with 8 MATE processes they only instantiate 4 threads each and have more channels to saturate the NIC's bandwidth. Finally, the MATE (2 Threads) variant suffers a much slower degradation, performing slower than MPI only as message sizes reach 6.4kb.

Our results show that an MPI-based application requires as many concurrently communicating processes as possible to keep a high bandwidth saturation. Similar results have been presented in a study by Doerfler *et al.* [37]. The authors use the *Sandia MPI Micro-benchmark* (*msgrate*) [113] on Cori Phase II (Cray Aries) to measure the network bandwidth saturation between any two nodes by continuously issuing simultaneous messages send and receive operations of varying sizes (from 64 bytes to 1Mb). The study shows experiment results when using 1, 2, 4, 8, 16, 32, and 64 MPI processes per node. Their results show a stable saturation from 1MB down to 32kb for all variants, with 64 MPI processes achieving the best saturation. Below 32kb, saturation levels suffer a steep drop for the smaller rank-per-node experiments due to their incapability to saturate the FMA channel. On the other hand, experiments with larger process

count have a slower degradation. These results closely match our observations.

C.3.1 Possible Solution

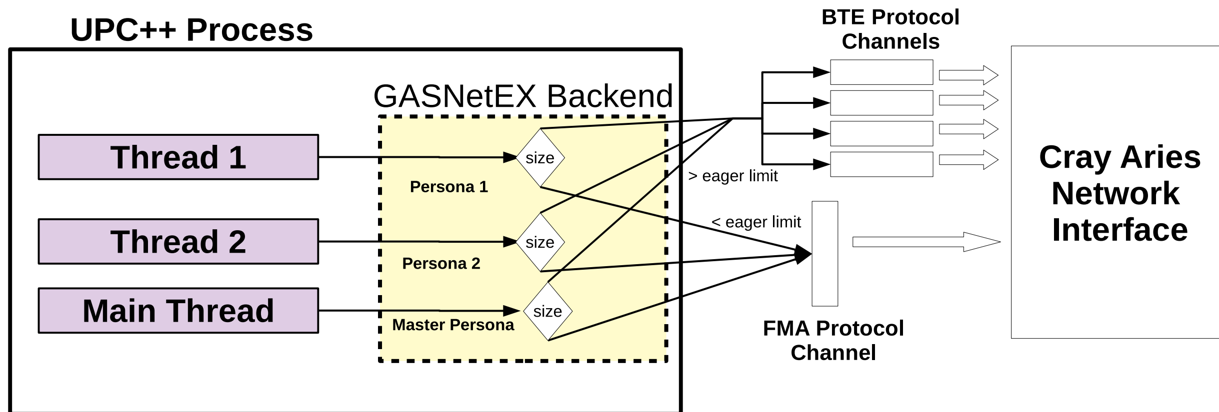

Figure C.4: Multi-threaded UPC++ can own different personas to avoid the need for process-wide mutual exclusion mechanisms.

A solution for our threading concurrency problems could be to use UPC++ as communication backend. UPC++ provides support for multiple personae (see Fig. C.4), allowing multiple threads to operate concurrently without the need of a process-wide lock. Each persona represents a separate collection of communication operations that can be assigned to any thread, one thread at a time. While owning a persona, a thread is less likely to suffer concurrency issues and can access the communication backend without any process-wide mutual exclusion. Since multiple threads can access the backend, multi-threaded UPC++ applications could saturate the NIC's FMA channels just as optimally as a non-threaded UPC++ or MPI application.

Appendix D

Load Balancing Algorithms

D.1 Consecutive Rebalancer

The consecutive balancing algorithm[1] distributes workload by moving partition boundaries without changing their relative order of the partition. Fig. D.1 shows the MATLAB pseudo-code for this algorithm. The input argument *MateProcessCount* indicates the number of MATE processes, while the *ComputeTimes* argument is an array with the per-rank execution times. This algorithm assumes that every process executes the same number of ranks. The output argument *ProcessRankMapping* is an array indicating the new number of ranks for each process.

This algorithm divides into two phases. In phase I, the algorithm finds the maximum optimal workload that can be assigned to each process. This phase starts defining a lower and an upper bound for the maximum load. The lower bound is initialized as the largest computation time among all ranks, which defines the minimum possible load for any given MATE process. The upper bound is initialized as the sum of all ranks, which is the maximum possible load. The solver will approach the optimal solution logarithmically by halving the upper bound (or increasing the lower bound). The optimal solution is the minimum load for every process that

[1]This algorithm is an adaptation of the solution to the Painter's Partitioning Problem provided in [101].

requires a minimum number of process equal to *MateProcessCount* and allocates every rank. During Phase II, the algorithm assigns ranks to processes in consecutive order, stopping only before exceeding the maximum load determined in Phase I.

```
1  function [ProcessRankMapping] = consecutiveBalancer(MateProcessCount, ComputeTimes)
2  % Phase I: Find Maximum Process Workload
3  rankCount            = size(ComputeTimes,1);
4  [lowerBound, maxIdx] = max(ComputeTimes);
5  upperBound           = sum(ComputeTimes);
6
7  while lowerBound < upperBound
8   midPoint = (lowerBound + upperBound)/2;
9   minProcessCount = 1;
10  total = 0;
11  for i = 1:rankCount
12   total = total + ComputeTimes(i);
13   if total > midPoint
14    total = ComputeTimes(i);
15    minProcessCount = minProcessCount + 1;
16   end
17  end
18  if minProcessCount <= MateProcessCount
19   upperBound = midPoint;
20  else
21   lowerBound = midPoint+0.001;
22  end;
23 end
24
25 maxLoad = lowerBound;
26
27 % Phase II: Find the optimal consecutive Rank->Process Mapping
28 currentRank = 1;
29 ProcessRankMapping = zeros(MateProcessCount,1);
30 for i = 1:MateProcessCount
31  processSum = 0;
32  while currentRank <= rankCount && processSum + ComputeTimes(currentRank) <= maxLoad
33   processSum = processSum + ComputeTimes(currentRank);
34   currentRank = currentRank + 1;
35   ProcessRankMapping(i) = ProcessRankMapping(i) + 1;
36 end; end; end
```

Figure D.1: Consecutive Balancing Algorithm.

D.2 Shuffling Rebalancer

We developed a shuffling balancing algorithm that works by smoothing peak workloads inside each node, exchanging and shuffling ranks among each MATE processes. Fig. D.2 shows the MATLAB pseudo-code of this algorithm. The input argument *ProcessRankMap* is a matrix that contains a row for each MATE process inside a node, a column for every rank (up to the

maximum number of ranks), and each entry contains the id of the rank. Processes with less than the maximum number of ranks, pad their row with *zero* entries. The *ComputeTimes* argument represents an array with the per-rank execution times. This algorithm assumes that every process executes the same number of ranks. The output argument *newMap* is a re-balanced mapping of the one provided in *ProcessRankMap*.

This algorithm iteratively improves the input rank-process distribution through reshuffling transformations. First, it calculates the initial per-process workload into the *processCost* variable. Next, it iteratively improves the workload distribution by swapping a rank in the most heavily loaded process with a rank in another process, such that the swap reduces the maximum workload over all processes. Each iteration costs $O(P^2r^2)$, where P is the number of processes, and r is the number of ranks.

```
1  function [newMap] = shufflingBalancer(ProcessRankMap, ComputeTimes)
2  processCount = size(ProcessRankMap,1); maxRankIdx =  size(ProcessRankMap,2);
3  processCost = zeros(processCount,1); newMap = ProcessRankMap;
4  for i = 1:processCount;  for j = 1:maxRankIdx;  if newMap(i,j) ~= 0
5   processCost(i) = processCost(i) + ComputeTimes(newMap(i,j));
6  end; end; end
7
8  % Start Intra-Node Shuffling
9  for iters = 1:50; for j = 1:processCount; for k = 1:processCount;  if j < k
10     maxProcCost = max(processCost(j), processCost(k));
11     exchanged = false;
12     for m = 1:maxRankIdx
13      if exchanged == true || newMap(j,m) == 0; break; end
14      for n = 1:maxRankIdx
15       if exchanged == true || newMap(k,n) == 0; break; end
16       newProcJCost = processCost(j)-ComputeTimes(newMap(j,m))+ComputeTimes(newMap(k,n));
17       newProcKCost = processCost(k)-ComputeTimes(newMap(k,n))+ComputeTimes(newMap(j,m));
18       if (newProcJCost < maxProcCost)
19       if (newProcKCost < maxProcCost)
20        tmp = newMap(k,n); newMap(k,n) = newMap(j,m); newMap(j,m) = tmp;
21        processCost(j) = newProcJCost;  processCost(k) = newProcKCost;
22        exchanged = true;
23 end; end; end; end; end; end; end; end
```

Figure D.2: Shuffling Balancing Algorithm.

Appendix E

MATE Application Programming Interface

The MATE API (Application Programming Interface) comprises a set of functions that can be called from a MATE-translated application. We expose three interfaces: The *MATE Model* interface, the *Runtime System* interface, and the supported MPI functions interface.

E.1 MATE Model Interface

The MATE Model Interface is a set of function that guide the execution of ranks under MATE's 2-level hierarchical decomposition model, synchronize and exchange information among local ranks, and measure the performance of MATE ranks and workers. A MATE-application inserts these calls into the code directly before translation.

- *void Mate_local_rank_id(int* localRankId)*

  Output: *localRankId*, local identifier of the calling rank.
- *void Mate_local_rank_count(int* localRankCount)*

  Output: *localRankCount*, number of ranks in the process.
- *void Mate_global_process_id(int* globalProcessId)*

  Output: *globalProcessId*, process identifier of the calling rank.

- *void Mate_global_process_count(int* globalProcessCount)*

  Output: *globalProcessCount*, number of MATE processes.

- *void Mate_LocalBcast(void* ptr, size_t size, int local_root)*

  Broadcast a message among all the local ranks in the current MATE process.

  Input: *ptr*, source pointer (for local root rank); *size*, number of bytes to exchange; *local_root*, local id of the root rank.

  Output: *ptr*, destination pointer (for non-local root ranks).

- *int Mate_LocalBarrier())*

  Blocks local ranks in the MATE process until all of them have reached this function.

  Output: *(return value)*, always returns *MPI_SUCCESS*.

- *void Mate_SetFirstLocalTime()*

  Registers the clock time of the first local rank to reach this point. This is useful for process-wide time measurements.

- *void Mate_SetLastLocalTime()*

  Registers the clock time of the last local rank to reach this point. This is useful for process-wide time measurements.

- *double Mate_GetProcessTime()*

  Returns the clock time difference between local ranks reaching the designated first and last time functions.

  Output: *(return value)*, a time lapse in seconds.

- *void Mate_AddLocalNeighbor(int localRankId)*

  Registers all inter-rank dependencies to the specified local rank id.

  Input: *localRankId*, the local identifier of the new rank neighbor.

- *void Mate_local_thread_count(int* localThreadCount)*

  Output: *localThreadCount*, number of local worker threads.

- *void Mate_global_thread_count(int* globalThreadCount)*

Output: *globalThreadCount*, number of global worker threads.

- *Mate_StartTimers()*

  Starts all MATE internal clocks for time measurements.

- *Mate_StopTimers()*

  Stops all MATE internal clocks.

- *Mate_ResetTimers()*

  Resets all MATE internal clocks.

- *void Mate_GetGlobalThreadScheduleTime(double* scheduleTimes)*

  Determines the total time that each MATE worker spent waiting for communication across all MATE processes.

  Output: *scheduleTimes*, an array of size equal to the number of all the MATE workers in execution.

- *void Mate_GetGlobalThreadExecuteTime(double* computeTimes)*

  Determines the total time that each MATE worker spent executing ranks across all MATE processes.

  Output: *computeTimes*, an array of size equal to the number of all the MATE workers in execution.

- *vector<double>* Mate_GetLocalThreadTimeLapses(int threadId)*

  Returns the duration of every separate operation (execute, wait) that a local thread went through during execution.

  Input: *threadId*, the id of a local MATE worker.

  Output: *computeTimes*, the pointer to an array of doubles containing the duration of each operation.

- *vector<int>* Mate_GetLocalThreadTimeTaskIds(int threadId)*

  Returns the operation type per each time lapse executed by the thread.

  Input: *threadId*, the id of a local MATE worker.

Output: *computeTimes*, the pointer to an array of ints containing an operation type. A *-1* value, indicates a waiting operation, while a > 0 value indicates an rank execution with the number indicating the rank's id.

E.2 Runtime System Interface

The Runtime System Interface is a set of functions that describe and guide the execution of graph block regions. This interface is exposed to the MATE Translator and is required to interpret MATE *graph* and *region* directives. We have also implemented a Fortran 90 binding for each of these calls. MATE-application programmers do not need to add calls to any of these functions, unless translation is not possible (*e.g.* working with a *fortran* application).

- *void Mate_AddRegion(int regionId)*

  Input: *regionId*, the position in the rank's regions array to allocate the new region.

  Fortran hook: *void mate_addregion_(INTEGER)*

- *void Mate_EnableRegion(int regionId)* Input: *regionId*, the position in the rank's regions array to enable.

  Fortran hook: *void mate_enableregion_(INTEGER)*

- *void Mate_DisableRegion(int regionId)*

  Input: *regionId*, the position in the rank's regions array to disable.

  Fortran hook: *void mate_disableregion_(INTEGER)*

- *Mate_InterRank(int srcRegionId, int depRegionId, int delay)*

  Input: *srcRegionId*, the id of the dependent region. *depRegionId*, the id of the depended region. *delay*, the step number of the dependent region in which the dependency becomes active.

  Fortran hook: *void mate_interrank_(INTEGER, INTEGER, INTEGER)*

- *Mate_AddDependency(int srcRegionId, int depRegionId, int delay)*

Input: *srcRegionId*, the id of the dependent region. *depRegionId*, the id of the depended region. *delay*, the step number of the dependent region in which the dependency becomes active.

Fortran hook: *void mate_adddependency_(INTEGER, INTEGER, INTEGER)*

- *int Mate_GetNextRegionID()*

  Output: *(return value)*, the id of the region to execute.

  Fortran hook: *void mate_getregionid_()*

- *void Mate_TaskMapping(int* nTasksPerProcess, int **taskMapping)*

  Remaps the assignment of ranks to each MATE process with a custom number and global Ids of MATE ranks.

  Input: *nTasksPerProcess*, an integer array of size equal to the number of MATE process in execution indicating how many ranks each process will contain.

  taskMapping, an array of integer arrays of size indicated by nTasksPerProcess that contain the global ids of the ranks for each MATE process.

E.3 Supported MPI Functions

Our current implementation of the MATE runtime system supports the following MPI functions: MPI_Init, MPI_Finalize, MPI_Barrier, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Reduce, MPI_Bcast, MPI_Isend, MPI_Send, MPI_Irecv, MPI_Recv, MPI_Wait, MPI_Waitall, MPI_Comm_size, MPI_Comm_rank, MPI_Abort, MPI_Wtime, MPI_Address, MPI_Get_count, MPI_Op_create, MPI_Type_contiguous, MPI_Type_vector, MPI_Type_struct, MPI_Type_create_struct, MPI_Type_commit, MPI_Type_indexed, MPI_Type_free, MPI_Type_size, MPI_Pack, and MPI_Unpack

Bibliography

[1] ADAMS, D. A. *A Computation Model with Data Flow Sequencing*. PhD thesis, Stanford, CA, USA, 1969. AAI6913919.

[2] ADAMS, M., BROWN, J., SHALF, J., STRAALEN, B. V., STROHMAIER, E., AND WILLIAMS, S. HPGMG 1.0: a benchmark for ranking high performance computing systems.

[3] AFTOSMIS, M., BERGER, M., AND ADOMAVICIUS, G. A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries. In *AIAA'00*.

[4] ALMÁSI, G., HEIDELBERGER, P., ARCHER, C. J., MARTORELL, X., ERWAY, C. C., MOREIRA, J. E., STEINMACHER-BUROW, B., AND ZHENG, Y. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing* (2005), ACM, pp. 253–262.

[5] ALMGREN, A., BECKNER, V., ET AL. CASTRO: A new compressible astrophysical solver. I. Hydrodynamics and self-gravity. *The Astrophysical Journal 715*, 2 (2010), 1221.

[6] ALVERSON, B., FROESE, E., KAPLAN, L., AND ROWETH, D. Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112* (2012).

[7] AMARASINGHE, S., CAMPBELL, D., ET AL. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009), 1–153.

[8] ARVIND, AND NIKHIL, R. S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers 39*, 3 (Mar 1990), 300–318.

[9] ASHBY, S., BECKMAN, P., CHEN, J., COLELLA, P., COLLINS, B., CRAWFORD, D., DONGARRA, J., KOTHE, D., LUSK, R., MESSINA, P., ET AL. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee* (2010), 1–77.

[10] AUGUIN, M., AND LARBEY, F. OPSILA: an advanced SIMD for numerical analysis and signal processing. In *Microcomputers: developments in industry, business, and education,*

Ninth EUROMICRO Symposium on Microprocessing and Microprogramming, Madrid, September 13 (1983), vol. 16, pp. 311–318.

[11] BABB, R.G., I. Parallel Processing With Large-Grain Data Flow Technique. *Computer 17*, 7 (July 1984), 55–61.

[12] BAK, S., MENON, H., WHITE, S., DIENER, M., AND KALÉ, L. V. Multi-Level Load Balancing with an Integrated Runtime Approach. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018* (2018), pp. 31–40.

[13] BALAJI, P., BUNTINAS, D., GOODELL, D., GROPP, W., KUMAR, S., LUSK, E., THAKUR, R., AND TRÄFF, J. L. MPI on a Million Processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2009), M. Ropo, J. Westerholm, and J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 20–30.

[14] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[15] BALLARD, G., CARSON, E., DEMMEL, J., HOEMMEN, M., KNIGHT, N., AND SCHWARTZ, O. Communication lower bounds and optimal algorithms for numerical linear algebra. 1–155.

[16] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 66:1–66:11.

[17] BHANDARKAR, M., KALE, L. V., DE STURLER, E., AND HOEFLINGER, J. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074* (May 2001), pp. 108–117.

[18] CALVIN, C. Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication. *Parallel Computing 22*, 9 (1996), 1255 – 1279.

[19] CANNON, L. E. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[20] CARRIBAULT, P., PÉRACHE, M., AND JOURDREN, H. Thread-local storage extension to support thread-based MPI/openMP applications. In *International Workshop on OpenMP* (2011), Springer, pp. 80–93.

[21] CAVÉ, V., ZHAO, J., SHIRAKO, J., AND SARKAR, V. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (2011), ACM, pp. 51–61.

[22] CHAIMOV, N., IBRAHIM, K. Z., WILLIAMS, S., AND IANCU, C. Exploiting Communication Concurrency on High Performance Computing Systems. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (New York, NY, USA, 2015), PMAM '15, ACM, pp. 132–143.

[23] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not. 40*, 10 (Oct. 2005), 519–538.

[24] CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the* (1992), IEEE, pp. 120–127.

[25] CICOTTI, P. *Tarragon: A Programming Model for Latency-hiding Scientific Computations*. PhD thesis, University of California at San Diego, CA, USA, 2011. AAI3449479.

[26] CICOTTI, P., AND BADEN, S. B. Asynchronous Programming with Tarragon. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[27] COOLEY, J. W., AND TUKEY, J. W. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation 19*, 90 (1965), 297–301.

[28] COSTA, G. D., FAHRINGER, T., ET AL. Exascale Machines Require New Programming Paradigms and Runtimes. *Supercomputing Frontiers and Innovations 2*, 2 (2015).

[29] CUI, Y., OLSEN, K. B., ET AL. Scalable earthquake simulation on petascale supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–20.

[30] CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. LogP: Towards a realistic model of parallel computation. In *ACM Sigplan Notices* (1993), vol. 28, ACM, pp. 1–12.

[31] CULLER, D. E., KRISHNAMURTHY, A., DUSSEAU, A., GOLDSTEIN, S. C., LUMETTA, S., VON EICKEN, T., AND YELICK, K. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1993), Supercomputing '93, ACM, pp. 262–273.

[32] DANALIS, A., KIM, K.-Y., POLLOCK, L., AND SWANY, M. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Nov 2005), pp. 58–58.

[33] DAREMA, F. The SPMD Model: Past, Present and Future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2001), Y. Cotronis and J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 1–1.

[34] DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE Press, p. 4.

[35] DEBUDAJ-GRABYSZ, A., AND RABENSEIFNER, R. Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2005), B. Di Martino, D. Kranzlmüller, and J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 18–27.

[36] DENNIS, J. Data Flow Supercomputers. *IEEE Computer 13*, 11 (1980), 48–56.

[37] DOERFLER, D., AUSTIN, B., COOK, B., DESLIPPE, J., KANDALLA, K., AND MENDYGRAL, P. Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC.

[38] DONGARRA, J., BECKMAN, P., MOORE, T., AERTS, P., ALOISIO, G., ANDRE, J.-C., ET AL. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl. 25*, 1 (Feb. 2011), 3–60.

[39] DUBROW, A. Supercomputers Assist in Search for New, Better Cancer Drugs. *https://www.tacc.utexas.edu/-/supercomputers-assist-in-search-for-new-better-cancer-drugs* (2017).

[40] FLYNN, L. J. Intel halts development of 2 new microprocessors. *The New York Times 8* (2004).

[41] FÜRER, M. Faster integer multiplication. *SIAM Journal on Computing 39*, 3 (2009), 979–1005.

[42] GAULTIER, S. ONERA-M6 Wing, Star of CFD. *https://www.onera.fr/ en/news/onera-m6-wing-star-of-cfd* (2013).

[43] GEORGANAS, E., BULU, A., CHAPMAN, J., HOFMEYR, S., ALURU, C., EGAN, R., OLIKER, L., ROKHSAR, D., AND YELICK, K. HipMer: an extreme-scale de novo genome assembler. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2015), pp. 1–11.

[44] GEORGANAS, E., GONZÁLEZ-DOMÍNGUEZ, J., SOLOMONIK, E., ZHENG, Y., TOURIÑO, J., AND YELICK, K. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 100:1–100:11.

[45] HEMSOTH, N. The Supercomputing Strategy That Makes Airbus Soar. *https://www.nextplatform.com/2015/07/22/the-supercomputing-strategy-that-makes-airbus-soar/* (2015).

[46] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[47] HILBERT, D. Über die stetige Abbildung einer Linie auf ein Flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*. Springer, 1935, pp. 1–2.

[48] HOEFLER, T., DINAN, J., BUNTINAS, D., BALAJI, P., BARRETT, B., BRIGHTWELL, R., GROPP, W. D., KALE, V., AND THAKUR, R. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. *Computing 95* (2013), 1121–1136.

[49] HUANG, C., LAWLOR, O., AND KALÉ, L. V. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958* (College Station, Texas, October 2003), pp. 306–322.

[50] IANCU, C., HOFMEYR, S., BLAGOJEVI, F., AND ZHENG, Y. Oversubscription on multicore processors. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (April 2010), pp. 1–11.

[51] J. QUINLAN, D. ROSE: Compiler Support for Object-Oriented Frameworks. 215–226.

[52] KALÉ, L. V. The virtualization approach to parallel programming: Runtime optimizations and the state of the art. In *Los Alamos Computer Science Institute Symposium-LACSI* (2002).

[53] KALE, L. V., AND KRISHNAN, S. 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* (New York, NY, USA, 1993), OOPSLA '93, ACM, pp. 91–108.

[54] KAMAL, H., AND WAGNER, A. FG-MPI: Fine-grain MPI for multicore and clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)* (April 2010), pp. 1–8.

[55] KAMIL, S., OLIKER, L., PINAR, A., AND SHALF, J. Communication requirements and interconnect optimization for high-end scientific applications. *IEEE Transactions on Parallel and Distributed Systems 21*, 2 (2010), 188–202.

[56] KIM, J., BALFOUR, J., AND DALLY, W. Flattened Butterfly Topology for On-Chip Networks. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)* (Dec 2007), pp. 172–182.

[57] KIM, J., DALLY, W. J., SCOTT, S., AND ABTS, D. Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 International Symposium on Computer Architecture* (June 2008), pp. 77–88.

[58] KOWARSCHIK, M., AND WEISS, C. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*. Springer, 2003, pp. 213–232.

[59] KRISHNAMURTHY, A., CULLER, D. E., DUSSEAU, A., GOLDSTEIN, S. C., LUMETTA, S., VON EICKEN, T., AND YELICK, K. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1993), Supercomputing '93, ACM, pp. 262–273.

[60] KUMAR, V., ZHENG, Y., CAVÉ, V., BUDIMLIĆ, Z., AND SARKAR, V. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014), ACM, p. 5.

[61] LAVRIJSEN, W., AND IANCU, C. Application Level Reordering of Remote Direct Memory Access Operations. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 988–997.

[62] LEISERSON, C. E. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers 100*, 10 (1985), 892–901.

[63] LU, H., SEO, S., AND BALAJI, P. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems* (Aug 2015), pp. 444–454.

[64] MALLINSON, A. C., BECKINGSALE, D. A., GAUDIN, W. P., HERDMAN, J. A., LEVESQUE, J. M., AND JARVIS, S. A. Cloverleaf : preparing hydrodynamics codes for exascale. In *A New Vintage of Computing : CUG2013* (2013), Cray User Group, Inc.

[65] MANJIKIAN, N., AND ABDELRAHMAN, T. S. (R) Scheduling of Wavefront Parallelism on Scalable Shared-memory Multiprocessors. In *icpp* (1996), IEEE, p. 0122.

[66] MARJANOVIĆ, V., LABARTA, J., AYGUADÉ, E., AND VALERO, M. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 5–16.

[67] MARTIN, S. M., BERGER, M. J., AND BADEN, S. B. Toucan - A Translator for Communication Tolerant MPI Applications. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 998–1007.

[68] MOIN, P. *Fundamentals of engineering numerical analysis*. Cambridge University Press, 2010.

[69] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 1965).

[70] NGUYEN, T., CICOTTI, P., BYLASKA, E., QUINLAN, D., AND BADEN, S. Automatic translation of MPI source into a latency-tolerant, data-driven form. *Journal of Parallel and Distributed Computing 106* (2017), 1 – 13.

[71] NGUYEN, T., CICOTTI, P., BYLASKA, E., QUINLAN, D., AND BADEN, S. B. Bamboo – Translating MPI applications to a latency-tolerant, data-driven form. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (Nov 2012), pp. 1–11.

[72] NYBERG, P. The Critical Role of Supercomputers in Weather Forecasting. *www.cray.com/blog/the-critical-role-of-supercomputers-in-weather-forecasting/* (2013).

[73] OPENMP, A. OpenMP 4.0 specification, 2013.

[74] PATRIZIO, A. U.S. Army plans for a 100 petaflop supercomputer. *itworld.com/article/2889072/u-s-army-plans-for-a-100-petaflop-supercomputer.html*.

[75] PEANO, G. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen 36*, 1 (Mar 1890), 157–160.

[76] PEREZ, J. M., BADIA, R. M., AND LABARTA, J. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing* (Sept 2008), pp. 142–151.

[77] PJEŠIVAC-GRBOVIĆ, J., ANGSKUN, T., BOSILCA, G., FAGG, G. E., GABRIEL, E., AND DONGARRA, J. J. Performance analysis of MPI collective operations. *Cluster Computing 10*, 2 (2007), 127–143.

[78] RAMKUMAR, B., SINHA, A., SALETORE, V., AND KALE, L. The CHARM parallel programming language and system: Part I - Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems* (1994).

[79] SCHMITT, V., AND CHARPIN, F. Pressure Distributions on the ONERA-M6 Wing at Transonic Mach Number. Tech. Rep. AGARD AR 138, 1979.

[80] SCHÖNHAGE, A., AND STRASSEN, V. Schnelle multiplikation grosser zahlen. *Computing 7*, 3-4 (1971), 281–292.

[81] SHIRES, D. R., POLLOCK, L. L., AND SPRENKLE, S. Program Flow Graph Construction For Static Analysis of MPI Programs. In *PDPTA* (1999).

[82] SNIR, M., GROPP, W. D., AND KOGGE, P. Exascale research: preparing for the post-Moore era.

[83] STEELE, JR., G. L. Making Asynchronous Parallelism Safe for the World. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1990), POPL '90, ACM, pp. 218–231.

[84] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal 30*, 3 (2005), 202–210.

[85] TANG, H., AND YANG, T. Optimizing Threaded MPI Execution on SMP Clusters. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM, pp. 381–392.

[86] TERPSTRA, D., JAGODE, H., YOU, H., AND DONGARRA, J. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009* (Berlin, Heidelberg, 2010), M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Springer Berlin Heidelberg, pp. 157–173.

[87] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[88] THORNTON, J. E. Parallel Operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems* (New York, NY, USA, 1965), AFIPS '64 (Fall, part II), ACM, pp. 33–40.

[89] TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM J. Res. Dev. 11*, 1 (Jan. 1967), 25–33.

[90] TRAFF, J. L. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference* (2002), IEEE, pp. 28–28.

[91] V. EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: A Mechanism for Integrated Communication and Computation. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 256–266.

[92] VALIANT, L. G. A Bridging Model for Parallel Computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[93] WANG, E., ZHANG, Q., ET AL. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188.

[94] [WEB]. Boost C++ Coroutine Library.
https://www.boost.org/doc/libs/release/libs/coroutine/.

[95] [WEB]. Cart3D Documentation, NASA Advanced Supercomputing Division.
https://www.nas.nasa.gov/publications/software/docs/cart3d/.

[96] [WEB]. Cloverleaf3D - A 3D Lagrangian-Eulerian hydrodynamics benchmark.
http://uk-mac.github.io/CloverLeaf3D/.

[97] [WEB]. Cray MPI Library. *https://pubs.cray.com/*.

[98] [WEB]. Edison Design Group, Inc. - C++ Front End.
https://www.edg.com/docs/edg_cpp.pdf.

[99] [WEB]. Intel MPI Library. *https://software.intel.com/en-us/intel-mpi-library*.

[100] [WEB]. Intel Xeon Phi processor family.
https://www.intel.com/content/www/us/en/products/processors/xeon-phi.html.

[101] [WEB]. Leetcode.com - The Painters Partition Problem.
https://articles.leetcode.com/the-painters-partition-problem/.

[102] [WEB]. MPI 4.0 Standard. *https://www.mpi-forum.org/mpi-40/*.

[103] [WEB]. MPI Standardization Forum. *https://www.mpi-forum.org/*.

[104] [WEB]. MPICH Library. *http://www.mpich.org/*.

[105] [WEB]. MVAPICH Library. *http://mvapich.cse.ohio-state.edu/*.

[106] [WEB]. NERSC Cori Interconnect.
http://www.nersc.gov/users/computational-systems/cori/configuration/interconnect/.

[107] [WEB]. NERSC Cori Supercomputer Configuration.
http://www.nersc.gov/users/computational-systems/cori/configuration/.

[108] [WEB]. NVIDIA Volta Tensor Core GPU Architecture.
https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[109] [WEB]. Oak Ridge National Laboratory, Summit Supercomputer.
https://www.olcf.ornl.gov/summit/.

[110] [WEB]. OpenMPI Library. *https://www.open-mpi.org/*.

[111] [WEB]. Performance Evaluation of the Boost C++ Coroutine Library.
https://boost.org/doc/libs/1_68_0/libs/coroutine/doc/html/coroutine/performance.html.

[112] [WEB]. Portable Operating System Interface (POSIX) Standard.
http://standards.ieee.org/develop/wg/POSIX.html.

[113] [WEB]. Sandia MPI Micro-Benchmark Suite (SMB). Version 1.0-1. *http://www.cs.sandia.gov/smb/*.

[114] [WEB]. Top500.org Supercomputer List (Jun/18). *https://www.top500.org/lists/2018/06/*.

[115] [WEB]. Topcoder.com - The Fair Workload Problem. *https://community.topcoder.com/stat?c=problem_statement&pm=1901&rd=4650*.

[116] WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[117] WOZNIAK, J. M., ARMSTRONG, T. G., WILDE, M., KATZ, D. S., LUSK, E., AND FOSTER, I. T. Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (May 2013), pp. 95–102.

[118] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.

[119] ZHANG, Q., JOHANSEN, H., AND COLELLA, P. A Fourth-Order Accurate Finite-Volume Method with Structured Adaptive Mesh Refinement for Solving the Advection-Diffusion Equation. *SIAM Journal on Scientific Computing 34*, 2 (2012), B179–B201.

[120] ZHENG, G., NEGARA, S., MENDES, C. L., KALE, L. V., AND RODRIGUES, E. R. Automatic Handling of Global Variables for Multi-threaded MPI Programs. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems* (Dec 2011), pp. 220–227.

[121] ZHENG, Y., KAMIL, A., DRISCOLL, M. B., SHAN, H., AND YELICK, K. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (May 2014), pp. 1105–1114.

[122] ZINGALE, M., ALMGREN, A., ET AL. Meeting the Challenges of Modeling Astrophysical Thermonuclear Explosions: Castro, Maestro, and the AMReX Astrophysics Suite. In *Journal of Physics: Conference Series* (2018), vol. 1031, IOP Publishing, p. 012024.