WARWICK
THE UNIVERSITY OF WARWICK

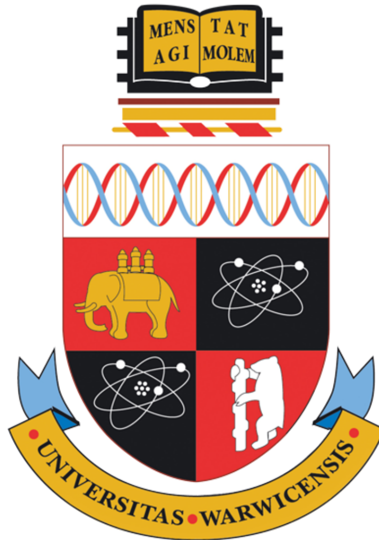
A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:
http://wrap.warwick.ac.uk/185102

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications



Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2

by

Peduru Hewage Suneth Dasantha Ekanayake

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

September 2023

Contents

List of Tables vi

List of Figures ix

List of Listings xiii

List of Algorithms xiv

Acknowledgements xvi

Declarations xvii

Sponsorship and Grants xix

Abstract xx

Abbreviations xxi

Notations xxiv

Chapter 1 Introduction **1**

1.1 Motivation and Problem Statement . . . 3
1.2 Thesis Contributions . . . 5
1.3 Thesis Overview . . . 7

Chapter 2 Background **9**

2.1 High-Performance Computing (HPC) Architectures . . . 9
2.1.1 Flynn's Taxonomy . . . 10
2.1.2 Multi-Core and Many-Core Architectures . . . 12
2.1.3 GPU Architecture . . . 12
2.2 Performance Metrics and Optimization Laws . . . 13
2.2.1 Key Performance Metrics . . . 14
2.2.2 Amdahl's Law and Gustafson's Law . . . 15
2.2.3 Roofline Model . . . 16
2.3 Parallel Programming Models . . . 18

2.3.1 Shared-Memory Systems 18
2.3.2 Distributed-Memory Systems 19
2.4 Performance Optimization Techniques 21
2.4.1 Loop Transformation Techniques 21
2.4.2 Communication Optimization 23
2.4.3 Polyhedral Model 25
2.5 Domain-Specific Languages (DSLs) and Frameworks in HPC 26
2.5.1 Mesh Types 27
2.5.2 Firedrake 28
2.5.3 Devito 28
2.5.4 TiDA 29
2.5.5 STELLA 29
2.5.6 Halide 29
2.5.7 Pochoir 30
2.5.8 Patus 31
2.5.9 PLuTo 31
2.5.10 Polly 32
2.5.11 Summary 33
2.6 OP2 DSL 35
2.6.1 Sets and Maps 35
2.6.2 Data 36
2.6.3 Parallel Loops 36
2.6.4 Data Layout 38
2.6.5 Kernels 39
2.6.6 Architecture 39
2.6.7 Supported Back-Ends 40

Chapter 3 Communication-Avoiding Optimizations on Shared-Memory Systems **44**
3.1 Motivation 44
3.2 Concepts and Approach 45
3.2.1 Sparse Tiling 46
3.2.2 Full Sparse Tiling and Generalized Full Sparse Tiling 46
3.2.3 Inspector/Executor Schemes 46
3.2.4 Loop-chain Abstraction 47
3.2.5 OP2 Shared-Memory Parallelism 50
3.3 SLOPE Library 51
3.3.1 Inspection Phase 52
3.3.2 Execution Phase 56
3.3.3 OP2-SLOPE Integration 56
3.4 Performance 57

3.4.1 Airfoil . . . 58
3.4.2 MG-CFD . . . 65
3.4.3 Volna . . . 72
3.4.4 OP2 Hydra . . . 79
3.5 Factors Impacting Performance . . . 84
3.5.1 Tile Size . . . 84
3.5.2 Mesh Partitioner . . . 85
3.5.3 Loop Fusion Scheme . . . 86
3.6 Conclusion . . . 87

Chapter 4 Communication-Avoiding Optimizations on Distributed-Memory Systems **88**
4.1 Need for Communication-Avoidance . . . 89
4.2 Communication-Avoidance Back-End . . . 89
4.2.1 OP2 Distributed-Memory Parallelism . . . 90
4.2.2 Halo Exchanges . . . 91
4.2.3 OP2 Loop Execution . . . 92
4.2.4 Multi-Layered Halo Data Structure . . . 93
4.2.5 Loop-chains with CA . . . 93
4.2.6 Inspection Phase . . . 95
4.2.7 Execution Phase . . . 101
4.2.8 Automatic Code Generation . . . 103
4.2.9 OP2-CA Integration . . . 104
4.3 Analytic Model for Loop-chain Performance . . . 105
4.3.1 OP2 Loop Execution . . . 105
4.3.2 CA Loop-chain Execution . . . 105
4.3.3 Insights from Performance Model . . . 106
4.3.4 Challenges in Performance Modeling . . . 107
4.4 Performance . . . 108
4.4.1 MG-CFD . . . 109
4.4.2 OP2 Hydra . . . 117
4.5 Conclusion . . . 120

Chapter 5 Integrating Shared- and Distributed-Memory Communication-Avoiding Optimizations for CPUs **123**
5.1 Shared- and Distributed-Memory Parallelism . . . 123
5.2 OP2 based SDMP for CPU Clusters . . . 124
5.3 Integrating CA Back-End and SLOPE Library . . . 126
5.3.1 CA+SLOPE Inspection for Distributed-Memory Parallelism . . . 126
5.3.2 SLOPE Inspection with Distributed-Memory Parallelism . . . 127
5.3.3 SLOPE Execution with Distributed-Memory Parallelism . . . 128

5.3.4 OP2-CA-SLOPE Integration 129
5.4 Performance Model Extension for CA with On-Node Sparse Tiling 130
5.5 Performance 130
5.5.1 MG-CFD 131
5.5.2 OP2 Hydra 135
5.6 Conclusion 136

Chapter 6 Integrating Shared- and Distributed-Memory Communication-Avoiding Optimizations for GPUs 137
6.1 OP2 based SDMP for GPU Clusters 137
6.2 CA Back-End for GPUs 138
6.3 Performance Model Extension for CA with GPUs 139
6.4 Performance 139
6.4.1 MG-CFD 139
6.4.2 OP2 Hydra 145
6.5 Conclusion 148

Chapter 7 Conclusions and Future Work 149
7.1 Contributions and Conclusions 150
7.2 Thesis Limitations and Future Work 152
7.2.1 Analytical Model Enhancements and Further Validations 152
7.2.2 CA Back-End Memory Enhancements 153
7.2.3 Automate Code Generation with Lazy Evaluation 153
7.2.4 SLOPE Evaluation for Distributed-Memory Parallelism 154
7.2.5 Evaluation Scope 154

Bibliography 155

Appendix A Analytical Performance Model and Extensions 169
A.1 Model Parameters 169
A.2 Model Equations 170

Appendix B Runtimes of Benchmarked Applications 171
B.1 Chapter 3 Runtimes 171
B.1.1 Airfoil Runtimes 171
B.1.2 MG-CFD Runtimes 184
B.1.3 Volna Runtimes 190
B.1.4 OP2 Hydra Runtimes 196
B.2 Chapter 4 Runtimes 205
B.2.1 MG-CFD on ARCHER2 205
B.2.2 OP2 Hydra on ARCHER2 206
B.3 Chapter 5 Runtimes 207

B.3.1 MG-CFD on ARCHER2 . . . 207
B.3.2 OP2 Hydra on ARCHER2 . . . 208
B.4 Chapter 6 Runtimes . . . 209
B.4.1 MG-CFD on Cirrus . . . 209
B.4.2 OP2 Hydra on Cirrus . . . 210

List of Tables

2.1 Summary of DSLs/Frameworks . . . 33

3.1 Systems specifications . . . 58
3.2 Best SLOPE Airfoil performance on Skylake, Telos, and ARCHER2 . . . 64
3.3 SLOPE MG-CFD loop fusion schemes . . . 67
3.4 SLOPE MG-CFD single socket performance summary on Scyrus for loop fusion schemes . . . 68
3.5 Best SLOPE MG-CFD performance on Skylake, Telos, and ARCHER2 . . . 68
3.6 Dataset properties . . . 72
3.7 Best SLOPE Volna performance on Skylake, Telos, and ARCHER2 . . . 74
3.8 Loop-chain information . . . 80
3.9 Best SLOPE Hydra loop-chain performance on Skylake, Telos, and ARCHER2 84

4.1 System specifications . . . 109
4.2 MG-CFD on ARCHER2 - 8M Mesh - Model Components . . . 112
4.3 MG-CFD on ARCHER2 - 24M Mesh - Model Components . . . 113
4.4 OP2 Hydra loop-chains with single halo layer ($HE_l = 1$). . . . 118
4.5 OP2 Hydra loop-chains with multiple halo layers ($HE_l >= 1$) . . . 119
4.6 Hydra loop-chains (LCs) on ARCHER2 - 8M Mesh - Model Components . 121
4.7 Hydra loop-chains (LCs) on ARCHER2 - 24M Mesh - Model Components 122

6.1 System specification . . . 140
6.2 MG-CFD on Cirrus - 8M Mesh - Model Components . . . 141
6.3 MG-CFD on Cirrus - 24M Mesh - Model Components . . . 142
6.4 Hydra loop-chains (LCs) on Cirrus - 8M Mesh - Model Components . . . 147
6.5 Hydra loop-chains (LCs) on Cirrus - 24M Mesh - Model Components . . . 147

A.1 Model Parameters . . . 169
A.2 Model Equations for CA in CPUs . . . 170
A.3 Model Equations for CA+SLOPE in CPUs . . . 170
A.4 Model Equations for CA in GPUs . . . 170

B.1 OP2 Airfoil single socket runtimes on Scyrus (Figure 3.5 and Figure B.1 runtimes in seconds) 172
B.2 SLOPE Airfoil single socket runtimes on Scyrus (Figure 3.5 and Figure B.1 runtimes in seconds) 172
B.3 OP2 Airfoil dual socket runtimes on Scyrus (Figure 3.6 and Figure B.2 runtimes in seconds) 174
B.4 SLOPE Airfoil dual socket runtimes on Scyrus (Figure 3.6 and Figure B.2 runtimes in seconds) 174
B.5 OP2 Airfoil single socket runtimes on Telos (Figure 3.7 and Figure B.3 runtimes in seconds) 176
B.6 SLOPE Airfoil single socket runtimes on Telos (Figure 3.7 and Figure B.3 runtimes in seconds) 176
B.7 OP2 Airfoil dual socket runtimes on Telos (Figure 3.8 and Figure B.4 runtimes in seconds) 178
B.8 SLOPE Airfoil dual socket runtimes on Telos (Figure 3.8 and Figure B.4 runtimes in seconds) 178
B.9 OP2 Airfoil single socket runtimes on ARCHER2 (Figure 3.9 and Figure B.5 runtimes in seconds) 180
B.10 SLOPE Airfoil single socket runtimes on ARCHER2 (Figure 3.9 and Figure B.5 runtimes in seconds) 180
B.11 OP2 Airfoil dual socket runtimes on ARCHER2 (Figure 3.10 and Figure B.6 runtimes in seconds) 182
B.12 SLOPE Airfoil dual socket runtimes on ARCHER2 (Figure 3.10 and Figure B.6 runtimes in seconds) 182
B.13 OP2 MG-CFD runtimes on Scyrus (Figure 3.14 and Figure 3.15 runtimes in seconds) 184
B.14 SLOPE MG-CFD runtimes on Scyrus (Figure 3.14 and Figure 3.15 runtimes in seconds) 184
B.15 OP2 MG-CFD runtimes on Telos (Figure 3.16 and Figure 3.17 runtimes in seconds) 186
B.16 SLOPE MG-CFD runtimes on Telos (Figure 3.16 and Figure 3.17 runtimes in seconds) 186
B.17 OP2 MG-CFD runtimes on ARCHER2 (Figure 3.18 runtimes in seconds) 188
B.18 SLOPE MG-CFD runtimes on ARCHER2 (Figure 3.18 runtimes in seconds) 188
B.19 OP2 Volna runtimes on Scyrus (Figure 3.21 and Figure 3.22 runtimes in seconds) 190
B.20 SLOPE Volna runtimes on Scyrus (Figure 3.21 and Figure 3.22 runtimes in seconds) 190
B.21 OP2 Volna runtimes on Telos (Figure 3.23 and Figure 3.24 runtimes in seconds) 192

B.22 SLOPE Volna runtimes on Telos (Figure 3.23 and Figure 3.24 runtimes in seconds) . . . 192
B.23 OP2 Volna runtimes on ARCHER2 (Figure 3.25 and Figure 3.26 runtimes in seconds) . . . 194
B.24 SLOPE Volna runtimes on ARCHER2 (Figure 3.25 and Figure 3.26 runtimes in seconds) . . . 194
B.25 OP2 Hydra single socket runtimes on Scyrus (Figure 3.29 runtimes in seconds) 196
B.26 SLOPE Hydra single socket runtimes on Scyrus (Figure 3.29 runtimes in seconds) . . . 196
B.27 OP2 Hydra dual socket runtimes on Scyrus (Figure 3.30 runtimes in seconds) 197
B.28 SLOPE Hydra dual socket runtimes on Scyrus (Figure 3.30 runtimes in seconds) . . . 197
B.29 OP2 Hydra single socket runtimes on Telos (Figure 3.31 runtimes in seconds) 199
B.30 SLOPE Hydra single socket runtimes on Telos (Figure 3.31 runtimes in seconds) . . . 199
B.31 OP2 Hydra dual socket runtimes on Telos (Figure 3.32 runtimes in seconds) 200
B.32 SLOPE Hydra dual socket runtimes on Telos (Figure 3.32 runtimes in seconds) . . . 200
B.33 OP2 Hydra single socket runtimes on ARCHER2 (Figure 3.33 runtimes in seconds) . . . 202
B.34 SLOPE Hydra single socket runtimes on ARCHER2 (Figure 3.33 runtimes in seconds) . . . 202
B.35 OP2 Hydra dual socket runtimes on ARCHER2 (Figure 3.34 runtimes in seconds) . . . 203
B.36 SLOPE Hydra dual socket runtimes on ARCHER2 (Figure 3.34 runtimes in seconds) . . . 203
B.37 MG-CFD on ARCHER2 with CA (Figure 4.8 and Figure 4.9 Runtimes in seconds) . . . 205
B.38 Hydra loop-chains (LCs) on ARCHER2 with CA (Figure 4.15 and Figure 4.16 Runtimes in seconds) . . . 206
B.39 SLOPE MG-CFD on ARCHER2 with CA+SLOPE (Figure 5.6 and Figure 5.7 Runtimes in seconds) . . . 207
B.40 Hydra loop-chains (LCs) on ARCHER2 with CA+SLOPE (Figure 5.9 and Figure 5.10 Runtimes in seconds) . . . 208
B.41 MG-CFD on Cirrus with CA (Figure 6.1 and Figure 6.2 Runtimes in seconds) 209
B.42 Hydra loop-chains (LCs) on Cirrus with CA (Figure 6.7 and Figure 6.8 Runtimes in seconds) . . . 210

List of Figures

1.1 50 years of microprocessor trend data [1] 2

2.1 Flynn's Taxonomy 10
2.2 NVIDIA Tesla V100 architecture [2] 13
2.3 Amdahl's Law 15
2.4 Roofline Model 17
2.5 Loop nest interchange transformation 25
2.6 Differnet types of meshes 27
2.7 Pochoir two-phase compilation strategy [3] 30
2.8 PLuTo source-to-source transformation system [4] 31
2.9 Polly architecture [5] 32
2.10 OP2 data layout for direct data access 38
2.11 OP2 data layout for indirect data access 39
2.12 OP2 code generation 40

3.1 Example unstructured-mesh with nodes, edges, and quadrilateral cells (data values in parenthesis) 48
3.2 Mesh partitioning and coloring in inspection phase for the OP2 loop-chain in Listing 3.3 53
3.3 Conflicting colors 55
3.4 OP2 code generation with SLOPE 56
3.5 Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: single socket, 12 threads) 61
3.6 Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: dual socket, 24 threads) 61
3.7 Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: single socket, 24 threads) 62
3.8 Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: dual socket, 48 threads) 62
3.9 Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: single socket, 64 threads) 63

3.10 Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads) 63
3.11 180k airfoil mesh partitioning and coloring 64
3.12 Roofline graph of SLOPE Airfoil and OP2 Airfoil 64
3.13 1M MG-CFD mesh partitioning and coloring 68
3.14 Runtime variation of SLOPE MG-CFD with tile sizes on Scyrus (Configurations: single socket, 12 threads) 69
3.15 Runtime variation of SLOPE MG-CFD with tile sizes on Scyrus (Configurations: dual socket, 24 threads) 69
3.16 Runtime variation of SLOPE MG-CFD with tile sizes on Telos (Configurations: single socket, 24 threads) 70
3.17 Runtime variation of SLOPE MG-CFD with tile sizes on Telos (Configurations: dual socket, 48 threads) 70
3.18 Runtime variation of SLOPE MG-CFD with tile sizes on ARCHER2 (Configurations: single socket, 64 threads) 71
3.19 Roofline graphs of SLOPE MG-CFD and OP2 MG-CFD 71
3.20 Volna Catalina mesh partitioning and coloring 74
3.21 Runtime variation of SLOPE Volna with tile sizes on Scyrus (Configurations: single socket, 12 threads) 75
3.22 Runtime variation of SLOPE Volna with tile sizes on Scyrus (Configurations: dual socket, 24 threads) 75
3.23 Runtime variation of SLOPE Volna with tile sizes on Telos (Configurations: single socket, 24 threads) 76
3.24 Runtime variation of SLOPE Volna with tile sizes on Telos (Configurations: dual socket, 48 threads) 76
3.25 Runtime variation of SLOPE Volna with tile sizes on ARCHER2 (Configurations: single socket, 64 threads) 77
3.26 Runtime variation of SLOPE Volna with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads) 77
3.27 Roofline graphs of SLOPE Volna and OP2 Volna 78
3.28 RR Trent XWB engine (©Rolls Royce plc. Reproduced with permission.) 79
3.29 Runtime variation of SLOPE Hydra loop-chains with tile sizes on Scyrus (Configurations: single socket, 12 threads) 81
3.30 Runtime variation of SLOPE Hydra loop-chains with tile sizes on Scyrus (Configurations: dual socket, 24 threads) 81
3.31 Runtime variation of SLOPE Hydra loop-chains with tile sizes on Telos (Configurations: single socket, 24 threads) 82
3.32 Runtime variation of SLOPE Hydra loop-chains with tile sizes on Telos (Configurations: dual socket, 48 threads) 82
3.33 Runtime variation of SLOPE Hydra loop-chains with tile sizes on ARCHER2 (Configurations: single socket, 64 threads) 83

3.34 Runtime variation of SLOPE Hydra loop-chains with tile sizes on ARCHER2 (Configurations: single socket, 128 threads) 83

4.1 OP2 partitioning over two MPI ranks and resulting halos on each rank [6] 90
4.2 op_dat and op_map data structures with (a) single and (b) multiple halo levels 92
4.3 Halo layer with depth 1 94
4.4 Halo layer with depth 2 94
4.5 Grouped halo array 102
4.6 OP2 code generation/translation process 103
4.7 OP2 code generation with CA 104
4.8 MG-CFD CA performance with 8M mesh on ARCHER2 111
4.9 MG-CFD CA performance with 24M mesh on ARCHER2 111
4.10 MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 2) 114
4.11 MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 2) 114
4.12 MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 32) 115
4.13 MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 32) 115
4.14 MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 64, #Loops - 32) 116
4.15 Hydra CA performance with 8M mesh on ARCHER2 120
4.16 Hydra CA performance with 24M mesh on ARCHER2 120

5.1 Mesh partitioning for processes 126
5.2 Coloring Rank X 127
5.3 Coloring Rank Y 128
5.4 OP2 code generation with CA+SLOPE 129
5.5 MG-CFD single node runtime variation for different Process × Thread combinations on ARCHER2 for different loop counts (⟨version⟩ - ⟨loop count⟩) 131
5.6 MG-CFD CA+SLOPE performance with 8M mesh on ARCHER2 133
5.7 MG-CFD CA+SLOPE performance with 24M mesh on ARCHER2 133
5.8 MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 64, #Loops - 16) 134
5.9 Hydra CA+SLOPE performance with 8M mesh on ARCHER2 135
5.10 Hydra CA+SLOPE performance with 24M mesh on ARCHER2 135

6.1 MG-CFD CA performance with 8M mesh on Cirrus 140
6.2 MG-CFD CA performance with 24M mesh on Cirrus 140

6.3 MG-CFD MPI data exchange in bytes for the synthetic loop-chain (Configurations: #Nodes - 8, #Loops - 2) 143
6.4 MG-CFD HtoD and DtoH data exchange in bytes for the synthetic loop-chain in a single process. (Configurations: #Nodes - 8, #Loops - 2) . . . 143
6.5 MG-CFD MPI data exchange in bytes for the synthetic loop-chain (Configurations: #Nodes - 8, #Loops - 16) 144
6.6 MG-CFD HtoD and DtoH data exchange in bytes for the synthetic loop-chain in a single process. (Configurations: #Nodes - 8, #Loops - 16) . . . 144
6.7 Hydra CA performance with 8M mesh on Cirrus 146
6.8 Hydra CA performance with 24M mesh on Cirrus 146

B.1 Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: single socket, 12 threads) 171
B.2 Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: dual socket, 24 threads) 173
B.3 Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: single socket, 24 threads) 176
B.4 Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: dual socket, 48 threads) 178
B.5 Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: single socket, 64 threads) 180
B.6 Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads) 182

List of Listings

2.1 Loop tiling example . . . 23
2.2 OP2 sets and maps . . . 35
2.3 OP2 data . . . 36
2.4 Parallel loop written in OP2 API . . . 37
2.5 Direct parallel loop written in OP2 API . . . 37
2.6 Indirect parallel loop written in OP2 API . . . 38
2.7 Sequential back-end kernel . . . 40
2.8 OpenMP back-end kernel . . . 41
2.9 MPI back-end kernel . . . 42
2.10 CUDA back-end kernel . . . 42
3.1 Sequential loops in C . . . 48
3.2 Loops written in OP2 API . . . 49
3.3 Section of an OP2 program to explain full sparse tiling with the SLOPE library. Example inspired by Strout et al. [7]. . . . 51
3.4 SLOPE inspector API. Generated for the OP2 loop-chain in Listing 3.3. 52
3.5 Airfoil loop-chain with SLOPE . . . 59
3.6 MG-CFD RK loop-chain with SLOPE . . . 66
3.7 Volna loop-chain with SLOPE . . . 73
4.1 MPI halo send and recv . . . 92
4.2 Loop-chain ($\mathbb{L}$) example written using OP2 API for CA . . . 96
4.3 `op_halo_info_core` data structure . . . 98
4.4 `halo_list_core` data structure and its use in the CA back-end . . . 99
4.5 Expandable synthetic *2-loop-chain* . . . 110
5.1 Expandable synthetic *2-loop-chain* written using SLOPE API . . . 132

List of Algorithms

3.1 Loop-chain execution with shared-memory, SLOPE [8] 56

4.1 Algorithm to determine a halo exchange and dirty bit management [9] . . . 91
4.2 OP2 loop execution 93
4.3 Loop-chain execution with CA 95
4.4 `calc_halo_layers` 97

5.1 Loop-chain inspection/execution with CA+SLOPE 125

6.1 MPI+CUDA halo exchange [9] 138

to my parents, wife, and sister
with love and gratitude

Acknowledgements

It is my utmost pleasure to extend my heartfelt acknowledgements to the people who supported, influenced, and guided me throughout my PhD journey.

First and foremost, I would like to express my sincere thanks to my supervisor, Prof. Gihan Mudalige for his invaluable guidance and support. I am grateful for his optimistic and flexible approach in facilitating my research, as well as the invaluable support provided for publishing and presenting my research at international conferences. I am grateful to Prof. Stephen Jarvis for acting as my second supervisor and providing his insightful feedback on my progress. My sincere appreciation goes to Dr. Fabio Luporini and Dr. Istvan Reguly for their valuable directions in navigating my research to success.

I am thankful for the funding provided by Rolls Royce plc., the ASiMoV project, UK EPSRC, and the Department of Computer Science at the University of Warwick for my research. My research would not have been possible without the use of HPC resources, including ARCHER2 and Cirrus computing clusters supported by the EPCC, for which I am thankful. I am deeply appreciative of the HPC research community at the University of Warwick, including Dr. Kamalavasan Kamalakkannan, Dr. Arun Prabhakar, Dr. Dean Chester, Archie Powell, and Zaman Lantra for their collaboration and friendship.

I extend my thanks to Prof. Rohan Munasinghe, Prof. Dileeka Dias, and Dr. Dilum Bandara from the University of Moratuwa for their recommendations for my academic journey. I am grateful to my teachers at Kalutara Vidyalaya National School and Bombuwala Primary School for laying a strong foundation for my education. My sincere thanks go to the colleagues at LSEG Technology who helped in enriching my engineering skills. I am grateful to my friends, including Pasan, Sameera, Hasith, Vidura, Tharaka, and many others for creating unforgettable memories with countless trips in Sri Lanka.

Finally, I am profoundly thankful to my family members. I am indebted to my father, Sepala Ekanayake, and my mother, Shyamalee Ekanayake, for being the strength of my life and encouraging and facilitating my studies in every way possible. I also extend my heartfelt gratitude to my grandparents, who have been a source of wisdom and inspiration throughout my life. I am grateful to my sister, Thiekshani, and brother-in-law, Dinushka, for their invaluable support during my challenging times. I am incredibly grateful to my wife's parents and brother, Pasindu for their love and continuous support. I want to express my sincere gratitude to my lovely wife, Pavithri, for always being there for me and inspiring me to do my best. Her unwavering support gave me the determination to complete this journey. Without her encouragement, this journey would have been much tougher. I dedicate this thesis to them with immense love and gratitude.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy in Computer Science. I, Peduru Hewage Suneth Dasantha Ekanayake, declare that this thesis titled, 'Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2' has been composed by myself and has not been submitted in any previous application for any degree. I confirm that:

- This work was done wholly or mainly while in candidature for the research degree at this University.
- The work presented (including data generated and data analysis) was carried out by the author.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

Portions of this work have appeared in the following publication:

- Parts of Chapter 4, 5, and 6 in [10]:
  Suneth Dasantha Ekanayake, István Zoltán Reguly, Fabio Luporini, and Gihan Ravideva Mudalige. 2023. Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2. In 52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA. ACM, NewYork, NY, USA, 12 pages. https://doi.org/10.1145/3605573.3605604

The OP2 communication-avoidance framework, SLOPE library, and the applications used in this work are available as open-source software:

Frameworks/Libraries

- OP2 CA Framework [11]: Communication-Avoidance framework developed for the OP2 DSL
  URL: https://github.com/OP-DSL/OP2-Common/tree/feature/mpi_comm_avoid
- SLOPE CA Framework [12]: SLOPE library modifications to support OP2 CA framework
  URL: https://github.com/coneoproject/SLOPE/tree/feature/comm_avoid

Applications

- Airfoil [13]: Airfoil benchmarking application changes to demonstrate SLOPE library enhacements
  URL: https://github.com/OP-DSL/OP2-Common/tree/perf/airfoil_noRMS

- MG-CFD [14]: MG-CFD Application changes to demonstrate SLOPE library enhacements
  URL: https://github.com/warwick-hpsc/MG-CFD-app-OP2/tree/feature/slope

- Volna [15]: Volna Application changes to demonstrate SLOPE library enhacements
  URL: https://github.com/reguly/volna/tree/feature/slope

- MG-CFD (CA) [16]: MG-CFD Application changes to demonstrate OP2 CA framework enhacements
  URL: https://github.com/warwick-hpsc/MG-CFD-app-OP2/tree/feature/mpi_comm_avoid

Sponsorship and Grants

This research was supported by Rolls Royce plc., and by the UK EPSRC (EP/S005072/1 – Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems – ASiMoV).

This work used the ARCHER2 UK National Supercomputing Service and the Cirrus UK National Tier-2 HPC Service at EPCC funded by the University of Edinburgh and EPSRC (EP/P020267/1).

Abstract

This thesis presents data movement-reducing and communication-avoiding optimizations and their practicable implementation for large-scale unstructured-mesh numerical simulation applications. Utilizing the high-level abstractions of the OP2 domain-specific library, we reason about techniques for reduced communications across a consecutive sequence of loops – a *loop-chain*. The optimizations are explored for shared-memory systems where multiple processors share a common memory space and distributed-memory systems that comprise separate memory spaces across multiple nodes. We elucidate the challenges when executing unstructured-mesh applications on large-scale high-performance systems that are specifically related to data sharing and movement, synchronization, and communication among processes. A key feature of the work is to mitigate these problems for real-world, large-scale applications and computing kernels, bringing together proven and effective techniques within a DSL framework.

On shared-memory systems, We explore cache-blocking tiling, a key technique for exploiting data locality, in unstructured-mesh applications by integrating the SLOPE library, a cache-blocking tiling library, with OP2. For distributed-memory systems, we analyze the trade-off between increased redundant computation in place of data movement and design a new communication-avoiding back-end for OP2 that applies these techniques automatically to any OP2 application targeting CPUs and GPUs.

The communication-avoiding optimizations are applied to two non-trivial applications, including the OP2 version of Rolls Royce's production CFD application, Hydra, on problem sizes representative of real-world workloads. Results demonstrate how, for select configurations, the new communication-avoiding back-end provides between 30 – 65% runtime reductions for the loop-chains in these applications on both an HPE Cray EX system and an NVIDIA V100 GPU cluster. We model and examine the determinants and characteristics of a given unstructured-mesh loop-chain that lead to performance benefits with communication-avoidance techniques, providing insights into the general feasibility and profitability of using the optimizations for this class of applications.

Abbreviations

AI	Arithmetic Intensity
API	Application Programming Interface
AVX	Advanced Vector Extensions
BW	Bandwidth
CA	Communication-Avoidance
CFD	Computational Fluid Dynamics
CLANG	C Compiler
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSL	Domain-Specific Language
EPCC	Edinburgh Parallel Computing Centre
EPYC	AMD EPYC Processor
FE	Finite Element
FEA	Finite Element Analysis
FEM	Finite Element Method
FIFO	First-In, First-Out
FLOPS	Floating-Point Operations Per Second
FVM	Finite Volume Method
GB	Gigabyte
GNU	GNU's Not Unix
GPC	General-Purpose Computing
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HPCG	High-Performance Conjugate Gradient
HPE	Hewlett Packard Enterprise
ID	Identifier

INC	Increment
IR	Intermediate Representation
LCA	Lowest Common Ancestor
LES	Large Eddy Simulation
LFU	Least Frequently Used
LIFO	Last-In, First-Out
LINPACK	Linear Algebra PACKage
LLVM	Low-Level Virtual Machine
LRU	Least Recently Used
MAX	Maximum
METIS	Mesh Partitioning and Graph Partitioning
MG	Multi-grid
MG-CFD	Multi-grid Computational Fluid Dynamics
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MPI	Message Passing Interface
MPICH	MPI Implementation
MPT	Massively Parallel Technologies
MRU	Most Recently Used
NACA	National Advisory Committee for Aeronautics
NASA	National Aeronautics and Space Administration
NULL	Null Pointer
NUMA	Non-Uniform Memory Access
OMP	OpenMP
OS	Operating System
PAPI	Performance Application Programming Interface
PCI	Peripheral Component Interconnect
PGAS	Partitioned Global Address Space
RANS	Reynolds-Averaged Navier-Stokes
RDMA	Remote Direct Memory Access
READ	Random Early Detection
RK	Runge-Kutta
RR	Rolls Royce

RW	Read-Write
SDMP	Synchronous Dynamic Memory Protection
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SLES	SUSE Linux Enterprise Server
SMM	Simultaneous Multithreading
SMP	Symmetric Multiprocessing
SSE	Streaming SIMD Extensions
SYCL	Standard C++ for heterogeneous computing
TLB	Translation Lookaside Buffer
UFL	Unified Form Language
URANS	Unsteady Reynolds-Averaged Navier-Stokes
VTK	Visualization Toolkit

Notations

δ Size of a data element of an `op_dat`, d (in bytes)

γ_l Compute time for one iteration of the loop, l with OpenMP threads

Λ Latency in a GPU cluster

B Bandwidth

c Message packing and unpacking cost

d_l Number of `op_dat`s in loop, l

g_l Compute time for one iteration of the loop, l

h_l Halo extension for loop, l

L Latency

m^r Maximum grouped message size (in bytes) sent to each neighbor with r halo layers

m_l^1 Maximum message size (in bytes) sent to a neighbor by loop, l with a single halo extension

r Maximum number of halo layers

S_l^1 Execute halo number of iterations in loop, l

S_l^c Number of *core* iterations, in loop, l

S_d^{eeh,h_l} *eeh* size of `op_dat`, d in level, h_l

S_d^{enh,h_l} *enh* size of `op_dat`, d in level, h_l

$T_{ca,\mathbb{L}}$ Total CA runtime of the full loop-chain, $\mathbb{L}$

$T_{op,\mathbb{L}}$ Total OP2 runtime of the full loop-chain, $\mathbb{L}$

$T_{op2,l}$ Time taken by an OP2 loop, l

Chapter 1

Introduction

The end of frequency scaling in the middle of the last decade has led processor architectures to move towards massively parallel designs. According to the trend data shown in Figure 1.1, it is clear that single-thread performance and frequency scaling of processors have reached their limits. This suggests that future improvements in computing power will have to come from other sources, such as parallel processing and more efficient algorithms. Therefore, modern processors have featured a proliferation of arithmetic capability in the form of increasing discrete processor cores, both on traditional CPUs as well as accelerator devices such as GPUs. For example, a snapshot of the number of processor cores on a high-end CPU currently has close to 100 [17], while GPUs are designed with over 2000 cores [18], albeit cores that are comparatively much simpler. Large-scale clusters of these devices have reached hundred-thousands to millions of cores, as seen from the recently unveiled exascale supercomputers [19].

However, the speed of memory and network channels interconnecting the processors and system/device memories have largely lagged behind [20], leading to significant memory bandwidth bottlenecks. As a result, the performance of many conventional algorithms optimized for fast floating-point operations has stalled. This phenomenon is reflected in benchmarks like LINPACK (Linear Algebra PACKage) and HPCG (High-Performance Conjugate Gradient), which provide critical insights into the challenges faced by modern parallel applications [21, 22]. LINPACK [21], a benchmark focusing on solving dense systems of linear equations through LU factorization, traditionally emphasizes floating-point performance and has long been a key metric for ranking supercomputers on the TOP500 list [23]. Simultaneously, HPCG evaluates the performance of HPC systems for sparse linear algebra operations, specifically the Conjugate Gradient method, placing a heavier emphasis on communication and memory access patterns [22]. The main challenge identified by LINPACK and HPCG is the imbalance between the increasing computational capabilities of processors and the slower growth in the speed of memory and network channels. This imbalance leads to memory bandwidth limitations and bottlenecks in performance, emphasizing the critical need for innovative algorithms that address these challenges. Developing algorithms with reduced data movement, or

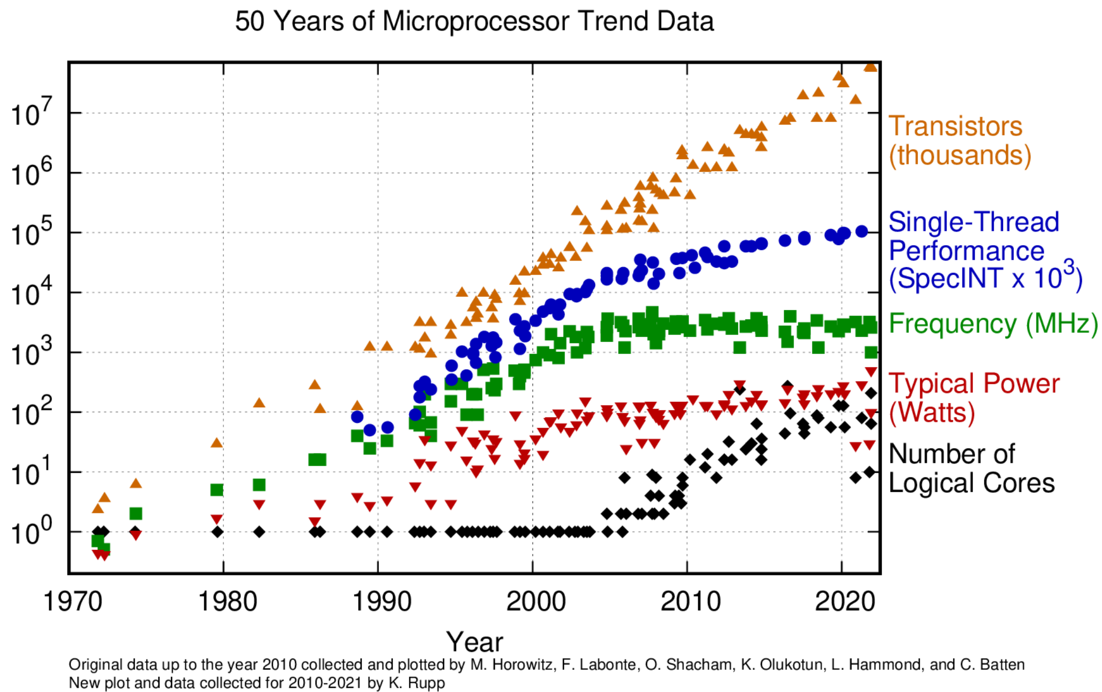


Figure 1.1: 50 years of microprocessor trend data [1]

communication-avoiding (CA) algorithms, have therefore become an intense area of research [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]. The underlying motivation of these works is to exploit the significantly high computing capability of these processors in place of communications, aiming to obtain higher performance gains.

The main challenge in adopting communication-avoiding optimizations or techniques in real-world applications is the significant effort and difficulty in implementing them and the subsequent code maintenance. The specific optimizations are highly complex and involved, usually obfuscating the source code with platform-specific low-level features. A large body of work has developed the underlying theory for CA techniques, including tiling [7, 35, 36, 37, 38, 39, 40, 41, 42, 43] and reduced distributed-memory systems communications [8, 37, 44, 45]. Compile time application of these optimizations, based on the polyhedral model [46, 47, 48] have been developed within compiler frameworks such as LLVM's Polly [5] and Pochoir [3]. Another strand of works such as by Demmel et al. [49, 50, 51, 52, 53, 54, 55, 56, 57, 58] have successfully developed libraries that applications can utilize CA-optimized numerical methods. More recently, domain-specific languages (DSLs) and similar high-level frameworks have demonstrated a pathway in applying these exotic optimizations to larger, non-trivial applications. Key DSLs with CA capabilities include Firedrake [59, 60] for unstructured-mesh based FE applications and OPS [36] and Devito [61, 62] for structured-mesh based applications.

This research aims to examine and apply key data movement-reducing techniques to large-scale, real-world applications. To this end, we build upon previous work by Luporini et al. [8] bringing together techniques for reducing data movement for the

unstructured-mesh applications class, codifying them through the OP2 DSL [6], an embedded domain-specific language for developing unstructured-mesh applications. We begin with a focus on optimizations with shared-memory parallelism, then move on to distributed-memory parallel execution, and finally, the combination of shared-memory and distributed-memory parallelizations on multi-core and many-core processors. A new CA back-end for OP2 is designed such that any application using OP2 can utilize the optimizations. The new back-end is then applied to our main application of interest, Hydra [63], a production computational fluid dynamics (CFD) application used for aero-engine design by Rolls Royce plc.

1.1 Motivation and Problem Statement

Communication-avoiding algorithms have a rich literature with reducing data movement identified as a fundamental optimization [64]. Out of these techniques, improving data locality by restructuring loops or rescheduling loop iterations, generally known as tiling or loop-blocking, have long been well understood [41, 42, 65, 66]. The theoretical underpinnings of these loop transformations have been comprehensively described through the polyhedral model [43, 67].

Many frameworks and libraries have been developed based on the polyhedral model. Key works include PLuTo [4], a fully automatic polyhedral program optimization system, Polly [5], an LLVM framework for high-level loop and data locality optimizations, Pochoir [3], a compiler and runtime system for implementing stencil computations on multicore processors and PolyMage [68] and Halide [69] which specifically targets image processing pipelines. All these works create polyhedra – multi-dimensional sub-iteration spaces within loops with static, regular access patterns/dependencies. Such loops can be readily viewed as iterations over a structured-mesh [70] with regular stencils defining the dependency neighborhood. The extension of these algorithms to distributed-memory systems requires them to account for the dependencies in the loop nests when the iteration space is spread over a number of processes or disparate memory areas. This leads to the need for those dependencies to be satisfied through communications of extra halo layers. These extensions have been developed for several of the aforementioned frameworks, for PLuTo [71, 72] and Distributed Halide [73] in addition to other works such as R-STREAM [74] and work by Classen and Griebl [75].

However, applying these optimizations, particularly the ideas of the polyhedral model for unstructured-mesh applications, are relatively limited in literature as a consequence of the added complexity of managing the irregular dependencies specified by the explicit connectivity of the unstructured-mesh. The dependence structure arises in these applications due to their characteristic indirect memory accesses, specifically indirect increments (e.g., `D[map[i]] += f(...)`) and indirect reads, through explicit connectivity *mappings* [8]. The indirect accesses lead to the need to do the dependence analysis via mappings, as opposed to the static dependence neighborhoods (e.g., specified by a stencil)

commonly exploited in general static loop optimizations. As such, the analysis needs to be carried out dynamically, at runtime, as demonstrated by Luporini et al. [8] based on the *loop-chain* abstraction introduced in [24].

Sparse tiling [8] has been previously integrated into Firedrake [59, 60] using the SLOPE library. Firedrake is a high-level DSL framework for the automated solution of finite element computations. It requires problems to be specified in the Unified Form Language (UFL) [76]. The specification is then used to generate parallel executables on a range of hardware platforms. Both shared-memory and distributed-memory sparse tiling is implemented in Firedrake [8] and the performance of a seismic exploration application, Seigen [77], is benchmarked on a CPU cluster. One other work of note by Sarje et al. [31] identified several performance-related challenges in unstructured-mesh-based applications when running on distributed systems. The wait times incurred by processes due to the imbalance of work and delays due to unstructured data access patterns are examined. It presents a cost model for a partition consisting of the communication and computation costs and attempts to identify a partitioning that minimizes load imbalances. Sarje et al. [31] also present data reordering techniques to increase the spatial and temporal locality of data in the memory hierarchy by using space-filling curves such as Morton, Hilbert [78, 79] in comparison to a Cuthill-McKee ordering.

Despite the work mentioned above, we perceive that the study on communication-avoidance work related to unstructured-mesh applications has key research gaps.

Firstly, as previously stated, the utilization of CA optimizations, especially within the context of the polyhedral model for unstructured-mesh applications, remains limited due to the intricate management of irregular dependencies that arise from the explicit connectivity of unstructured-meshes. Importantly, the analysis of these dependencies must occur dynamically at runtime. Furthermore, these indirect data accesses cause data races, leading to complex halo structures, which are the additional mesh elements exchanged among the processes for calculations. These halo structures need to be arbitrarily extendable due to the varying loop counts in loop-chains, which we do not find in the available developed frameworks.

Secondly, to our knowledge, all the unstructured-mesh-based CA work has been applied or demonstrated on small-scale applications and benchmarks. Research on their application to large-scale, real-world codes or solvers is practically non-existent. Understandably developing a practical implementation to apply these techniques to such applications is challenging. There is a compelling need for techniques such as DSLs to apply these optimizations automatically to substantially complex unstructured-mesh-based applications.

Thirdly, we found that utilizing sparse tiling simultaneously at both levels of parallelism - on-node/shared-memory (OpenMP threads and CUDA) and distributed-memory parallelism (over MPI) - has not been demonstrated before for unstructured-mesh-based applications.

Finally, due to the indirect data accesses in these types of applications, it is not possible to determine the profitability of these optimizations through compile-time or static analysis, necessitating runtime analysis. In some cases, performance may degrade on some loop-chains due to these CA techniques. Hence, it is important to develop techniques to model the performance using analytic modeling to support and guide the optimizations.

These open strands of work motivate the research in this thesis. We examine sparse tiling on large-scale applications, develop a performance portable framework that supports the execution of CA techniques on both CPU and GPU clusters, carry out extensive performance benchmarking and analysis on the empirical results, and study the profitability of the optimizations by developing analytical models.

1.2 Thesis Contributions

More specifically, the principal contributions of the thesis are as follows:

- **Integration of shared-memory CA with OP2 (Chapter 3):** We analyze shared-memory communication-avoiding and data-movement reducing techniques such as loop-chaining, sparse tiling, and cache-blocking tiling using the OP2 version of standard benchmarking applications and mini apps. We integrate the SLOPE library [8], which facilitates these techniques with the OP2 DSL with key improvements to its coloring algorithm for these evaluations. We generate roofline graphs and utilize performance analysis tools to analyze the empirical performance improvements gained with these experiments.

- **Design of a novel distributed-memory CA back-end (Chapter 4):** We design a novel CA back-end for the OP2 DSL, focusing on its distributed-memory parallel operation and reason about techniques for reducing the number of MPI send-receive messages exchanged during the execution of a sequence of consecutive loops, a *loop-chain*. A key new feature of the back-end is its ability to run standard loops over unstructured-mesh sets interspersed with selected/annotated loop-chains to obtain the best overall performance for the application.

- **Analytical modeling of loop-chain performance with CA (Chapter 4, 5, and 6):** The performance of loop-chains with CA is analytically modeled. The model is developed by investigating the careful trade-off of increasing computations at shared MPI halos to satisfy loop dependencies in place of data movement via message passing. The analytic model provides techniques to characterize the loop-chains and insights if a given loop-chain will benefit from the CA back-end of OP2.

- **Integrating shared- & distributed-memory CA (Chapter 5 and 6):** The distributed-memory back-end is combined with shared-memory communications avoiding and data-movement reducing techniques, specifically (1) sparse tiling, a cache-blocking tiling optimization method by integrating the SLOPE [8] library utilizing OpenMP threads on multi-core CPUs and (2) leveraging reduced MPI message passing when executing in a cluster of GPUs, enabling lower overheads in GPU-to-GPU communication with CUDA.
- **Real-world, large-scale application benchmarking with CA (Chapter 3, 4, 5, and 6):** Finally, the CA back-end and extensions are applied to the Hydra CFD application, using its recently re-engineered OP2 version [80], OP2 Hydra, for a number of use cases, with mesh sizes of 8M and 24M. Benchmarking is carried out on the ARCHER2 supercomputer, a Cray-HPE EX system with AMD EPYC 7742 cores, and the Cirrus GPU cluster with NVIDIA V100 GPUs at EPCC.

Results indicate significant performance gains: up to 65% on select node counts on ARCHER2 (CPU) and Cirrus (GPU) clusters. However, they also point to the need to carefully select which loop-chains have CA optimizations turned on, as in some chains they lead to performance degradation over the non-CA version. To our knowledge, our work presents one of the few, if not the only practical implementation of CA techniques for a large-scale production code, at the size and scale of Hydra, from the unstructured-mesh domain together with benchmarking and performance analysis on *both* multi-core (CPU) and many-core (GPU) cluster systems. Reasoning about the viability and profitability of these optimizations for real-world codes is also novel, particularly through the development of an analytic model. Our work provides insights into the key determinants and characteristics of a given general unstructured-mesh loop-chain that can lead to performance benefits with CA optimizations.

In our research, we are implementing CA enhancements using OP2 as the main DSL. While conducting the literature survey, we discovered that only a few libraries/DSLs support execution with unstructured-mesh applications, and Firedrake [59, 60] is one such DSL. However, to apply these enhancements to an application through Firedrake, the application must be written using the UFL (Unified Form Language). Further, our primary interest lies in Hydra [63], which has a version developed with the OP2 DSL [81]. Our main aim is to apply these CA enhancements to the Hydra application eventually. Several other applications, such as Airfoil, MG-CFD, and Volna, have already been converted to their OP2 versions, which can be used to apply CA enhancements using the OP2 DSL. Representing a set of loops using the OP2 API makes it easier to decide on the appropriate CA enhancements. More importantly, the automatic code generation support existing with the OP2 DSL will help us apply these enhancements effectively and efficiently to these applications. Due to these reasons, we are choosing the OP2 DSL for the implementation of these CA enhancements in our research. However, we believe that these insights are more broadly applicable, even to applications developed without OP2.

On the other hand, the work will also demonstrate how a high-level abstraction framework such as OP2 can seamlessly deliver these complex and exotic code transformations to real-world applications without affecting the science source, maintaining performance portability.

The shared-memory parallelization performance of the SLOPE library [37] analysis indicates that implementing cache-blocking tiling can lead to noteworthy performance enhancements on shared-memory systems, with select configurations demonstrating gains more than 40% over the respective OP2 OpenMP version of the application on the selected computing platforms. Nevertheless, the effectiveness of cache-blocking tiling is heavily influenced by various factors, such as the mesh partitioning approach, loop fusion strategy, and tile size. It is crucial to carefully adjust these variables to optimize the shared-memory parallelism performance for every application and platform.

Analysis of communication-avoidance with integrated shared-memory and distributed-memory parallelism strategies showed that the combination of these does not deliver better performance gains than the best-performing MPI version of the application with communication-avoidance. The delay in thread-synchronizations in the OpenMP parallelized sections of the code significantly impacts performance. However, applying communication-avoidance enhancements in a GPU cluster to achieve reduced MPI communication and lower overheads in GPU-to-GPU communication yielded considerably better performance improvements, achieving up to 65% with GPU on-node parallelizations, as previously stated.

1.3 Thesis Overview

This thesis consists of seven chapters, with the current chapter outlining the motivation and problems addressed, as well as the specific contribution of this research. The rest of the thesis is organized as follows:

Chapter 2 provides a literature survey related to HPC, including computer architectures, performance metrics, fundamental laws and theories, parallel programming models, performance optimization techniques, and domain-specific languages and frameworks in HPC. Overall, this chapter offers an overview of existing work related to communication-avoidance enhancements and a foundation for the thesis.

Chapter 3 analyzes the performance benefits of the SLOPE library [37] for shared-memory parallelism using four main applications on three computing environments. Airfoil [82], MG-CFD [83], Volna [84], and OP2 Hydra [63] are used as applications and two Intel-based computing nodes, Scyrus and Telos and one AMD-based supercomputer, ARCHER2, are used for performance analysis.

Chapter 4 explains the development of the communication-avoidance framework designed for distributed-memory systems for the OP2 DSL. This describes the changes made to the OP2 DSL when introducing the novel CA framework, explaining how the new OP2 framework can automatically generate CA optimizations for loop-chains. Additionally, this chapter details the analytical performance model for the newly developed CA framework for the OP2 DSL, which is used to identify the profitable loop-chains for CA optimizations. This model can be used to analytically model the runtime of the existing OP2 MPI execution and new CA back-end-based execution of loop-chains. We also present a comprehensive performance analysis of the newly developed CA framework using the two applications, MG-CFD [83] and Hydra [63].

Chapter 5 explains the integration of the novel CA back-end with shared-memory communication-avoidance and data-movement reducing optimizations through sparse tiling, a cache-blocking tiling optimization method by integrating the SLOPE [8] library utilizing OpenMP threads on multi-core CPUs. Performance benchmarking of MG-CFD [83] and Hydra [63] on the ARCHER2 supercomputer showcases the effectiveness of these integrated CA techniques.

Chapter 6 explores the synergy of the newly developed CA back-end with shared-memory communication-avoidance and data-movement reducing optimizations targetting reduced MPI message passing when operating in a cluster of GPUs, facilitating lower overheads in GPU-to-GPU communication through CUDA. The performance evaluation of MG-CFD [83] and Hydra [63] on the Cirrus GPU cluster highlights the successful application of these integrated CA techniques.

Chapter 7 gives the research conclusions and explains further enhancements for the current work with problems unsolved.

Chapter 2

Background

This chapter establishes the foundation of the thesis, giving a comprehensive explanation of: (1) High-Performance Computing (HPC) Architectures; (2) Performance Metrics and Optimization Laws; (3) Parallel Programming Models; (4) Performance Optimization Techniques; and, (5) Domain-Specific Languages (DSLs) and Frameworks in HPC.

2.1 High-Performance Computing (HPC) Architectures

High-performance computing enables us to tackle complex problems that demand extensive computing power and speed. With advanced computing resources and techniques, we can discover innovative solutions to the toughest challenges in scientific research, engineering, financial modeling, simulations, and data analysis. High-performance computing uses supercomputers, computer clusters, and parallel processing techniques to perform computations at much faster speeds than typical desktops, laptops, or server computers. The systems are *highly scalable*, allowing additional computing and storage resources to be added as needed to accommodate larger datasets, more complex simulations, and future computing needs. Efficient communication among computing nodes is critical for parallel processing. *High-speed interconnections* with low latencies are necessary for data exchange in HPC systems. Programmers must use various *programming models* and *specialized software*, such as MPI and OpenMP, to optimize HPC performance. Algorithms are developed to address challenges such as increased power consumption due to data movement among computing nodes. Programs running on HPC systems produce and consume massive amounts of data during calculations, requiring *large-scale storage* devices with low latencies that can be expanded according to changing demands. HPC enables scientists to *speed up time-to-solution* and gain insights into complex problems through advanced simulations that may not be possible with conventional computing.

Several computer architectures facilitate HPC. In this section, we will give an overview of the main computer architectures, with a specific emphasis on multi-core, many-core, and GPU architectures. Grasping the concept of these architectures is essential to unlock their enormous computational capacity.

2.1.1 Flynn's Taxonomy

In 1966, Michael Flynn described the computing process as the performance of a sequence of instructions on a set of data, in its essential form [85, 86]. He classified computers into four categories based on two *streams*, instruction and data, which are sequences of instructions or data seen by the machine during program execution as illustrated in Figure 2.1. The categories are:

- Single Instruction Stream - Single Data Stream (SISD)
- Single Instruction Stream - Multiple Data Stream (SIMD)
- Multiple Instruction Stream - Single Data Stream (MISD)
- Multiple Instruction Stream - Multiple Data Stream (MIMD)

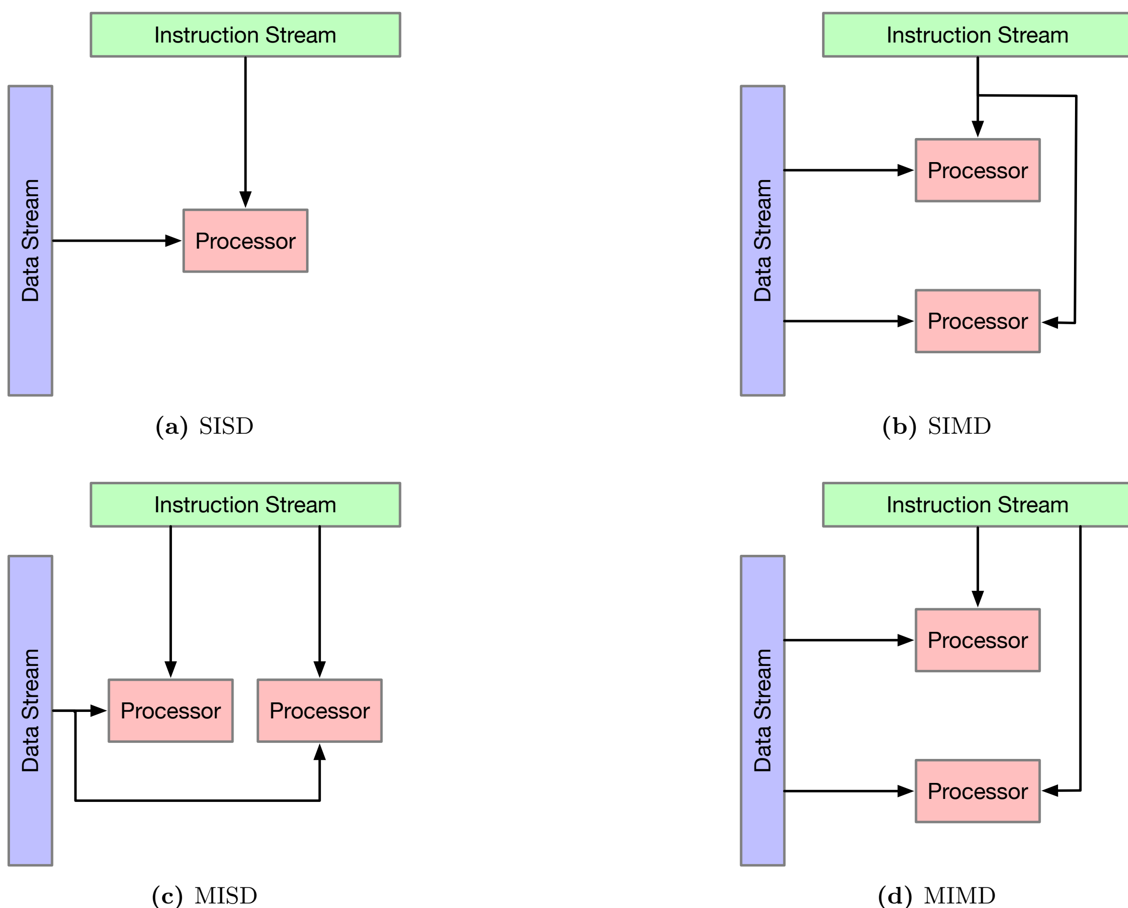


Figure 2.1: Flynn's Taxonomy

SISD In a computer with only one processor, there is only one stream of instructions from the program that is applied to a single stream of data. Although this concept is straightforward, improving the performance of such a system is challenging nowadays as we have almost reached the limit of frequency scaling for processors.

SIMD A program's single instruction stream is sent to multiple processor cores to be applied simultaneously to various data streams within those processor cores. This method is beneficial because most applications function on data arrays that require improved performance. The vectorization mechanism is used in today's CPUs to accomplish this. Intel's Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are examples of this, which allow for larger registers and more complex instructions. This mechanism is also present in modern GPU processors.

MISD Some computing systems and algorithms designed for redundancy use the mechanism of applying multiple instructions to the same data within a single clock cycle, which is not commonly seen in HPC.

MIMD Several programs, or even multiple instances of the same program, can run independently on different processors. This allows for multiple instructions to be executed on various data streams simultaneously within a single clock cycle. This approach is frequently used in high-performance computing (HPC) systems and requires message-passing and synchronization mechanisms, such as MPI, to ensure the output of the algorithms or programs is correctly finalized.

It is important to note that Flynn's taxonomy has evolved, and some modern architectures may have characteristics that blur the boundaries between categories. Some of the extensions or variations of Flynn's taxonomy include:

- Single Instruction - Multiple Thread (SIMT)
- Single Program - Multiple Data (SPMD)
- Multiple Program - Multiple Data (MPMD)

SIMT This represents a blend of single instruction (SIMD) and multiple instruction (MIMD) models where a single instruction is applied to multiple threads simultaneously. Each thread operates on its own set of data, often associated with modern GPUs. It efficiently enables parallel processing, especially in graphics rendering and scientific simulations, showcasing the evolution of parallel computing architectures.

SPMD This is a subtype of MIMD where multiple processors or threads execute the same program but may operate on different sets of data. This model is commonly associated with parallel programming paradigms, such as those used in parallel computing environments or distributed systems.

MPMD This is another form of MIMD where different processors or threads execute distinct programs concurrently, each working on its specific set of data. This model is often seen in heterogeneous computing environments, where diverse tasks are handled by different processing units.

2.1.2 Multi-Core and Many-Core Architectures

After the end of the frequency scaling of the processors around 2004, there was a shift in focus towards massive parallelism by increasing the processing core counts in a processor which helps to increase the overall performance of applications. Multi-core and many-core processors have been introduced to enhance computing systems' efficiency and performance. The distinction between these processors is based on the number of cores, although the exact boundary is unclear.

Most modern computers, smartphones, and other devices are equipped with multi-core processors. These processors have two to eight cores on a physical chip, which can perform instructions independently. By sharing resources such as cache memory and memory-controlling mechanisms, these cores can work together to boost system performance. They are also capable of carrying out load-balancing tasks among them to improve overall efficiency.

As technology advances, many-core systems are becoming more prevalent in HPC systems. These systems integrate dozens to several hundred cores on the same physical chip, allowing multiple cores to work in parallel to solve complex computational problems across various domains. These cores have a simple architecture and share resources such as caches and registers. Many-core processors are mainly used in supercomputers and specialized computing clusters for tasks such as simulations, artificial intelligence work, and data-intensive processing. With thousands of computing cores working together, these systems can handle the most challenging computational problems with ease.

Both these architectures are built to support parallel computing and many-core systems are specially employed in high-performance demanding systems.

2.1.3 GPU Architecture

Image and video processing requires a significant amount of parallel operations. To address this, Graphics Processing Units (GPUs) were developed. These GPUs consist of numerous efficient and dedicated cores, which are ideal for handling these types of applications. These cores operate at lower frequencies, have smaller caches and enable a large number of threads to run in parallel, unlike CPU cores. This architecture facilitates the SIMD (Single Instruction Stream - Multiple Data Stream) paradigm, where the same operation is performed on a vector of contiguous data. A parallel PCI-Express Interconnect connects these GPUs to the CPU.

At first, GPU architectures were primarily used for graphical purposes, such as in video games. However, their potential for high performance in parallel applications, especially in matrix and vector computations, led to their increased popularity. To simplify GPU programming, various languages have been developed, such as CUDA for NVIDIA GPUs and OpenCL for a wider range of GPUs, including both NVIDIA and AMD architectures. Nowadays, GPUs have advanced to support general-purpose massively parallel computations, moving them closer to CPUs in terms of functionality. In fact, around

Figure 2.2: NVIDIA Tesla V100 architecture [2]

35% of modern supercomputers in the Top 500 are now equipped with GPUs to support general-purpose applications [23].

In this thesis, we have NVIDIA Tesla V100 as the primary GPU type in one of our testing platforms, Cirrus [87]. The architecture of this GPU is shown in Figure 2.2. It is based on the Volta architecture and consists of 80 Streaming Multiprocessors (SMs) grouped in six Graphical Processing Clusters (GPC) [2]. It also has a shared L2 cache of 6MB and a 4096-bit memory interface. Each SMM has 96KB shared-memory and 64 FP32 cores, resulting in a total of 5120 cores in the GPU, with 21.1 billion transistors. The Tesla V100 has 16 GB or 32 GB of high-bandwidth (HBM) memory, providing excellent memory capacity and bandwidth for handling large datasets and complex computations. Its NVLink technology enables seamless GPU-to-GPU communication in multi-GPU configurations. Furthermore, the Tesla V100 supports hardware-accelerated deep learning and AI workloads with its Tensor Cores.

However, the latest available NVIDIA GPU architecture at the time of writing this thesis is the ADA LOVELACE GPU architecture [88].

2.2 Performance Metrics and Optimization Laws

Measuring and optimizing the performance of an HPC system is a critical task. In this section, we will define key performance metrics and explore optimization laws such as Amdahl's Law and Gustafson's Law, which guide our understanding of the scalability and efficiency of an application. We also examine the roofline model, which assists in identifying the bottlenecks that affect an application's use of system bandwidth and processing capacity. By doing so, we can work towards enhancing the overall system performance.

2.2.1 Key Performance Metrics

In computing, engineering, and other fields, performance metrics are numerical measures used to assess a system's efficiency, effectiveness, and quality, process, or operation. These metrics offer unbiased data and insights to determine how well a task or component is performing compared to previous known statuses. Performance metrics are essential for enhancing systems, identifying problem areas, and making informed decisions to achieve improvements. In this thesis, we refer to key metrics such as execution time, setup time, speedup, scalability, efficiency, and latency, and this section explains those metrics.

Execution Time Evaluating the effectiveness of systems and processes is crucial, and performance metrics play a pivotal role in this evaluation. Runtime, also known as Execution Time, measures the time taken by a computational process to reach completion. It serves as a fundamental benchmark for optimizing the efficiency and responsiveness of software applications, scientific simulations, and computational tasks. Reducing execution time signifies improved performance and user experience.

Setup Time The essential preparatory period before a task or process begins, influencing workflow efficiency and system readiness, is represented by Setup Time. During our thesis experiments, we found that certain enhancements aimed at reducing execution time can increase the setup time. However, we observed that this increase in setup time is offset by the large number of iterations that these applications undergo.

Speedup Performance gain achieved by parallelizing tasks compared to their sequential counterparts is measured by Speedup. High speedup values indicate effective utilization of multiple processing units, which is crucial in accelerating resource-intensive computations.

Scalability The system's ability to adapt to increasing workloads without proportional performance degradation is estimated by Scalability. We can identify whether an application is capable of accommodating dynamic workloads according to its scalability measures.

Efficiency Effectively utilizing resources such as CPU, memory, and energy to accomplish tasks is the essence of Efficiency. Given the limited availability of these resources, it is important to increase efficiency in our applications and systems.

Latency Measuring the time delay between taking an action and receiving a response, known as Latency, is a crucial metric for analyzing performance in real-time systems and networks. Minimizing latency is essential for ensuring uninterrupted interactions and data transmission.

When determining the effectiveness of an application or task, it is crucial to consider performance metrics. We utilized various metrics throughout our thesis to draw conclusions.

2.2.2 Amdahl's Law and Gustafson's Law

The speedup of a given program on a multi-processor system is defined as the ratio between the time spent on running the best sequential version of the program on a single processor to the time spent on running the parallel version of the program on a multi-processor system.

$$\text{Speedup}, S(n) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multi-processor with } n \text{ processors}} = \frac{t_s}{t_p} \tag{2.1}$$

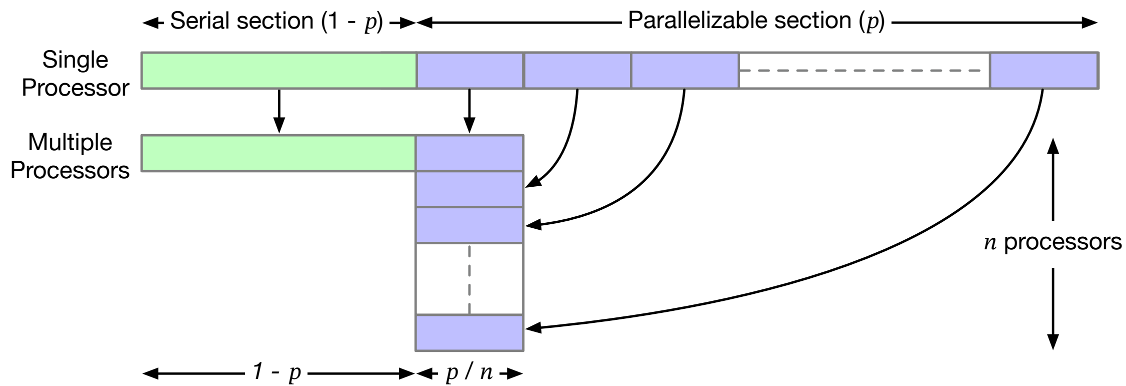


Figure 2.3: Amdahl's Law

Gene Amdahl proposed a method in 1967 to identify the theoretical speedups in latency that can be achieved by considering the serial and parallel parts of the program in a system where the resources can be improved. The performance of the serial section of the program can be improved by increasing the clock frequency or improving the internal layouts of the processor whereas the parallel sections can be improved by increasing the number of computing cores in the system as shown in Figure 2.3.

$$S(n) = \frac{1}{\underbrace{(1-p)}_{\substack{\text{proportion of time}\\ \text{spent on serial parts}}} + \underbrace{\frac{p}{n}}_{\substack{\text{proportion of time}\\ \text{spent on parallel parts}}}} \tag{2.2}$$

where:

$S(n)$ – speedup of the program

p – proportion of the program that can be parallelized

n – number of processors or cores

John L. Gustafson formulated Gustafson's Law in 1988, which is a principle in parallel computing that provides a different perspective on the performance of parallel systems compared to Amdahl's Law. Gustafson's Law focuses on the scalability of parallel systems as the number of processors or cores increases.

The central idea behind Gustafson's Law is to adjust the problem size as more processors are added, instead of fixing the problem size and analyzing how execution time changes with varying numbers of processors like Amdahl's Law. This means that parallel systems should be designed to solve larger problems in the same amount of time as the number of processors increases.

Mathematically, Gustafson's Law can be represented as follows:

$$S(n) = 1 + (n - 1) \cdot p \tag{2.3}$$

where:

$$\begin{aligned} S(n) &- \text{speedup of the program} \\ p &- \text{proportion of the program that can be parallelized} \\ n &- \text{number of processors or cores} \end{aligned}$$

In this equation, as the number of processors (n) increases, the speedup ($S(n)$) also increases proportionally. The formula assumes that the fraction of the program that can be parallelized (p) remains constant, which means that the portion of the program that can be efficiently parallelized grows with the number of processors.

Gustafson's Law is often applied in high-performance computing and parallel processing environments where the goal is to solve more substantial problems efficiently as more computational resources become available. This approach aligns with the idea that many real-world applications can benefit from parallelization and by adapting the problem size, we can take full advantage of the available computational resources to solve larger and more complex problems in a reasonable amount of time.

2.2.3 Roofline Model

The Roofline Model [89] is a performance modeling and optimization framework used in high-performance computing. It is used to visualize and analyze the performance of computational kernels or applications on modern computer architectures. It helps to identify performance bottlenecks and opportunities for optimization.

The Roofline Model is simple yet powerful, as it ties together floating point operation (FLOP) performance, arithmetic intensity (AI), and memory performance in a 2D graph. One way to measure peak memory performance is by using STREAM benchmarks [90, 91]. By measuring AI for a given code snippet or kernel, one can find a point on the horizontal axis. The performance of that kernel can then be determined by the vertical line that passes through that point. For a given AI, the roofline represents the maximum FLOP performance that can be achieved by the code on the specific hardware architecture used. This can be calculated using the formula:

$$\text{Max Attainable GFLOPs/s} = \min \begin{cases} \text{Peak Memory Bandwidth (BW)} \times \text{AI} \\ \text{Peak FLOP Performance} \end{cases} \tag{2.4}$$

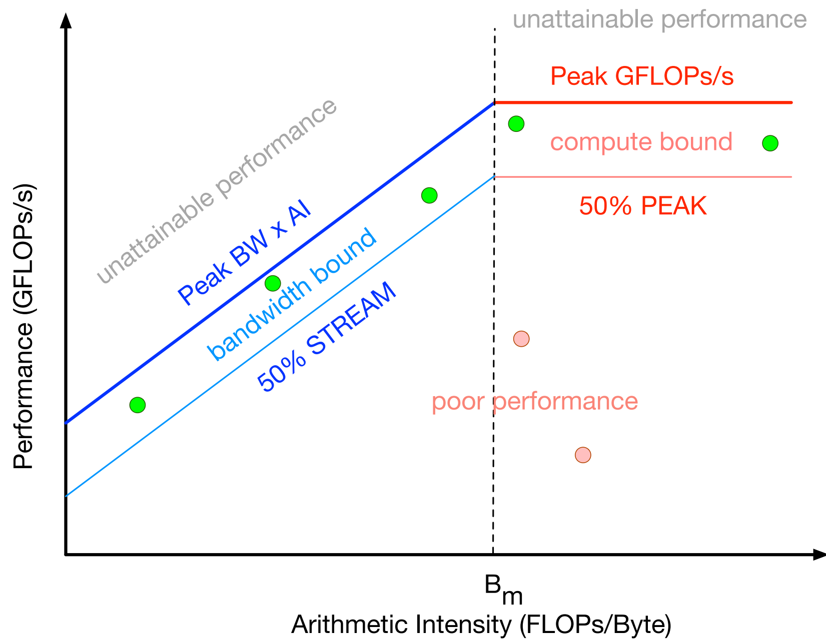


Figure 2.4: Roofline Model

In the Roofline Model graph shown in Figure 2.4:

- The X-axis represents the computational/arithmetic intensity (FLOPs/Byte).
- The Y-axis represents the achievable performance (GFLOPs/s).
- The Peak Performance Line represents the hardware's theoretical peak performance.
- Different performance limits can be represented by rooflines. There are four distinct performance regions that can be identified:
  - Unattainable performance
  - Bandwidth bound performance
  - Compute bound performance
  - Poor performance
- Data points can be plotted to show how specific applications or kernels perform in relation to these limits.

The ridge point on the roofline model graph is characterized by its abscissa, which is equal to the machine balance. The machine balance represents the point at which the computational resources for computation and memory access are well-matched. At the ridge point, the performance of a code on a specific hardware architecture is optimized, indicating a balance between computation and memory access that maximizes the achievable FLOP performance. In other words, the ridge point represents an ideal balance where the computational resources are effectively utilized without being bottlenecked by either computation or memory bandwidth. Kernels that lie close to the roofline are making good use of the hardware resources. It is important to note that kernels with low GFLOPs/s

can still make good use of a machine if they have a high % STREAM. Conversely, kernels can have relatively high GFLOPs/s but still make poor use of a machine if they have a low % Peak GFLOPs/s.

The Roofline Model helps developers identify if an application is under-utilizing the hardware's potential and provides insights into where optimization efforts should be focused, such as improving computational intensity or reducing memory bandwidth requirements.

2.3 Parallel Programming Models

In modern times, CPUs have evolved to include advanced multi-core technology. In the past, it was customary to have one process attached to one computing core. However, with the addition of more cores within a CPU, using the traditional MPI approach resulted in more overhead when communicating between processes. To mitigate this issue, we can increase the usage of shared-memory and caches by implementing *shared-memory parallelism*. However, it is important to be mindful when replacing message passing with multi-threaded execution since both approaches introduce different overheads. The *pragma*-based OpenMP approach is a popular option for parallelizing work with multiple threads due to its ease of implementation. Furthermore, several other task-based parallelization methods have been introduced as alternatives to OpenMP-based parallelism.

To fully utilize the powerful computing capabilities of parallel architectures, we must use various programming models. The *distributed-memory parallelization* model is the highest-level approach, distributing the problem across the computing nodes of the system. The de facto standard for this has been the Message Passing Interface (MPI) for a long time. However, the Partitioned Global Address Space (PGAS) model has emerged as an alternative to MPI. This approach divides a shared-memory region across distributed processes and provides direct remote access to them.

In recent times, there have been notable advancements in General Purpose Graphics Processor Units (GPGPUs), leading to the emergence of various programming models. These models utilize specific languages tailored to harness the computing power of these architectures. Notably, the CUDA programming language was developed specifically for NVIDIA GPUs, while OpenCL is used for programming both NVIDIA- and AMD-based GPUs. These parallel programming approaches aim to get the most out of GPGPUs.

2.3.1 Shared-Memory Systems

In a shared-memory system, all the processors or computing units access a shared global memory. Processors use shared variables to communicate with each other and they can read and write to any place in the *shared-memory space*. Sometimes, the memory units that form the common memory system may not be available in the same location. Still, they share a common address space when addressing the locations in the memory.

These systems are relatively *simple* and provide simple programming approaches. The *data sharing mechanisms* among the processors and *shared data structures* are not complex due to the usage of common address space.

There will be *scalability issues* due to the complexities arising in interconnections and memory-controlling mechanisms when increasing the number of computing cores in the system. It will become *challenging to keep memory consistency* as multiple processors try to access the same memory location simultaneously creating race conditions.

OpenMP

OpenMP is a popular shared-memory parallelization strategy that uses `pragma`s to create and manage threads. Its ease of use and effectiveness make it widely used in applications. For example, to parallelize a for loop, we use the *omp parallel for* `pragma`. This splits the iterations into chunks and assigns them to different threads for parallel execution. At the end of the parallel section, there is a thread-synchronization step where the solutions are aggregated for the final solution.

The effectiveness and scalability of OpenMP parallelism is largely impacted by the sequential parts of the program. However, this approach is suitable for regular applications with loops that have dense computations [92].

2.3.2 Distributed-Memory Systems

In a distributed-memory system, each processor has a dedicated memory space of which the management is assigned to the individual processor, and the processors communicate with each other by exchanging messages.

These systems are easily *scalable* to a large number of processors since each individual processor possesses and manages its own memory which in return gives greater *flexibility* in configuring the system.

Due to the nature of the system, various complex programming models such as MPI must be used to exchange messages among the processors to exploit the optimum computing power of the system. There are challenges in *data distribution* and *synchronization* in these systems due to the distributed execution of the programs. Some additional overheads coming due to message passing and *communication* among processors may affect the performance of programs running on distributed-memory systems.

In general, there will be a combination of both the shared- and distribute-memory systems leveraging their advantages, minimizing the issues in scalability and memory management and giving a simple programming model enhancing the performance of applications.

MPI

MPI, or the Message Passing Interface, has become the standard for achieving distributed-memory parallelism. There are several MPI implementations available, including MPICH [93], OpenMPI [94], and Intel-MPI [95]. The concept behind MPI is to offer an interface for parallelizing work by creating multiple processes known as *MPI ranks* or *tasks*.

The standard MPI communication model supports point-to-point communication between two processes. This involves exchanging data through private memory regions. Both the sender and receiver must participate in the process, with the sender explicitly indicating the message's memory location and size. The message is then copied to an internal communication buffer and sent to the receiver, where it is copied back to the relevant memory location. To address latency issues, two message exchange mechanisms are available: blocking and non-blocking.

In blocking-MPI communication, the functions `MPI_Send` and `MPI_Recv` are utilized. The call to `MPI_Send` function does not return until the message being sent has been copied to the receiver's buffer and communication has been completed. For non-blocking communications, the functions `MPI_Isend` and `MPI_Irecv` are used. These functions return immediately, allowing the sender to continue with their computations or processing while the receiver completes the communication. The function `MPI_Wait` ensures the completion of the MPI communication that was initiated.

MPI uses two message exchange protocols, known as *Eager* and *Rendezvous*, for both blocking and non-blocking communication. The *Eager* protocol is suitable for small message exchanges, while *Rendezvous* is ideal for larger message transfers. The protocol selection can be influenced by setting environment variables that specify message size limits.

Other collective and one-sided MPI communication methods can be employed through functions such as `MPI_Gather`, `MPI_Scatter`, `MPI_Put`, `MPI_Get`, and various others.

PGAS

Partitioned Global Address Space (PGAS) is another technique used to leverage distributed-memory parallelism. It utilizes a global memory space that is partitioned among the distributed processes. Each process has its own local space within this global address space, enabling direct read and write.

PGAS is a highly effective solution for unstructured-mesh applications with irregular data access patterns [96]. With its one-sided communication approach, data movement and memory duplication are reduced by writing directly to the receiver's memory location. This approach utilizes a Remote Direct Memory Access (RDMA) compatible network to minimize CPU resources, with the network controller handling communication. Furthermore, there is no need for synchronization among processes due to this approach, as the

receiver only needs to check whether incoming data is received in the buffer. Communication is also carried out in parallel with computations, preventing delays caused by communication.

Although PGAS has some benefits, there are also some drawbacks to using it. One of these is that the sender must know the receiver's local information to package its data, which can be an added burden. Additionally, many existing codes are built around MPI communication, so integrating PGAS would require significant changes to the existing frameworks that could be error-prone and resource-intensive.

Task-Based Parallelism

Task-based parallelism involves dividing a computation into a sequence of independent tasks, each representing a specific unit of work. These tasks can be executed simultaneously using multi-core processors, GPUs and even distributed computing clusters. This approach is ideal for applications with irregular and unbalanced workloads. Its capacity to efficiently use available computing resources and adapt to changing workloads makes it a powerful tool. Numerous frameworks have been developed to accomplish task-based parallelism, such as StarPU [97] and PaRSEC [98].

2.4 Performance Optimization Techniques

To make the most of the computing and bandwidth resources in a system, we need to use different performance-optimizing techniques. This involves arranging computations and communications in the application to match the running platform's architecture. In this section, we will discuss loop transformation and communication optimization methods, which can enhance an application's performance.

2.4.1 Loop Transformation Techniques

Here, we will explore various loop transformation techniques that standard compilers and libraries use to improve memory locality and parallelism in loop nests. A *loop nest* is a situation in computer programming where one or more loops (typically for, while, or do-while loops) are nested within another loop. To learn more about these transformations, we primarily refer to the work of *Bacon et al.* [64]. We may also refer to other relevant research materials to enhance our discussion.

Different compilers and libraries apply unique techniques at varying levels based on their capabilities. Some only consider *perfect* loop nests, which have all statements inside the innermost loop. The transformations we discuss mainly involve loop reordering, which changes the execution order of loop iterations. To ensure the loop nest's semantics remain unaltered, a *data dependence analysis* is conducted before implementing any transformations. If no dependencies are found, the loop nest can be made completely parallelizable.

Loop Interchange

To improve data locality, vectorization, or parallelism, it may be necessary to switch the position of a loop within a perfect loop nest. However, it is important to be cautious and prevent the benefits of one switch from being negated by another. If dealing with imperfectly nested loops, additional customized changes may be required.

Loop Skewing

This transformation was introduced to facilitate the array accesses in loops for wavefront propagation computations [99, 100]. Array accesses are transformed like wave propagation in the outer loop nest with transformation functions such as $w = f(w-1, w+1)$. This is mostly associated with a loop interchange and the target polyhedron of the iteration space becomes skewed.

Loop Reversal

Loop reversal changes the direction that a loop nest is being executed such as if it was executed from the iteration, 0 to $n-1$, now it is from the iteration, $n-1$ to 0. This helps various programming languages and processor architectures to avoid branching conditions and temporary variables utilized during the loop nest execution. Loop reversal can sometimes pave the way to other loop enhancements such as loop interchange.

Strip Mining

This transformation is commonly used to adjust the level of parallel execution in a loop nest. It is particularly useful for vectorization scenarios where the number of parallel operations can be controlled through this transformation. When one instruction is applied to multiple data (SIMD), the granularity of the operation can be determined by this transformation. However, if the strip size is not a multiple of the loop size, a clean-up process will be required at the end of the loop execution.

Loop Distribution

The process of distributing or *fission/splitting* loops is primarily used for three purposes: (i) create perfect loop nests, (ii) reduce the memory footprint when multiple statements inside the same loop nest, and (iii) increase memory locality and avoid dependencies among the statements.

Loop Fusion

The process of combining two or more loops is known as loop fusion, which is the opposite of loop fission. The main goals of loop fusion are to (i) minimize the overhead of loops in a program, (ii) improve data and cache locality by fusing them properly, and (iii) increase

the number of instructions executed in parallel. Loop fusion is straightforward when the loop bounds of the fusing loops are the same. However, if the loop bounds differ, only certain parts of the loops can be fused, while the remaining iterations are placed in a separate loop.

Loop Tiling

Enhancing data locality in high-performance computing can be achieved through tiling, a useful technique. Loop tiling, in particular, involves dividing the iteration space of a loop into smaller sections to keep them in the cache until they are needed for calculations. This technique is helpful in reducing cache size requirements, as larger arrays can be broken down into smaller blocks that easily fit into the cache. However, data dependency analysis is crucial, just as with loop fusion. While tiling can be done manually for smaller applications such as matrix multiplications, automating tiling for larger-scale applications is becoming increasingly important, as highlighted by Reguly et al. [36]. The code example in Listing 2.1 shows a tiling example of matrix multiplication.

```
1 // Consider the matrices as A_{N×N} B_{N×N} and C_{N×N}
2
3  for (int i = 0; i < N; i++)
4    for (int j = 0; j < N; j++)
5      for (int k = 0; k < N; k++)
6        A[i][j] += B[i][k] * C[k][j];
7
8  // Take the tile size as w * w
9
10 for (int i = 0; i < N; i += w)
11   for (int j = 0; j < N; j += w)
12     for (int k = 0; k < N; k += w)
13       for (int i1 = i; i1 < min(i + w, N); i1++)
14         for (int j1 = j; j1 < min(j + w, N); j1++)
15           for (int k1 = k; k1 < min(k + w, N); k1++)
16             A[i1][j1] += B[i1][k1] * C[k1][j1];
```

Listing 2.1: Loop tiling example

2.4.2 Communication Optimization

When working with programs in a distributed environment, it is essential to share and update data between multiple processes before performing calculations. Our discussion is based on the communication optimization methods presented by *Bacon et al.* [64]. To improve the performance of the system, it is important to maximize data reuse, minimize the amount of data being used, and make optimal use of parallelism. Our research will explore techniques to achieve these goals.

Message Vectorization

When executing a loop in a distributed system, it is not efficient to send elements one by one to other processes. Instead, each process calculates the loop bounds for itself and other processes. Once this is done, the elements can be exchanged before the loops are executed. This allows the exchanged array elements to be kept in a contiguous memory location, which enables efficient use of internal buffers. This can lead to other compiler optimizations, such as strip mining.

Message Coalescing

Coalescing, the grouping of overlapping and adjacent message data, enhances communication along with message vectorization.

Message Aggregation

When sending messages to another process, there are fixed costs involved, including communication startup time, waiting time, and message packing and unpacking overheads. To reduce the frequency of these costs, multiple messages can be combined and sent as a single message. To achieve this, the program should have mechanisms in place for packing multiple message data into the same message buffer and unpacking them to the relevant data buffers at the receiving end.

Collective Communication

This is about the optimized features introduced by libraries and frameworks for message communication such as *scatter*, *gather*, *all-gather*, and *broadcast*. Utilizing them appropriately in a program significantly improves performance.

Message Pipelining

During message transmission, it is possible for processes to continue calculations on data that do not rely on non-local data. This improves performance by reducing waiting time for data. Using non-blocking send and receive methods where possible optimizes computing resources. In our performance enhancements, we also utilize the technique of *latency-hiding* while performing the core computations of datasets.

Redundant Message Elimination

When programming, it is crucial to be mindful of the data dependencies and messages shared between processes to prevent unnecessary transmission of redundant messages. Duplicate information may be present in some messages, and the data may remain unchanged since the last exchange, making it unnecessary to transmit updates to the dataset again. By detecting these scenarios, we can significantly enhance performance, given that the cost of data exchange between processes is usually high.

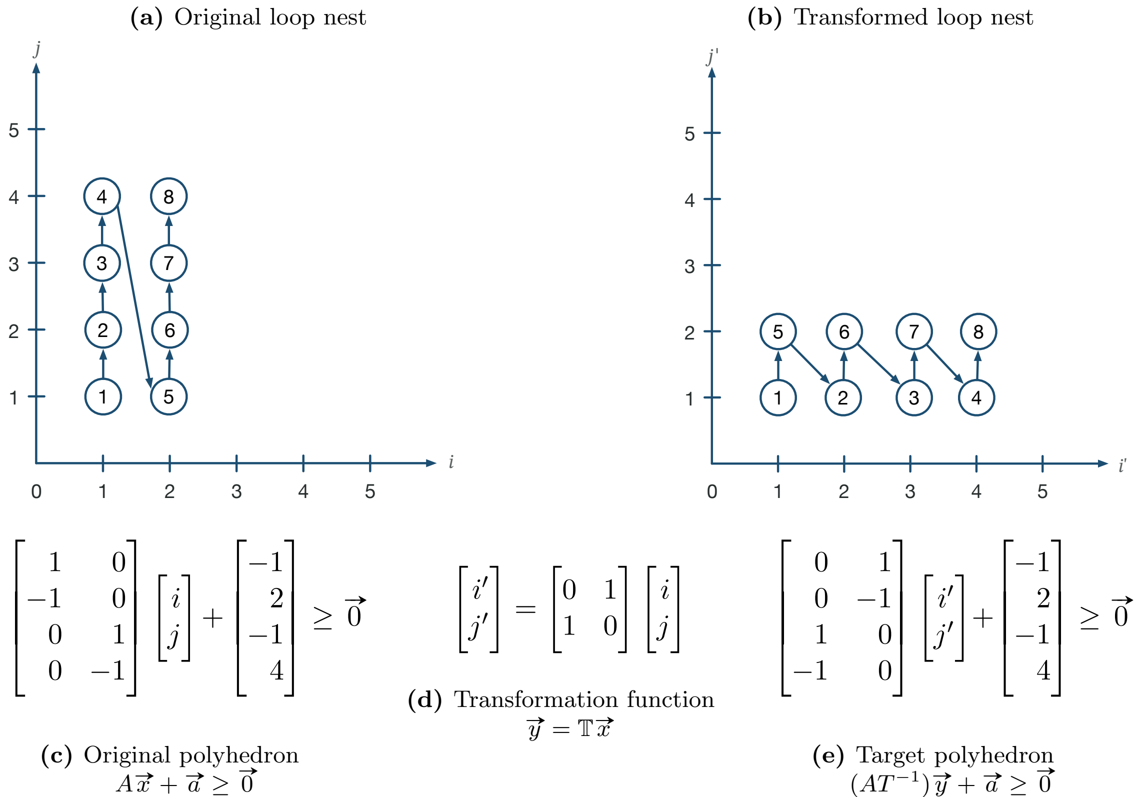
2.4.3 Polyhedral Model

```
1 for (int i = 1; i <= 2; i++) {
2   for (int j = 1; j <= 4; j++) {
3     A[i][j] = A[i][j] + 10;
4   }
5 }
```

$\overset{\mathbb{T}}{\Longrightarrow}$

```
1 for (int j = 1; j <= 4; j++) {
2   for (int i = 1; i <= 2; i++) {
3     A[j][i] = A[j][i] + 10;
4   }
5 }
```

(a) Original loop nest **(b)** Transformed loop nest



$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \\ -1 \\ 4 \end{bmatrix} \geq \vec{0}$$

(c) Original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

(d) Transformation function $\vec{y} = \mathbb{T}\vec{x}$

$$\begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \\ -1 \\ 4 \end{bmatrix} \geq \vec{0}$$

(e) Target polyhedron $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

Figure 2.5: Loop nest interchange transformation

We can use the polyhedral model as a way to analyze nested loop structures for applying the above-mentioned performance optimization methods, effectively. The polyhedral model or the polytope model is a well-established mathematical and geometrical representation of loop nests [5, 101]. It enables the iteration space of each statement to be represented through the lattice points of a polyhedron, an n-dimensional geometric object. This model is used in various frameworks such as Polly [5] as a powerful tool to optimize loop nests. However, it should be noted that the polyhedron model is primarily used to analyze affine loops. In other words, loop bounds and array references to data access of loop nests should be affine functions of loop iterators and other loop parameters, such as $f(i) = 4 * i + 5$ and $g(N) = 2 * N$, which are functions of the iterator, i and the element count, N of the loop nest. Non-linear functions, such as $f(i) = 4 * i * i + 5$ and $g(N) = 2 * N * i$, are considered non-affine functions and if they are involved in the loop bounds or array references, it is not possible to apply the polyhedral model for optimizations. Once the polyhedral model is built for a given loop nest, it is transformed through

affine mapping functions for various purposes, such as achieving memory enhancements and parallelism for efficiency. Such loop transformations include loop tiling, loop peeling, loop reversal, loop skewing, loop shifting and so on. It is important to note that in order to make any of the transformations valid, it has to preserve the semantics of the loop nest. For this reason, a careful data dependence analysis is performed on the transformed loop iteration space. Once the loop transformation is confirmed to be correct, it is applied through polyhedral scanning [102, 103, 104, 105, 106, 107].

Consider the example elaborated in Figure 2.5. Figure 2.5a shows a loop nest with affine loop bounds and array references which can be represented using the polyhedral model as in Figure 2.5c. Using the affine transformation function mentioned in Figure 2.5d, the original polyhedron is transformed to Figure 2.5e for the required optimizations. The resulting transformed loop nest is given in Figure 2.5b. This is a loop-interchange transformation that is used on several occasions to improve data cache locality in programs.

2.5 Domain-Specific Languages (DSLs) and Frameworks in HPC

In the fast-changing world of high-performance computing, maintaining code, achieving performance portability, and future-proofing are major challenges. To overcome these, Domain-Specific Languages and Active Libraries concentrate on a particular application domain and offer a high-level programming method. This allows them to use domain knowledge and achieve high performance on different hardware. Domain-specific languages (DSLs) and frameworks have recently gained traction in HPC as a way forward to alleviate these issues. There are two main classes of DSLs, (1) *standalone* DSLs that define a totally new language and (2) *embedded* DSLs that are hosted in a standard programming language using its syntax, features, and compiling tools. Most of the DSLs fall into the embedded category since it is very practical to utilize the existing efficient features of a standard programming language and introduce new features to support domain-specific tasks. Active libraries utilize their code generation facility to generate platform-specific optimized code for programs written using the API of that particular language. Examples of DSLs include OP2 [6] (which we use in this thesis) and Firedrake [59] for unstructured-mesh-based applications, OPS [36] and Devito [61] for structured-mesh-based applications, TiDA [108] as tiling data abstraction, STELLA [109] for structured-mesh stencil codes, Halide [110] for high-performance image processing, Pochoir [3] for implementing stencil computations on multi-core processors, and Patus [111] for earthquake simulations. Apart from that, there are several other frameworks developed for HPC applications that help to enhance the performance of an application such as LLVM's Polly [5] and Pluto [4].

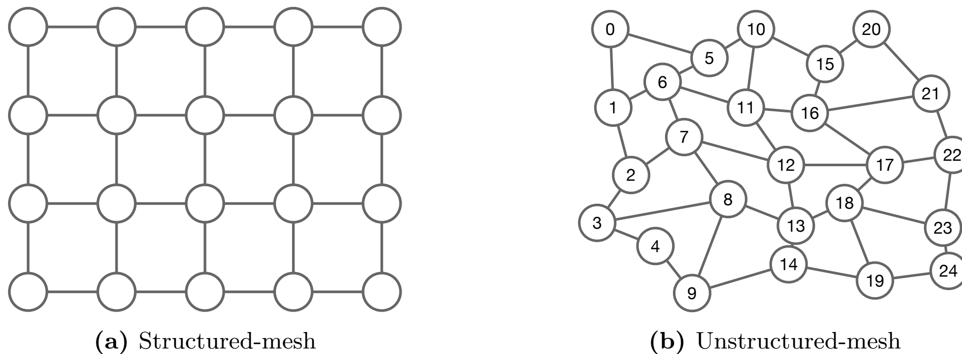
In this thesis, we utilize the OP2 DSL as the primary tool for implementing communication-avoidance optimizations. Therefore, we have a separate section dedicated to explaining the OP2 DSL after this section. This section covers other DSLs and frameworks, as well as necessary basics.

2.5.1 Mesh Types

In this section, we will explore different mesh types used in computational simulations. Mesh types are crucial not just for accurate simulations but also for designing domain-specific languages (DSLs) tailored to specific computational needs. These mesh structures form the basis of DSLs, enabling efficient data handling.

Meshing is the process of discretization of a surface or domain into smaller elements or cells. This is a fundamental concept in numerical methods such as finite element analysis (FEA) and computational fluid dynamics (CFD). This process helps to solve partial differential equations (PDEs) numerically.

Depending on the context, a mesh divides the geometry into a collection of discrete elements such as triangles, quadrilaterals, tetrahedrons, and so on. The shape and size of the elements should be decided mainly on the accuracy level of the expected numerical solution. In these physical problems, it is assumed that real-world physics happens at the intersections, along the edges or middle of the grid element or cell. In other terms, the calculations are performed on the data values assigned to the edges, nodes, or cells of the mesh. Based on the way of the mesh creation process, meshes are categorized into structured- and unstructured-meshes as shown in Figure 2.6.



(a) Structured-mesh **(b)** Unstructured-mesh

Figure 2.6: Differnet types of meshes

In a structured-mesh, the mesh elements are regularly arranged and the neighboring elements can be identified easily. The generation of a structured-mesh is easy and it is suitable for problems involving regular shapes such as pipes, cubes, and geometries which can be decomposed into regular shapes. Storing the mesh elements of such a mesh is memory efficient due to its regular nature. Most of the memory enhancement techniques can be implemented during the compile time of the program due to the definite mesh element arrangement.

In an unstructured-mesh, the mesh elements have various shapes such as triangles and tetrahedrons depending on the contours of the geometry being discretized. Storing of the mesh elements demands a higher memory footprint, due to its irregular nature. Neighboring mesh elements of this mesh type cannot be directly inferred, but have to be identified through indirect data accesses of the mesh storing data structure. These types of meshes can handle any type of geometry due to their element-storing pattern. Enhancements to data accesses in these types of meshes are challenging when compared to structured-meshes and those can be implemented at the runtime of the program which needs a lot of processing and memory resources.

2.5.2 Firedrake

Firedrake [59, 60] is a software framework that uses the Finite Element Method (FEM) to solve partial differential equations (PDEs) numerically. It is a well-known framework for its flexibility and ease of use in the computational science domain. Firedrake simplifies the development and execution of simulations by offering a domain-specific language for PDE expression and an automatic code generation framework.

Firedrake's FEM approach enables it to handle complex unstructured-meshes and accelerate simulations through the use of modern hardware such as multi-core CPUs and GPUs. One of its key features is automated code generation, which significantly reduces the coding effort required to run simulations. This allows researchers and engineers to focus on the mathematical aspects of their problems while simplifying the development process and encouraging other collaborations rather than focusing on optimizing code for individual computing platforms.

Overall, Firedrake is a valuable tool for computational scientists and engineers looking to solve PDEs numerically. It offers a user-friendly interface, automates code generation, and increases the efficiency of complex simulations.

2.5.3 Devito

Devito [61, 62] is a domain-specific language and code generation framework designed specifically for computational geophysics. It combines high-level symbolic representations with low-level code generation, providing a new approach to implementing complex numerical algorithms. One of Devito's key strengths is its ability to translate high-level symbolic algorithm descriptions into highly efficient, platform-independent code.

This ability to bridge the gap between mathematical representations and optimized code is significant. It simplifies the development of complex seismic simulations, releasing researchers and geophysicists to focus on the scientific aspects of their work.

Devito is also optimized to utilize modern hardware architectures, including CPUs and GPUs. This means that it not only simplifies development but also significantly speeds up the execution of seismic simulations. This acceleration is critical in geophysics, where processing large amounts of data efficiently is essential.

In summary, Devito's automated approach to stencil computations streamlines workflows, improves efficiency, and optimally utilizes the power of parallel and distributed computing resources. By simplifying complex seismic modeling and inversion tasks, Devito enables natural resource exploration.

2.5.4 TiDA

The TiDA (Tiling Data Abstraction) [108] offers a new approach to improve the efficiency of parallelism and data locality in computer programs, especially with the current trend of massive parallelism in modern hardware. Tiling is a technique that enhances the performance of loops by optimizing data access patterns. TiDA intends to create a long-lasting tiling abstraction that centralizes parameterized tiling information within array data types, with minimal changes to the existing source code.

TiDA incorporates data layout information into array data types, which allows the compiler and runtime system to manage parallelism, improve data locality, and schedule tasks more intelligently. This simplifies the optimization process for parallel execution and reduces the need for manual tuning.

2.5.5 STELLA

STELLA [109] is a domain-specific embedded language designed to facilitate the efficient implementation of stencil codes on structured grids. Stencil codes are fundamental in scientific simulations and numerical computations. STELLA offers a dedicated tool for expressing and executing stencil computations with notable advantages. It operates as an embedded language within a general-purpose programming environment, allowing seamless integration with existing codebases. STELLA's specialization ensures concise and expressive code that closely aligns with the mathematical representation of stencil operations.

Additionally, the language is optimized for structured grids, focusing on their regularity to enhance computational performance. STELLA is highly parallelizable, making it well-suited for modern multi-core and parallel computing architectures. By providing a high-level abstraction for stencil computations, STELLA seeks to improve code readability, maintainability, and productivity while delivering efficient performance in stencil-based simulations and computations.

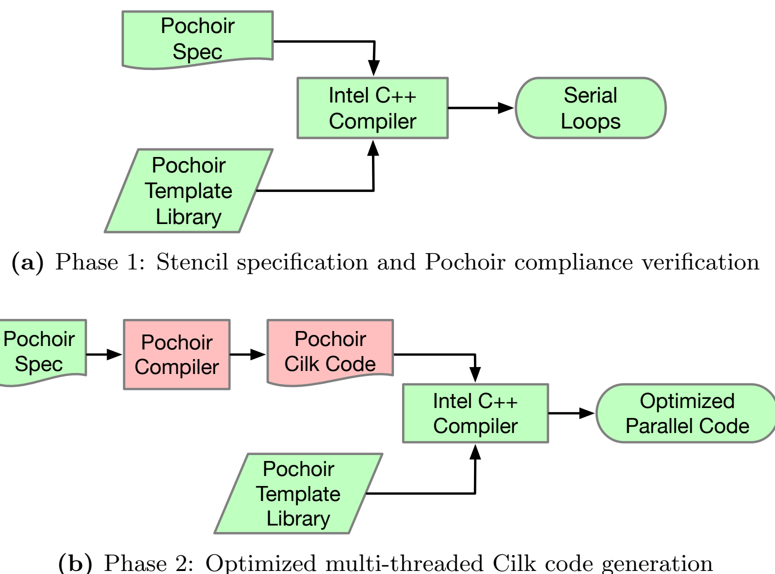
2.5.6 Halide

In the world of image processing, achieving both flexibility and high performance can be difficult. Halide [110] aims to resolve this issue by allowing developers to define image processing algorithms independently from how they are scheduled and executed on specific hardware. This approach allows programmers to focus on the core logic of their image-processing tasks, while Halide's compiler generates efficient execution plans for various

hardware platforms. This not only streamlines development but also facilitates optimizations, making it easier to adapt image processing code to diverse computing architectures while maintaining high performance.

2.5.7 Pochoir

Pochoir [3] is a compiler and runtime system for implementing stencil computations on multiple processors. A stencil refers to the value of a grid point in a d-dimensional spatial grid at time t, expressed as a function of neighboring grid points at times preceding t. In a stencil computation, the value of the grid point is updated repeatedly based on the values of neighboring points and the value of itself. Pochoir is also a stencil-based compiler that allows the developers to write their stencils using a domain-specific stencil language and then translate it into high-performing C-like code after a two-phase mechanism as shown in Figure 2.7 that has an efficient parallel cache optimizing algorithm.



(a) Phase 1: Stencil specification and Pochoir compliance verification

(b) Phase 2: Optimized multi-threaded Cilk code generation

Figure 2.7: Pochoir two-phase compilation strategy [3]

This two-phase compilation strategy aims to enhance the performance of stencil computations by utilizing the simplicity of template libraries and the optimization abilities of multi-threading, particularly with the Cilk programming language.

Phase 1 During stencil computations, a template library is utilized to simplify expressions. The code is compiled with the Intel C++ compiler, ensuring compliance with the Pochoir framework as shown in Figure 2.7a.

Phase 2 Stencil code is then transformed into efficient Cilk code, incorporating parallel constructs like *spawn* and *sync*. Designed for multi-threading, Cilk optimizes load balancing and data locality for effective execution on multi-core processors. The code is compiled using the Intel Cilk Plus compiler or similar tools as detailed in Figure 2.7b.

In summary, the Pochoir framework uses a two-phase compilation strategy which simplifies stencil specification by using a template library and improves speed through multi-threading via the Cilk programming language. This approach ensures that the Pochoir criteria are met in the first phase and produces efficient and parallelized code in the second phase, resulting in outstanding stencil computation performance.

2.5.8 Patus

Patus [111] is a software framework that simplifies the development of high-performance stencil computations, particularly in the context of earthquake simulations. Patus offers an easy-to-use environment that streamlines the creation of stencil-based code, while still maintaining strong performance capabilities. Its ability to automate optimizations and parallelism makes it ideal for modern multi-core and parallel computing architectures. Evaluation within earthquake simulations has demonstrated Patus' usefulness in real-world scientific contexts, showing that it can expedite and improve the accuracy of earthquake modeling and related computational tasks.

2.5.9 PLuTo

Many scientific engineering applications spend their time mostly on nested loops. PLuTo [4, 67, 72, 112] is an automatic polyhedral source-to-source transformation framework that can address this issue by transforming imperfectly connected nested loops for parallelism and data locality. There are several tools chained together, as shown in Figure 2.8 to implement the PLuTo framework. The sequences of nested loops of the program are identified by using a scanner/parser. The dependence tester is used to identify the data dependencies among the loops in the nested loop sequence. After that, polyhedral transformation is applied to the nested loop sequence so that it will possess data locality optimizations.

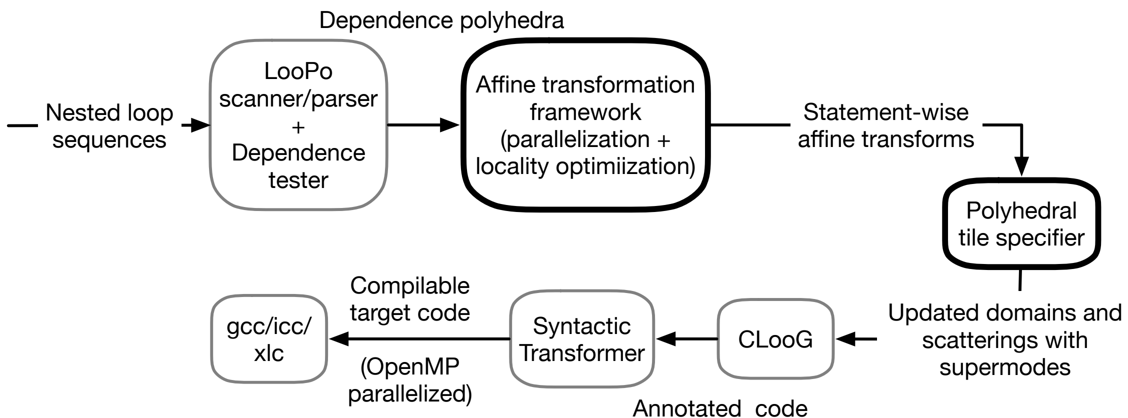


Figure 2.8: PLuTo source-to-source transformation system [4]

The PLuTo framework optimizes code for performance while preserving its readability. It determines the most effective loop transformations, such as tiling and skewing, to minimize synchronization overhead and increase parallelism. PLuTo is also adaptable to various hardware platforms, enabling developers to harness the performance benefits of different parallel computing architectures.

2.5.10 Polly

Polly [5, 113] is a powerful framework that leverages the polyhedral model to improve the data locality and loop parallelism of a program's LLVM intermediate representation (IR). LLVM is a collection of tools and technologies designed to create a compiler with a language and platform-independent intermediate representation [114, 115]. With Polly, a program's data access patterns are analyzed and the optimal memory access patterns are determined using the polyhedral model. Loop transformations, tiling, and loop fusion are some of the techniques used by Polly to enhance program performance [116].

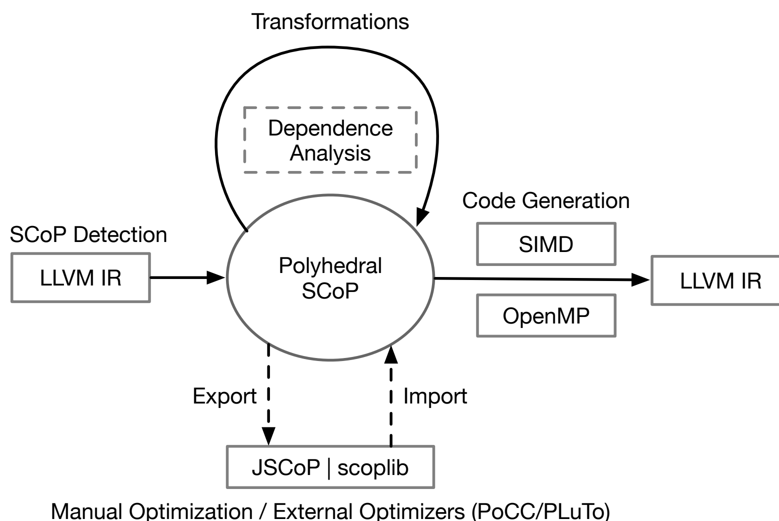


Figure 2.9: Polly architecture [5]

The Polly framework includes three main stages: front-end, middle-end, and back-end passes, which work together to optimize the program, as shown in Figure 2.9. During the front-end pass, the static control parts (SCoPs) of the program are identified and converted to the polyhedral model. Next, the middle-end pass locates data locality enhancements by analyzing the polyhedral model. These optimizations can be done through the codes integrated into the Polly framework or through external optimizers such as PoCC/PLuTo [4] using the interface provided by Polly. It is important to consider which option is best for the specific project and needs. The Polly framework is also capable of identifying the loops that can be represented using OpenMP parallel loops and SIMD instructions [113]. Finally, the back-end pass regenerates the optimized code with these enhancements.

2.5.11 Summary

Here, we will summarize the discussed DSLs and frameworks, highlighting their limitations, which led us to choose OP2 DSL for communication-avoidance enhancements.

Table 2.1: Summary of DSLs/Frameworks

DSL/ Framework	Category	Description	Limitations
Firedrake	Structured/ Unstructured	A finite element library for Python designed to solve PDEs in both structured and unstructured scenarios. Utilizes MPI for parallelism and PETSc for linear algebra. Major applications include Finite Element Analysis, with a focus on problems such as heat conduction and fluid dynamics.	Requires UFL (Unified Form Language) for expressing variational forms, which may be considered a limitation for users unfamiliar with UFL. Limited support for highly complex geometries.
Devito	Structured/ Unstructured	A high-level finite difference DSL for Python, suitable for structured and unstructured computations. Employs SymPy for symbolic mathematics, NumPy for numerical operations, and OpenMP for parallelization. Major applications include Numerical Simulations, particularly in seismic wave propagation studies.	Limited support for highly complex equations. Some complex expressions may not be optimally translated to low-level code. Performance may degrade for problems with irregular geometries.
TiDA	Structured	A C++ library for time domain analysis on structured grids, using OpenMP and MPI for parallelism. Major applications include Time Domain Analysis on Structured Grids, particularly for electromagnetic wave propagation studies.	Limited to time domain analysis on structured grids. Does not extend to frequency domain analysis. Limited support for hybrid parallelism, which may affect scalability on some architectures.
STELLA	Structured	A domain-specific embedded language designed for efficient implementation of stencil codes on structured grids. Highly parallelizable and optimized for computational performance on structured grids. Major applications include Stencil Code Implementation, with a focus on environmental modeling and climate simulations.	Relatively steeper learning curve due to its domain-specific nature.

Halide	Structured/ Unstructured	A C++ DSL for image processing designed for structured and unstructured image data. Primarily tailored for efficient image processing pipelines. Major applications include Real-time Image Processing in computer vision and photography.	Limited support for non-image processing tasks. Not suitable for general-purpose DSL use beyond image processing. Limited optimization for non-regular data access patterns.
Pochoir	Structured	A C++ DSL for stencil computations on structured grids, optimized for efficient stencil-based algorithms. Major applications include Stencil Computations, particularly in computational fluid dynamics simulations.	Limited to stencil computations on structured grids. May not be suitable for general-purpose DSL use. Limited support for dynamic adaptivity, restricting its application to static problems.
Patus	Structured/ Unstructured	A performance portable DSL for stencil computations in C++, OpenMP, and CUDA, aiming for portability across different hardware platforms. Commonly used for earthquake simulation and other Performance Portable Stencil Computations in seismology and geophysics.	Limited to stencil computations; potential portability challenges. May require careful tuning for optimal performance on diverse architectures. Lack of advanced debugging tools.
PLuTo	Structured	An automatic parallelization tool in C, based on the Polyhedral model, attempting to parallelize loop nests for performance improvement. Major applications include Automatic Parallelization for numerical simulations in scientific computing.	Limited support for irregular loop nests. Effectiveness highly depends on the regularity of loop structures. May produce sub-optimal code for certain algorithms.
Polly	Structured	A C++ polyhedral optimization framework for LLVM, focusing on optimizing structured computations through loop transformations. Major applications include Polyhedral Optimization in optimizing linear algebra computations.	Limited to polyhedral optimization in LLVM. May not provide significant benefits for unstructured computations. Limited support for complex loop structures, impacting the effectiveness of transformations.

2.6 OP2 DSL

OP2 [6] is an active library that allows users to easily translate their programs into a format that is suitable for the running platform and libraries to be linked (e.g., MPI, OpenMP, CUDA, SYCL, OpenCL, and OpenACC). It provides an abstraction framework for the parallel execution of unstructured-mesh applications, which is an extension of the OPlus [117] (The Oxford Parallel Library for Structured Solvers) library. As the field of high-performance computing evolves rapidly, developers face challenges such as performance portability, maintaining codes, and writing future-proof code. Fortunately, OP2 decouples the scientific code from platform-specific enhancements, allowing us to address these challenges with modifications to the library and achieve optimal solutions for new and emerging technologies.

The OP2 domain-specific language (DSL) [6] provides a framework to solve unstructured-mesh-based problems, representing them as a set of parallel loops with problem-specific *computational kernels*. The relevant unstructured-mesh topology is represented in OP2 by the sets of nodes, edges, and cells and their connectivity information using unique data structures. Further, general data structures such as arrays and vectors manage the data attached to these mesh elements.

All these mentioned essential elements and processes of the OP2 application execution process enable a successful run of an application. In this section, we will look into the basic concepts of OP2 in detail within the scope of the research.

2.6.1 Sets and Maps

OP2 describes the topology of a mesh (see Section 2.5.1 for mesh types) with abstract entities, sets, and mappings or the connection between set entities. Edges, cells, and nodes are the primary set elements that we see in applications. However, there may be derivations of these main types, such as periodic edges, boundary edges, boundary cells, and so on. To build the mesh inside the program, we need the information on how these mesh elements are connected with each other. Mappings represent the connection between a *source* set and a *target* set, with a fixed cardinality. For example, an *edges-to-nodes* mapping will have information on the two sets of nodes that create edges in the mesh, as shown in Listing 2.2.

```
1 op_set nodes = op_decl_set(nnode, "nodes");
2 op_set edges = op_decl_set(nedge, "edges");
3
4 op_map e2n = op_decl_map(edges, nodes, 2, en, "e2n");
```

Listing 2.2: OP2 sets and maps

2.6.2 Data

Three main types of data are used to perform calculations in an OP2 application, as shown in Listing 2.3. `op_dat` is the abstraction given by the OP2 framework to store an array/vector of data defined on the set elements of the mesh. These *dynamic* datasets in an application are modified and exchanged among the processes during the computations. We have *global* data, which is not associated with any of the sets and is commonly accessible and modifiable by the computing kernels when passed into them by the application. *Const* data is also not associated with any of the sets but read-only and is globally visible to the computing kernels where they are accessed by referring to the given unique name.

```
1  // dynamic datasets
2  op_dat d_node_val = op_decl_dat(nodes, 2, "double", nval, "nval");
3
4  // constants
5  double gam = 1.4f;
6  op_decl_const2("gam", 1, "double", &gam);
7
8  // global data, prepare it as an argument to send to a computing kernel
9  double rms = 0.0;
10 op_arg d_rms = op_arg_gbl(&rms, 1, "double", OP_INC);
```

Listing 2.3: OP2 data

2.6.3 Parallel Loops

All the computations in an OP2 application are performed inside *parallel loops* by mapping the computations over an *iteration set*. The complexities of the code, including platform-specific and platform-independent enhancements, data exchange between processes or between the host and the device, and parallel code execution, are hidden within these parallel loops. The computing kernel accessed inside the parallel loop is executed over a *set* of mesh elements such as edges, cells, and nodes. Sometimes, the iteration set used for computations may be limited to a subset of the main sets, such as *boundary edges*.

The abstraction provided in the OP2 API for the parallel loops is `op_par_loop` as shown in Listing 2.4. In the first three arguments, `op_par_loop` takes the function pointer of the computing kernel, its name, and the iteration set, respectively. The datasets used inside the computing kernel for the calculations are sent with the relevant dataset pointer, data element location inside an n-dimensional piece of data, required mapping to access data elements indirectly while performing iterations over the iteration set or `OP_ID` (identity mapping) for direct access of data, the data type of the dataset, and the *access descriptor*, packed in the `op_arg_dat` abstraction.

The access descriptors indicate how the datasets are accessed inside the computing kernel. The main access descriptors used in the OP2 DSL are `OP_READ` (read-only), `OP_WRITE` (write-only), `OP_INC` (increment), and `OP_RW` (read-write). Based on the access mode of the arguments (`op_args`) of the `op_par_loop`, the loops are categorized into two, *direct* and *indirect*.

```
void edge_kernel (double* nval1, double* nval2, double* pnval1,
 double* pnval2) {
  *nval1 += *pnval1 - *pnval2;
  *nval2 += *pnval2 - *pnval1;
}

// loop over edges, updating nodes: update node values
op_par_loop(edge_kernel, "edge_kernel", edges,
   op_arg_dat(dnval,  0, e2n, 2, "double", OP_INC),
   op_arg_dat(dnval,  1, e2n, 2, "double", OP_INC),
   op_arg_dat(dpnval, 0, e2n, 2, "double", OP_READ),
   op_arg_dat(dpnval, 1, e2n, 2, "double", OP_READ));
```

Listing 2.4: Parallel loop written in OP2 API

Direct Loops

If all the op_args in an op_par_loop are directly accessed inside the computing kernel, such parallel loops are called *direct loops*. The iterations happen only up to the local set size and no data exchange is required before executing the loop. The direct loop example shown in Listing 2.5 shows how the dataset elements are accessed using an offset inside the computing kernel with no mappings.

```
void node_kernel (double* nval, double* pnval) {
  for (int i = 0; i < 2; i++) {
    nval[i] += pnval[i];
  }
}

// loop over nodes, updating nodes: update node values
op_par_loop(node_kernel, "node_kernel", nodes,
  op_arg_dat(dnval,  -1, OP_ID, 2, "double", OP_INC),
  op_arg_dat(dpnval, -1, OP_ID, 2, "double", OP_READ));
```

Listing 2.5: Direct parallel loop written in OP2 API

Indirect Loops

If any of the op_args in an op_par_loop are accessed indirectly through a mapping within the computing kernel, the parallel loops are classified as *indirect loops*. If the loop involves indirect reading of any of the datasets, data exchange is required before executing the loop. The iterations are then performed, including any imported data elements from other processes. The loop example shown in Listing 2.6 is an indirect loop, where op_args are accessed both directly and indirectly.

```
1  void edge_kernel (double* nval1, double* nval2, double* eval) {
2    *nval1 += *nval2 - *eval;
3    *nval2 += *nval1 - *eval;
4  }
5
6  // loop over edges, updating nodes: update node values
7  op_par_loop(edge_kernel, "edge_kernel", edges,
8     op_arg_dat(dnval,  0, e2n,   2, "double", OP_INC),
9     op_arg_dat(dnval,  1, e2n,   2, "double", OP_INC),
10    op_arg_dat(deval, -1, OP_ID, 2, "double", OP_READ));
```

Listing 2.6: Indirect parallel loop written in OP2 API

2.6.4 Data Layout

The OP2 library supports datasets consisting of basic data types with multiple dimensions by employing the `op_dat` data structure. The data elements of the dataset, regardless of dimension, are stored in a single dimension, following the row-major order. The size of the data storing array is determined by multiplying the dataset's dimension, data type, and the size of the set on which the dataset is defined.

Direct Data Access

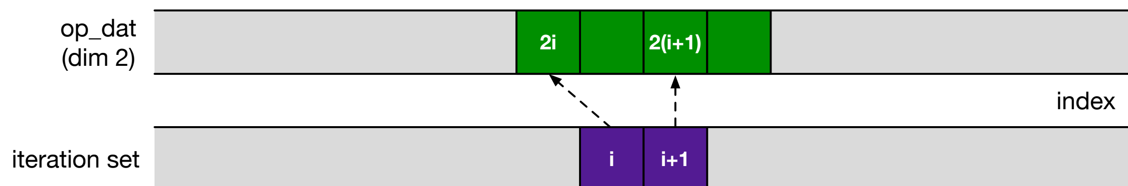


Figure 2.10: OP2 data layout for direct data access

During execution, the kernel is called several times depending on the iteration set's size and its halos. When accessing a dataset directly, as illustrated in Listing 2.5, the kernel receives a pointer, as in Figure 2.10, indicating the start of the data chunk referred to by the iteration set index, i. Since the iteration and dataset sets are identical, no data indirection is involved.

Indirect Data Access

In cases where the iteration and dataset sets differ as in Listing 2.4 and Listing 2.6, indirect data accesses occur during kernel execution as illustrated in Figure 2.11. The indirection map is used to find the pointer to the start of the data chunk referred to by the iteration set index, i. The kernel receives pointers equivalent to the arity/dimension of the map to access elements of the dataset corresponding to the single set element of the iteration set. For instance, in the given example, the kernel receives four pointers for the index i and accesses four different locations in the dataset given by the map entries, `(i, 0)`, `(i, 1)`, `(i, 2)`, and `(i, 3)`.

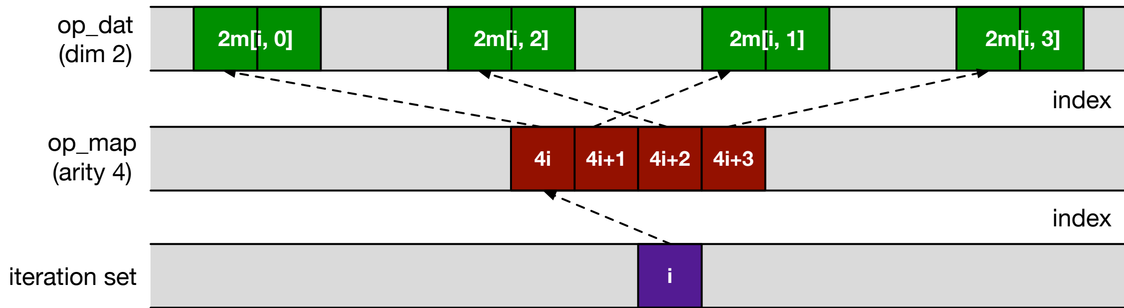


Figure 2.11: OP2 data layout for indirect data access

2.6.5 Kernels

The computations to be performed on each element of the iteration set are defined by a *kernel*. The kernel has a local perspective on these computations, regardless of the OP2 back-end type (i.e., MPI, OpenMP, CUDA, and others) linked to the application. For instance, if the application is running a multi-threaded version, the kernel has the computations to be performed by a single thread on a single call. The kernels shown in Listing 2.4, Listing 2.5, and Listing 2.6 clearly demonstrate the nature and API of a kernel in the OP2 library. Data accessed through `OP_READ`, `OP_INC`, and `OP_RW` is gathered by utilizing the pointers passed to the kernel and indirection maps before executing the kernel. Likewise, data modified through `OP_WRITE`, `OP_INC`, and `OP_RW` is scattered/written back to their memory locations using the indirection maps and pointers given to the kernel. The kernel must not violate the data access descriptors given when describing the kernel arguments using the `op_par_loop` API. Otherwise, it may lead to erroneous results due to unhandled data races and uninitialized values during kernel execution.

2.6.6 Architecture

The OP2 library provides an API for describing scientific computations related to unstructured-meshes, which is discussed in previous sections. After the code is written following the OP2 API, it is translated into a platform-specific kernel and optimized application code using a Python/CLANG LLVM-based translator [6]. The resulting code is then compiled using a traditional compiler and linked with platform-specific libraries, as explained in Figure 2.12. The resulting executable can be utilized to run the application on the designated hardware platform, accessing relevant mesh data files.

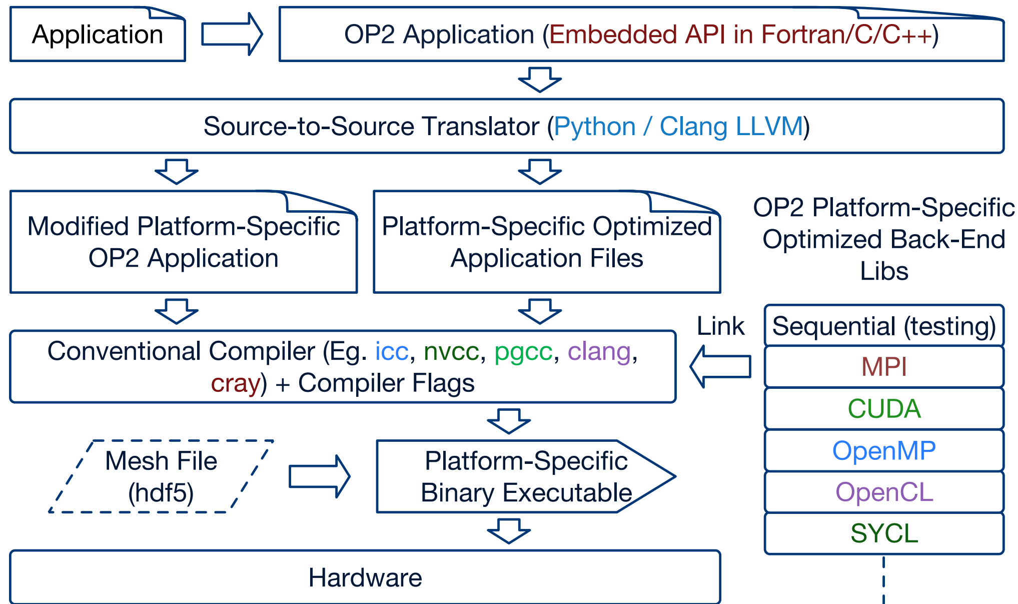


Figure 2.12: OP2 code generation

2.6.7 Supported Back-Ends

OP2 supports running the application on different multi-core and many-core architectures with suitable back-ends linked to it as specified in Figure 2.12. Even though the running platform and the linked back-end are different from each other, the user code does not need to be changed. The user is given a unified API to write the scientific code without having to consider the running platform. Our focus is on the back-ends used in our testing when comparing the communication-avoidance work. We are continuously developing and improving the supported back-ends and their features. The main supported back-ends include:

Sequential Back-End

```
1  for (int n = 0; n < set_size; n++) {
2    int map0idx;  int map1idx;
3    map0idx = arg0.map_data[n * arg0.map->dim + 0];
4    map1idx = arg0.map_data[n * arg0.map->dim + 1];
5
6    test_write_kernel(
7      &((double*)arg0.data)[5 * map0idx],
8      &((double*)arg0.data)[5 * map1idx]);
9  }
```

Listing 2.7: Sequential back-end kernel

The application runs sequentially on a single CPU core. The translator generates a code that calls the computations inside a `for` loop for each index of the iteration set as in Listing 2.7.

OpenMP Back-End

The program operates with multiple threads on an SMP (Shared-Memory Processor) CPU using OpenMP. The number of threads can be adjusted through a system environment variable. OpenMP `#pragma`s are included in the generated kernel to enable parallel execution with multiple threads, as shown in Listing 2.8. To prevent data races, a color scheme is created based on the problem size and number of threads, ensuring that adjacent indices in the iteration set are not executed by threads with the same color.

```
1  op_plan *Plan = op_plan_get_stage_upload(name, set, part_size,
2    nargs, args, minds, finds, OP_STAGE_ALL, 0);
3
4  // execute plan
5  int block_offset = 0;
6  for (int col = 0; col < Plan->ncolors; col++) {
7
8    int nblocks = Plan->ncolblk[col];
9
10   #pragma omp parallel for
11   for (int blockIdx = 0; blockIdx < nblocks; blockIdx++) {
12     int blockId  = Plan->blkmap[blockIdx + block_offset];
13     int nelem    = Plan->nelems[blockId];
14     int offset_b = Plan->offset[blockId];
15
16     for (int n = offset_b; n < offset_b + nelem; n++) {
17       int map0idx;
18       int map1idx;
19       map0idx = arg0.map_data[n * arg0.map->dim + 0];
20       map1idx = arg0.map_data[n * arg0.map->dim + 1];
21
22       test_write_kernel(
23         &((double*)arg0.data)[5 * map0idx],
24         &((double*)arg0.data)[5 * map1idx]);
25     }
26   }
27   block_offset += nblocks;
28 }
```

Listing 2.8: OpenMP back-end kernel

MPI Back-End

The problem is divided equally among the processes used by the application. Each process solves its assigned sub-problem sequentially, but all processes work in parallel. After solving the sub-problems, the final solution is generated by combining the solutions. The generated kernel can hide latency by overlapping computation and communication, and it can execute computations sequentially within a process, as shown in the example code in Listing 2.9.

```
int set_size = op_mpi_halo_exchanges(set, nargs, args);

if (set_size > 0) {
  for (int n = 0; n < set_size; n++) {
    if (n == set->core_size) {
      op_mpi_wait_all(nargs, args);
    }
    int map0idx;  int map1idx;
    map0idx = arg0.map_data[n * arg0.map->dim + 0];
    map1idx = arg0.map_data[n * arg0.map->dim + 1];

    test_write_kernel(
      &((double*)arg0.data)[5 * map0idx],
      &((double*)arg0.data)[5 * map1idx]);
  }
}

if (set_size == 0 || set_size == set->core_size) {
  op_mpi_wait_all(nargs, args);
}
```

Listing 2.9: MPI back-end kernel

CUDA Back-End

```
//set CUDA execution parameters
int nthread = OP_block_size;

for (int round = 0; round < 2; round++) {
  int start = round == 0 ? 0 : set->core_size;
  int end = round == 0 ? set->core_size : set->size + set->exec_size;
  if (end - start > 0) {
    int nblocks = (end - start - 1) / nthread + 1;

    op_cuda_test_write_kernel<<<blocks, nthread>>>(
      (double *)arg0.data_d,
      arg0.map_data_d,
      start, end, set->size + set->exec_size);
  }
}
cutilSafeCall(cudaDeviceSynchronize());
```

Listing 2.10: CUDA back-end kernel

The application will offload the computations to the GPU when linking the application to this back-end. The translator will generate a kernel as in Listing 2.10 considering the GPU block sizes.

MPI OpenMP Back-End

This is a combination of MPI and OpenMP where the problem is divided into small sub-problems, assigned to individual processes, and inside a process, the sub-problem is executed in parallel using multiple OpenMP threads. The application utilizes the kernel detailed in Listing 2.8 with some additions to support MPI message exchanges with latency hiding.

MPI CUDA Back-End

This is a combination of MPI and CUDA where the problem is divided into small sub-problems, assigned to individual processes, and inside a process, the sub-problem is executed in parallel using the threads in the GPU. The application utilizes the kernel detailed in Listing 2.10 with some additions to support MPI message exchanges with latency hiding.

MPI CommAvoid (CA) Back-End

This is the newest addition to the OP2 library, of which the details are explained in Chapter 4. This communication-avoidance back-end developed for the OP2 library is one of the main contributions of this research.

Chapter 3

Communication-Avoiding Optimizations on Shared-Memory Systems

It is challenging to improve the vertical data movement of current computing systems [118]. Vertical data movement refers to moving data through the system's memory hierarchy to the processing elements and vice versa. This intranodal performance enhancement has become cumbersome due to the nature of data access patterns mostly specific to individual programs. When a data element is accessed in a program, the relevant data element and its surrounding data elements are fetched from the main memory and brought to the fast cache of the system if it is not available in the cache at the time of accessing. The current data in the cache is evicted and sent back to the main memory with its updates to facilitate the incoming data. This fast cache is limited in size in modern-day computing systems and it is time and energy-consuming to move data to and from the cache. So, the data locality of the program plays a significant role in its performance and it has become a priority to improve the data locality when running a program. Various cache-replacing algorithms such as FIFO/LIFO (First-In, First-Out / Last-In, First-Out), LRU (Least Recently Used), LFU (Least Frequently Used), and MRU (Most Recently Used) algorithms have been developed and integrated into modern computing platforms to improve the data locality of the programs. However, the impact of these algorithms on enhancing the performance of numerous scientific applications remains constrained. This limitation arises from the unique and often intricate data access patterns inherent in scientific applications.

3.1 Motivation

Our research is centered on applications that use unstructured-meshes. These types of applications involve solving partial differential equations related to fluid, heat transfer,

and stress analysis using the finite element method (FEM) and finite volume method (FVM). In finite element analysis (FEA), these equations are transformed into a set of simultaneous algebraic equations and represented as data parallel loops within the application. This set of loops is then solved using variational methods in calculus by minimizing an error function after a large number of iterations.

In these unstructured-mesh-based applications, the mesh connectivity information must be given explicitly to the application through mappings such as edges-to-nodes and cells-to-nodes. These map indirections result in many indirect memory accesses when executing the application. Due to this reason, it is impossible to apply compile-time loop executing enhancements such as traditional tiling to these applications. At the same time, in the real world, the unstructured-mesh datasets used in these applications can become quite large, which in turn limits the possibility of keeping the working dataset in a level of cache of the processor until the application completes a considerable number of computations. As a result, the application experiences continuous cache misses, which force it to move data between various levels of memory very frequently, leading to a decrease in performance due to memory bandwidth limitations.

We have several such unstructured-mesh-based benchmarks and applications, Airfoil, MG-CFD, Volna, and Hydra, which use a set of data parallel loops to arrive at a final solution. Airfoil [82] is an industrial representative CFD benchmark application that uses a 2D unstructured-mesh and solves 2D Euler equations using scalar numerical dissipation. MG-CFD [83] is a *mini-application* that represents a geometric unstructured CFD code and Volna [84] is a CFD application used for the modeling of tsunami waves. Hydra [63] is our main application of interest, which Rolls Royce uses to simulate next-generation jet engine components. Improving the performance of these unstructured-mesh-based applications on a shared-memory system has become a pressing need, due to the limited performance we experience with bandwidth limitations in modern computing platforms.

Several algorithms have been developed to enhance data locality by optimizing data reuse within data-parallel loops in unstructured-mesh applications. Most of these techniques are runtime data access pattern enhancements with an *inspection/execution* scheme [7, 39]. SLOPE [37] is one such library, which is based on an *inspector/executor* scheme that targets *sparse tiling*. We now explore the SLOPE library, developed by Luporini et al. [37] embedding it to the OP2 DSL [6], to achieve performance enhancements over the current best shared-memory version of the aforementioned applications.

3.2 Concepts and Approach

Fundamental concepts necessary to understand shared-memory parallelization enhancements introduced by the SLOPE library [37] are explained in this section. These concepts include loop transformation with sparse tiling mechanisms, inspector/executor schemes, and loop-chain abstraction.

3.2.1 Sparse Tiling

In Section 2.4.1, we explained traditional tiling and loop fusion, which enhance the performance of a set of loops with improved data locality. When we combine these techniques, loop fusion and loop tiling, we call it *time tiling* [8]. In many scientific computational applications, a time iteration executes until a specified error function converges with a given accuracy. In traditional or spatial tiling, we exploit the data locality within the computation of a single time iteration. But in *time tiling*, we exploit data locality over the time iterations of these applications by generating and executing tiles over the time dimension. This *time tiling* concept was extensively studied for structured stencil codes [35, 67, 119, 120]. Time tiling is known as *sparse tiling* in the unstructured-mesh domain [39], which has little attention due to complexities arising in irregular memory accesses. The sparse tiling technique was initially developed by Douglas et al. [26] with the name of *unstructured cache-blocking*. There was also no available framework to utilize sparse tiling enhancements to unstructured-mesh applications until the SLOPE library [8, 37, 121] was developed. Before the development of the SLOPE library, these enhancements were manually applied to the applications, when required [39].

3.2.2 Full Sparse Tiling and Generalized Full Sparse Tiling

The sparse tiling method requires a sequential clean-up tile after computations, leading to limited parallel performance [26]. However, a more recent approach by Strout et al. [38, 39] partitions the unstructured-mesh, covering all iterations and eliminating the need for a sequential clean-up tile. This improved method is known as '*full* sparse tiling'.

Previously, sparse tiling techniques have been implemented manually for specific applications, such as moldyn, Gauss-Seidel, and the sparse matrix powers kernel [7]. It has been challenging to develop a generalized full sparse tiling mechanism due to potential overheads in the inspection phase. However, a possible solution was introduced by Strout et al. [7] with the generalized full sparse tiling (gFST) mechanism. This approach utilizes a newly developed loop-chain abstraction and a task-graph-based approach for achieving shared-memory parallelism. Luporini et al. [8] later created the SLOPE library [121] based on the gFST development, which uses a tile coloring algorithm instead of the task-graph approach.

3.2.3 Inspector/Executor Schemes

Structured-meshes can be efficiently represented by computer programs due to their regular grid and uniform connectivity. This means that the number of neighbors remains consistent, and the connectivity between mesh elements is clear, making it easy to access data elements defined on mesh elements such as nodes, edges, and cells. As a result, compilers and libraries can perform static performance analysis and compile-time enhancements, such as the vectorization of arrays, in structured-mesh-based applications.

However, in unstructured-meshes, the cell shapes are arbitrary and the mesh connectivity is irregular. The number of neighbors varies from mesh element to mesh element. The mesh nodes may be distributed unevenly and the represented geometry is more complex. The required memory footprint is higher since the explicit connectivity of the mesh needs to be stored. Due to these reasons, the nature of the mesh is unpredictable until it is loaded into the application. This prevents the compilers and the libraries from performing static enhancements to the unstructured-mesh-based code in an application. The need for separate phases to analyze and execute enhancements has emerged due to this. Dynamic analysis is necessary and completing these runtime enhancements requires significant time and resources.

Salz et al. [29] introduced an inspector/executor scheme to enhance data locality in unstructured-mesh applications. Further, various inspector/executor schemes suitable for specific types of applications were studied by different authors [7, 25, 38, 49].

In general, during the inspection phase, the iteration space execution order is designed and decided based on the data access patterns of datasets of the loop-chain. This may include reordering the iteration execution. The main objective of the inspection phase is to ensure that none of the enhancements affect the problem's final solution, meaning the most challenging part is preserving the program's semantics. The time and resources spent on the inspection phase depend on the complexity of the loop enhancements targetted for the loop-chain. The actual application computations are performed during the executor phase, using the iteration order generated in the inspection phase to achieve performance enhancements.

In real-world applications, the enhancements gained in the execution phase are much higher than the overheads added in the inspection phase, especially with larger datasets and a larger number of iterations. The generalized inspector/executor scheme developed by Strout et al. [7] demonstrates that the gains in the execution phase outperform the overheads in the inspection phase.

3.2.4 Loop-chain Abstraction

A loop-chain is a sequence of consecutive loops without any global synchronization points (such as global reductions) in between loops, specified or annotated with information to facilitate runtime dependency analysis. The information should be provided by the programmer or automatically derived from the code either through a code parse or a DSL's API. This information then enables us to reason about the dependencies of the sequence of the loops collectively. For loop-chains over unstructured-meshes, a *sparse tiling schedule* can be created from the analysis, providing an execution ordering of the iterations over the mesh [8]. This ordering which is semantically equivalent to the original can then be used to carry out the loop iterations.

For example, consider the two *sequential* loops detailed in Listing 3.1, over the unstructured-mesh shown in Figure 3.1. The mesh consists of nodes, edges, and cells,

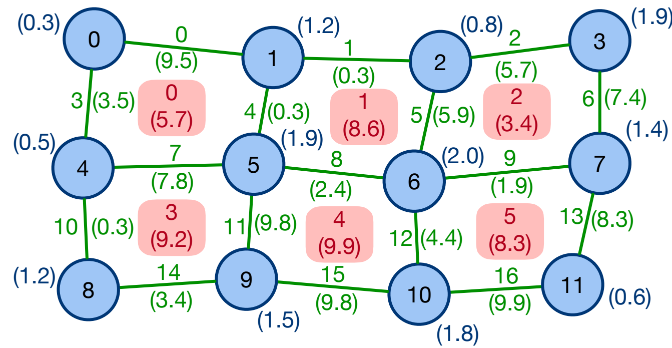


Figure 3.1: Example unstructured-mesh with nodes, edges, and quadrilateral cells (data values in parenthesis)

```
1  for (int t = 0; t < tmax; t++) { //main iteration loop
2    ...
3    // loop over edges, updating nodes: update residuals
4    for (int iter = 0; iter < nedges; iter++) {
5      int mapidx1 = en[it*2+0]; int mapidx2 = en[it*2+1];
6      res[2*mapidx1+0] += pres[2*mapidx1+0]-pres[2*mapidx1+1];
7      res[2*mapidx1+1] += pres[2*mapidx2+0]-pres[2*mapidx2+1];
8
9      res[2*mapidx2+0] += pres[2*mapidx2+1]-pres[2*mapidx2+0];
10     res[2*mapidx2+1] += pres[2*mapidx1+1]-pres[2*mapidx1+0];
11   }
12
13   // loop over edges, updating nodes: calculate edge flux
14   for (int iter = 0; iter < nedges; iter++) {
15     int mapidx1 = en[it*2+0]; int mapidx2 = en[it*2+1];
16     int mapidx3 = ec[it*2+0]; int mapidx4 = ec[it*2+1];
17
18     flux[2*mapidx1+0] += res[2*mapidx1+0]*cw[4*mapidx3+0]
19                        - res[2*mapidx1+1]*cw[4*mapidx3+1];
20
21     flux[2*mapidx1+1] += res[2*mapidx2+1]*cw[4*mapidx3+2]
22                        - res[2*mapidx1+0]*cw[4*mapidx3+3];
23
24     flux[2*mapidx2+0] += res[2*mapidx2+1]*cw[4*mapidx4+2]
25                        - res[2*mapidx1+0]*cw[4*mapidx4+3];
26
27     flux[2*mapidx2+1] += res[2*mapidx1+0]*cw[4*mapidx4+0]
28                        - res[2*mapidx1+1]*cw[4*mapidx4+1];
29   }
30   ...
31 }
```

Listing 3.1: Sequential loops in C

where a loop over edges, updating the nodes, at each end of the edge would require explicit connectivity information specified by a mapping of *edges-to-nodes*, `en`. Two such loops, occurring within a larger time-stepping iterative loop, are illustrated in lines 4-11 (`update`) and 14-29 (`edge_flux`) in Listing 3.1. Both the loops increment data held on the nodes, `res` and `flux` respectively, indirectly via the mapping array, `en`. The `edge_flux`

```
1  inline void update (double* res1, double* res2, double* pres1,
2   double* pres2) {
3    res1[0] += pres1[0]-pres1[1];   res1[1] += pres2[0]-pres2[1];
4    res2[0] += pres2[1]-pres2[0];   res2[1] += pres1[1]-pres1[0];
5  }
6
7  inline void edge_flux (double* flux1, double* flux2, double* res1,
8   double* res2, double* cw1, double* cw2) {
9    flux1[0] += res1[0]*cw1[0] - res1[1]*cw1[1];
10   flux1[1] += res2[1]*cw1[2] - res2[0]*cw1[3];
11
12   flux2[0] += res2[1]*cw2[2] - res1[1]*cw2[3];
13   flux2[1] += res1[0]*cw2[0] - res1[1]*cw2[1];
14 }
15
16 op_set nodes = op_decl_set(nnode, "nodes");
17 op_set edges = op_decl_set(nedge, "edges");
18 op_set cells = op_decl_set(ncell, "cells");
19
20 op_map e2n = op_decl_map(edges, nodes, 2, en, "e2n");
21 op_map e2c = op_decl_map(edges, cells, 2, ec, "e2c");
22
23 op_dat dres  = op_decl_dat(nodes, 2, "double", res,  "res" );
24 op_dat dpres = op_decl_dat(nodes, 2, "double", pres, "pres");
25 op_dat dcw   = op_decl_dat(cells, 4, "double", cw,   "cw"  );
26 op_dat dflux = op_decl_dat(nodes, 2, "double", flux, "flux");
27
28 for (int t = 0; t < tmax; t++) { //main iteration loop
29   ...
30   // loop over edges, updating nodes: update residuals
31   op_par_loop(update,   "update",   edges,
32     op_arg_dat(dres,  0, e2n, 2, "double", OP_INC  ),
33     op_arg_dat(dres,  1, e2n, 2, "double", OP_INC  ),
34     op_arg_dat(dpres, 0, e2n, 2, "double", OP_READ ),
35     op_arg_dat(dpres, 1, e2n, 2, "double", OP_READ ));
36
37   // loop over edges, updating nodes: calculate edge flux
38   op_par_loop(edge_flux,"edge_flux", edges,
39     op_arg_dat(dres,  0, e2n, 2, "double", OP_READ),
40     op_arg_dat(dres,  1, e2n, 2, "double", OP_READ),
41     op_arg_dat(dcw,   0, e2c, 4, "double", OP_READ),
42     op_arg_dat(dcw,   1, e2c, 4, "double", OP_READ),
43     op_arg_dat(dflux, 0, e2n, 2, "double", OP_INC ),
44     op_arg_dat(dflux, 1, e2n, 2, "double", OP_INC ));
45   ...
46 }
```

Listing 3.2: Loops written in OP2 API

loop indirectly reads data, cell weights (`cw`), held on the two cells next to an edge, via the mapping *edges-to-cells*, `ec`. The `update` and `edge_flux` loops taken consecutively can be viewed as a loop-chain with two *parallel* loops and specified using the OP2 DSL's API [6] as in Listing 3.2. According to the definition provided in [8], the loop-chain can be further elaborated as follows:

- Loop-chain $\mathbb{L} = L_0, L_1, ..., L_{n-1}$, an ordered sequence of n loops : `edge_kernel`, `cell_kernel` where $n = 2$, declared as `op_par_loops`.

- Iteration spaces $\mathbb{S} = S_0, S_1, ..., S_{m-1}$, a collection of disjoint iteration spaces representing mesh element types : `edges`, `nodes`, and `cells`, $m = 3$, declared as `op_sets`.

- Explicit connectivity between iteration spaces $\mathbb{M} = M_0, M_1, ..., M_{o-1}$, where $M : S_i \rightarrow S_j^a$ is a map with arity a : `e2n` and `c2n`, $o = 2$, declared as `op_maps`. For example, `e2n` has an arity of 2, mapping an edge to two nodes and `c2n` has an arity of 4, mapping a cell to four nodes.

- Access descriptors, one or more 2-tuples of the form of $<M,$ `mode`$>$ associated with a loop L_i where M is a map indicating indirect access or `OP_ID` (identity mapping) indicating direct access on data (specified by `op_dats` in OP2) defined on the iteration space of the loop. `mode` is the mode of data access, read (`OP_RW`), write (`OP_WRITE`), or increment (`OP_INC`). In OP2, the access descriptors are defined using `op_arg_dats` API.

The above definition provides information to carry out an inspection or analysis phase to create a set of tiles, i.e., the aforementioned sparse tiling schedule. The schedule will guarantee those data dependencies are not violated such that each tile can be executed in its entirety without any data access to/from outside the tile. Executing a tile, T_i entails executing all the iterations from L_0 belonging to that tile, followed by all the iterations in that tile for L_1 and so on up to L_{n-1}. Then the next tile, T_{i+1} is executed in a similar manner, continuing this pattern of execution until all the tiles have been completed. It is important to note that sparse tiling assumes that the order of execution of loop iterations does not affect the final result, at least within machine precision. This encompasses a large number of explicit numerical schemes, particularly when solving PDEs in numerical simulation applications.

3.2.5 OP2 Shared-Memory Parallelism

In this chapter, we are comparing the OP2 shared-memory parallelized version of an application with the SLOPE version. Therefore, it is crucial to identify the current OP2 shared-memory parallelization support when comparing results.

In OP2, shared-memory parallelized execution is based on the principle that the order in which iterations are executed should not affect the final outcome of the program. This is achieved by assigning different sets of iterations to different threads in a manner that prevents any race conditions from occurring. The OP2 library supports application parallelization through iteration space partitioning and prioritizes these iterations through coloring, executing the same colored iterations in parallel through OpenMP thread parallelization.

```
1  void edge_kernel_01 (double* nval1, double* nval2, double* pnval1,
2   double* pnval2) {
3    *nval1 += *pnval1 - *pnval2;
4    *nval2 += *pnval2 - *pnval1;
5  }
6  ...
7  for (int t = 0; t < T; t++) {
8    // L0: loop over edges, updating nodes: read node values
9    op_par_loop(edge_kernel_01, "edge_kernel_01", edges,
10     op_arg_dat(nodeval,      0, e2n, 2, "double", OP_INC),
11     op_arg_dat(nodeval,      1, e2n, 2, "double", OP_INC),
12     op_arg_dat(prev_nodeval, 0, e2n, 2, "double", OP_READ),
13     op_arg_dat(prev_nodeval, 1, e2n, 2, "double", OP_READ));
14
15   // L1: loop over cells, update nodes: read cell values
16   op_par_loop(cell_kernel, "cell_kernel", cells,
17     op_arg_dat(nodeval,  0, c2n,   4, "double", OP_INC),
18     op_arg_dat(nodeval,  1, c2n,   4, "double", OP_INC),
19     op_arg_dat(cellval, -1, OP_ID, 1, "double", OP_READ));
20
21   // L2: loop over edges, updating nodes: read edge values
22   op_par_loop(edge_kernel_02, "edge_kernel_02", edges,
23     op_arg_dat(nodeval,  0, e2n,   2, "double", OP_INC),
24     op_arg_dat(nodeval,  1, e2n,   2, "double", OP_INC),
25     op_arg_dat(edgeval, -1, OP_ID, 1, "double", OP_READ));
26 }
```

Listing 3.3: Section of an OP2 program to explain full sparse tiling with the SLOPE library. Example inspired by Strout et al. [7].

In the code example Listing 3.3, the first loop, which is the `edge_kernel_01`, iterates over the *edges* and `nodeval` is incremented indirectly using `e2n` mappings. The OP2 API is responsible for partitioning the *edge* iteration set and assigning colors in such a way that no adjacent partitions, consisting of edges that update the same node/nodes, have the same color. OP2 ensures that partitions of the same color are executed in parallel by different threads, while partitions of different colors are executed serially in an order arranged according to the data dependencies. The assigned thread executes elements in a partition serially. To achieve better data locality, OP2 utilizes third-party partitioning libraries such as Scotch [122] and METIS [123].

3.3 SLOPE Library

The SLOPE library, developed by Luporini et al. [8], is an open-source tool that enables the creation of loop-chains and supports sparse tiling through an inspector/executor scheme for unstructured-mesh applications.

Here, we utilize the loop-chain abstraction demonstrated in Section 3.2.4 in explaining the concepts of the SLOPE library. To illustrate the full sparse tiling process in the upcoming sections, we refer to the example code provided in Listing 3.3. In this

example, the first loop updates the `nodeval` dataset by iterating over the mesh edges and reading the `prev_nodeval` dataset, utilizing the *edges-to-nodes*, `e2n` mapping to access the datasets indirectly. The second loop updates the `nodeval` dataset by iterating over the mesh cells and reading the `cellval` dataset, utilizing the *cells-to-nodes*, `c2n` mapping to access the `nodeval` dataset indirectly. In the third loop, the `nodeval` dataset is updated while reading the `edgeval` dataset, with mapping indirections similar to the first loop.

3.3.1 Inspection Phase

```
1  int avg_tile_size = 5000;  int seed_loop = 1;
2
3  // sets and maps
4  set_t* sl_nodes = set("nodes", nnode);
5  set_t* sl_edges = set("edges", nedge);
6  set_t* sl_cells = set("cells", ncell);
7
8  map_t* sl_pedge = map("e2n", sl_edges, sl_nodes, edge, nedge*2);
9  map_t* sl_pcell = map("c2n", sl_cells, sl_nodes, cell, ncell*4);
10
11 // descriptors
12 desc_list edge01_desc({desc(sl_pedge, INC),
13                         desc(sl_pedge, READ)});
14
15 desc_list cell_desc({desc(sl_pcell, READ),
16                       desc(DIRECT, WRITE)});
17
18 desc_list edge02_desc({desc(sl_pedge, INC),
19                         desc(DIRECT, READ)});
20
21 // inspector initialization
22 map_list mesh_maps({sl_pedge, sl_pcell});
23 inspector_t* insp = insp_init(avg_tile_size, OMP, COL_DEFAULT, &mesh_maps);
24
25 // give the loop-chain arrangement
26 insp_add_parloop(insp, "edge_kernel_01", sl_edges, &edge01_desc);
27 insp_add_parloop(insp, "cell_kernel"   , sl_cells, &cell_desc);
28 insp_add_parloop(insp, "edge_kernel_02", sl_edges, &edge02_desc);
29
30 insp_run(insp, seed_loop);    // run the inspector
```

Listing 3.4: SLOPE inspector API. Generated for the OP2 loop-chain in Listing 3.3.

First, the information of the loops such as datasets and their data access patterns is given to the SLOPE library as illustrated in Listing 3.4 to run the inspector and generate the tiling schedule. In addition to the tiling schedule generation, the library provides an iteration assignment summary for each tile, depending on the log level. It also facilitates the generation of VTK files, which are used to visualize the tiling and coloring pattern of the mesh. We made use of this feature to generate figures (Figure 3.11a, Figure 3.11b, Figure 3.13a, Figure 3.13b, Figure 3.20a, and Figure 3.20b) that illustrate the partitioning and coloring of the meshes we tested.

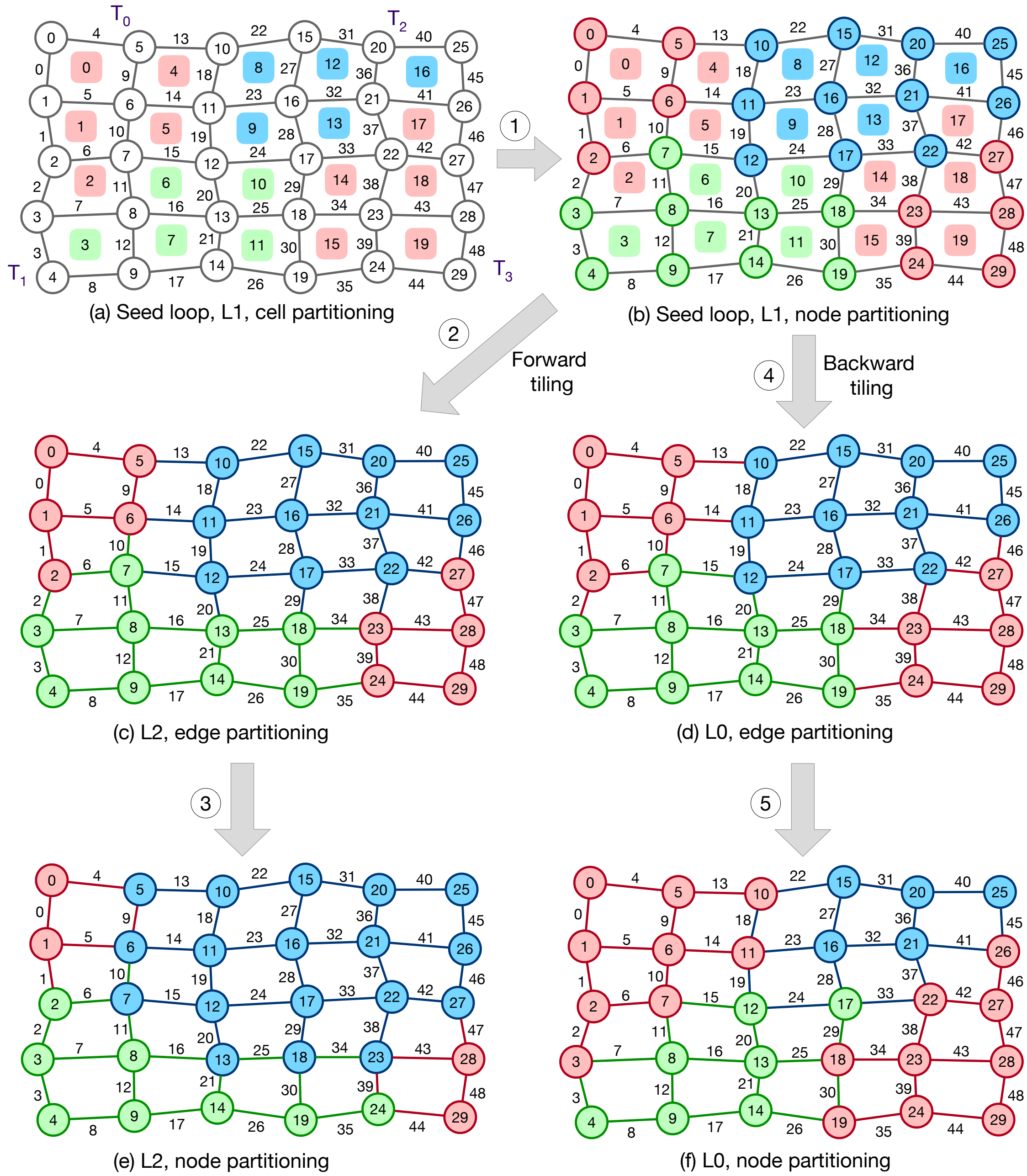


Figure 3.2: Mesh partitioning and coloring in inspection phase for the OP2 loop-chain in Listing 3.3

To begin, we need to find a *seed loop* for our loop-chain. *Seed loop* [8, 37] is the loop that initializes the tiles for the loop-chain, and it must fully represent the mesh. In this case, we have chosen loop L_1, which iterates over the cells, as our seed loop. The inspector then partitions the iteration space of the seed loop into initial partitions, where each partition (P_i - i^{th} partition) represents a tile (T_i - i^{th} tile). For example, the mesh shown in Figure 3.2a has been partitioned into four tiles, which are then colored using the minimum number of colors so that no adjacent tiles have the same color.

Once the tiles are colored, we assign an iteration order or execution priority for the seed loop tiles, with the priority determined by the assigned color. The lower the value of the assigned color, the higher the execution priority of the tile. The color of the ith tile will be represented as T_i^c. In this example, we assigned the colors T_0^R, T_1^G, T_2^B, and T_3^R to the tiles. Before assigning iterations of other loops to the initial tiles, we consider the data dependencies of the datasets.

Since L_1 updates the nodes of the mesh, from nodes ($\mathbb{N}$) to tiles ($\mathbb{T}$), a *projection* ($\phi : \mathbb{N} \rightarrow \mathbb{T}$) should be calculated before tiling the other loops' iteration spaces. For example, node 7 (n_7), which has four cells ($\mathbb{C}$) connected to it, can only be valid for reading after the lowest priority tile updates it. We have established the execution priority values as $R < G$, which can be achieved by applying the MAX function to the connected mesh element colors. Hence, it can be established that node, n_7 belongs to the tile, T_1^G. This assignment of nodes of the seed loop, L_1 to tiles is shown in Figure 3.2b.

We have two other loops, L_0 and L_2, in the loop-chain. Since we have chosen L_1 as the seed loop that is in the middle of the loop-chain, we have to perform *forward tiling* for L_2 and *backward tiling* for L_0. To create a projection, ϕ from the seed loop to L_2, which iterates over the edges ($\mathbb{E}$), we have to use the MAX function for the projection. For instance, the edge 15 (e_{15}), which is bounded by two nodes belonging to the Green and Blue tiles, needs to be assigned to the Blue tile considering its priority, $\texttt{MAX}(\phi(n_7),\ \phi(n_{12})) = \texttt{MAX}(G,\ B) = 3 = B$. This shows that e_{15} must be assigned to T_2^B. Otherwise, reading of e_{15} value may occur before the updates from the Blue tile to the node (if it were assigned to the Red tile), which will be erroneous. This edge partitioning of L_2 is detailed in Figure 3.2c. Nodes of L_2 should also be assigned using another projection similar to what we performed for L_1, which is elaborated in Figure 3.2e.

When tiling L_0, to create a projection, ϕ from the seed loop, L_1 to L_0, which also iterates over the edges, we have to use the MIN function for the projection, since we are using *backward tiling* for that. For instance, the edge 14 (e_{14}), which is bounded by two nodes belonging to the Red and the Blue tiles, needs to be assigned to the Red tile considering its priority since reading of e_{14} value should occur after the updates from the Red tile to the node, $\texttt{MIN}(\phi(n_6),\ \phi(n_{11})) = \texttt{MIN}(R,\ B) = R = 1$. This shows that e_{14} must be assigned to T_0^R. Nodes of L_0 should also be assigned using another projection similar to what we performed for L_1, using the MIN function. These two stages of partitioning L_0 edges and nodes are illustrated in Figure 3.2d and Figure 3.2f, respectively.

Conflicting Colors

The example in Figure 3.3 shows mesh partitioning and coloring of the same mesh illustrated in Figure 3.2 for the loop-chain given in Listing 3.3, but starting with a different seed loop partitioning as in Figure 3.3a. The stages illustrated in Figure 3.3b, Figure 3.3d, Figure 3.3f, and Figure 3.3c, Figure 3.3e are the outcomes of the same forward and backward tiling and consequent node partitioning process explained previously for Figure 3.2.

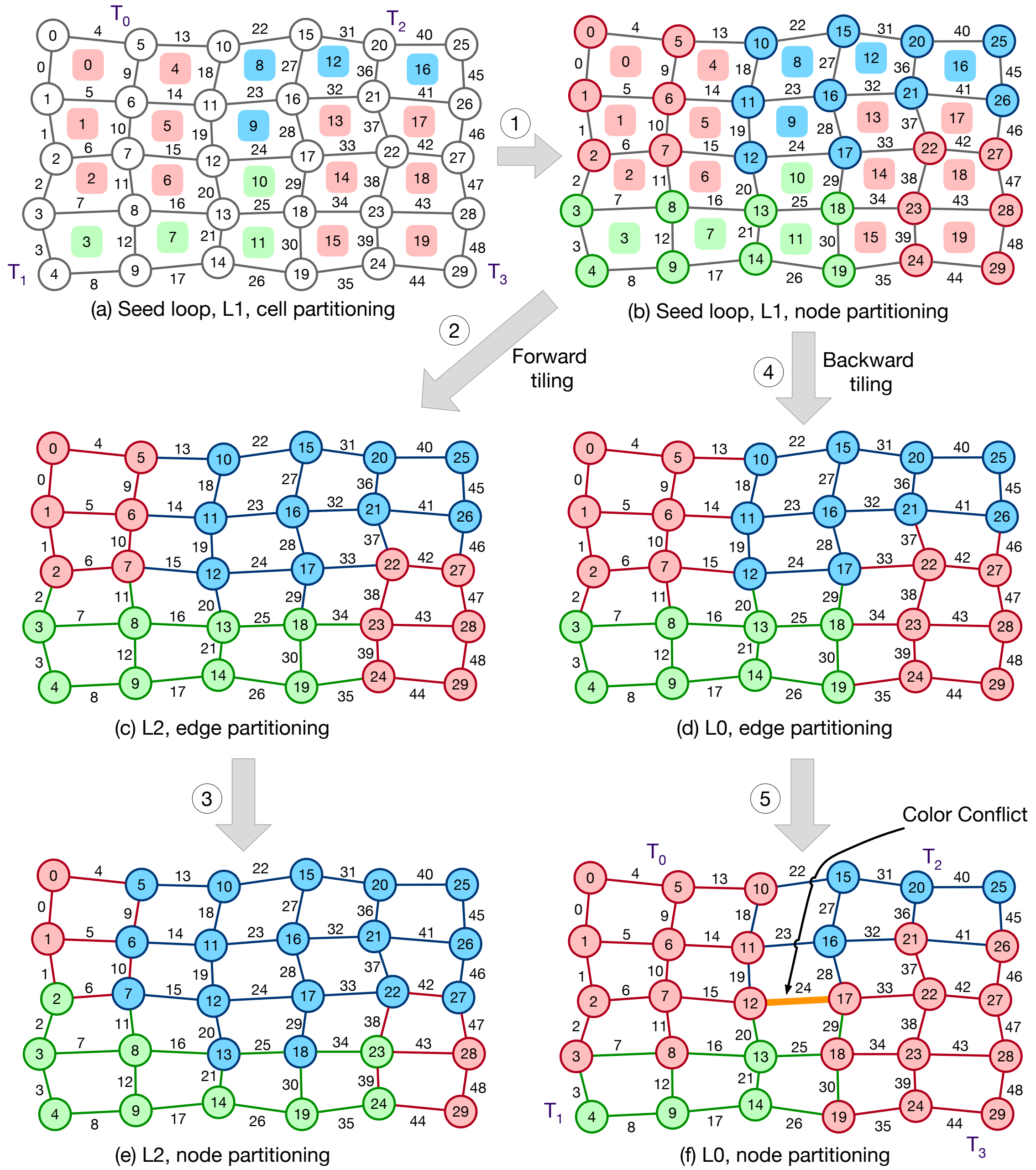


Figure 3.3: Conflicting colors

However, after the node partitioning of L_0 in Figure 3.3f, a color conflict occurs. This means that two adjacent tiles (T_0 and T_3) are assigned the same color, potentially leading to data races during execution. To address this issue, the SLOPE library adds a *fake connection* between the two conflicting tiles, which then restarts the coloring stage and the entire process [8, 37]. This repetition of the process will continue until the inspection phase is completed without any color conflicts.

The detailed algorithms for the inspection, forward and backward tiling, and projection are explained in [8, 37].

3.3.2 Execution Phase

Once the inspection phase is finished, the tiling schedule is passed to the execution phase. The shared-memory execution algorithm described in Algorithm 3.1 runs the iterations corresponding to tiles of the same color in parallel. The colors are chosen sequentially based on their priority assigned during the initial partitioning phase of the seed loop.

Algorithm 3.1: Loop-chain execution with shared-memory, SLOPE [8]

Input: A set of tiles ($\mathbb{T}$)
Result: Execute loop-chain

```
1 foreach color do
2     foreach tile T ∈ 𝕋 && T.color == color do
3         foreach iteration I ∈ T do
4             execute_iteration(I);
5         end foreach
6     end foreach
7 end foreach
```

3.3.3 OP2-SLOPE Integration

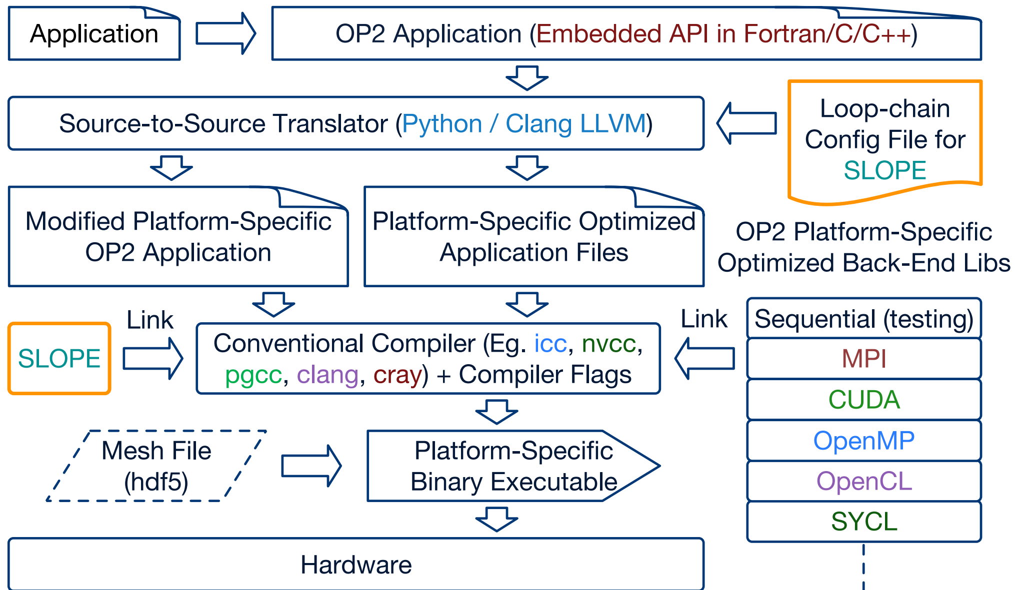


Figure 3.4: OP2 code generation with SLOPE

After recognizing the loop-chain improvements offered by the SLOPE library and constructing a framework to facilitate Fortran-based applications, we seamlessly incorporated the SLOPE library into the OP2 DSL. Consequently, the architecture of the OP2 library in Section 2.6.6 has been adapted to include the SLOPE library, as depicted in Figure 3.4. We use a configuration file that specifies the loop names and loop count of the loop-chain in the application. The OP2 code generator is then modified to automatically

apply SLOPE library enhancements to the loop-chains specified in the configuration file. We can directly generate an application binary linked with SLOPE using the OP2 DSL itself.

3.4 Performance

We now investigate the performance of the SLOPE library with shared-memory parallelism, applying it to four different existing applications developed with the OP2 library: (i) **Airfoil**, an industrial representative benchmark application, (ii) **MG-CFD**, a representative CFD mini-application, (iii) **Volna**, a tsunami wave simulation application, and (iv) **Hydra**, large-scale production CFD application used at Rolls Royce plc.

We test the performance of the SLOPE library on three platforms: **Scyrus**, **Telos**, and **ARCHER2**. Scyrus is a single computing node consisting of two Intel Xeon Silver 4116 CPU @ 2.10GHz processors, each with 12 cores (24 cores in total) arranged in 2×NUMA regions per node (12 cores per NUMA region). Telos is also a single computing node consisting of two Intel Xeon Gold 6252 CPU @ 2.10GHz processors, each with 24 cores (48 cores in total) arranged in a 2×NUMA regions per node (24 cores per NUMA region). These systems have 120GB and 384GB of memory, respectively, and a cache hierarchy of 32KB each for L1 instruction and data caches, 1MB for L2 cache, and 16MB for L3 cache, which is shared. The Intel icc compiler 2018 was used to compile application codes for both these systems with the compiler flags, `-O3 -fPIC -xHost`. ARCHER2, on the other hand, is a supercomputer with each node consisting of two AMD EPYC 7742 processors, each with 64 cores (128 total cores) arranged in 8×NUMA regions per node (16 cores per NUMA region) configuration. Each node is equipped with 256 GB of memory, and the nodes are connected by an HPE Cray Slingshot, 2×100 Gb/s bi-directional per node network. The GNU compiler collection version 10.2.0 was used on ARCHER2 with the compiler flags, `-O2 -eF -fPIC`.

During our testing, we compare the SLOPE version of the application with the standard OP2 OpenMP (*omp*) version on all three platforms mentioned in Table 3.1. We ensure that threads are bound to cores by setting environment variables, `export OMP_PLACES=cores` and `export OMP_PROC_BIND=close`. To identify the impact of NUMA (Non-Uniform Memory Access) on the performance of the applications, we conduct both single socket and dual socket runs on these platforms. For single socket runs, we use 12 threads on Scyrus, 24 threads on Telos, and 64 threads on ARCHER2. For dual socket runs, we use 24 threads on Scyrus, 48 threads on Telos, and 128 threads on ARCHER2. We experiment with various tile sizes and identify the rough range that shows potential performance benefits for a given system. We then vary the tile size with suitable intervals to capture the best performance of the sparse tiled version (i.e., the SLOPE version). Based on our empirical observations, we noticed that smaller tile sizes resulted in better performance gains across the three testing platforms. Therefore, we conduct tests by changing the tile size with smaller intervals for tile sizes below 10000 to determine the

Table 3.1: Systems specifications

System	**Scyrus** [17] Intel Skylake	**Telos** [17] Intel Cascade Lake	**ARCHER2** [125] HPE Cray EX
Processor	Intel(R) Xeon(R) Silver 4116 @ 2.10 GHz	Intel(R) Xeon(R) Gold 6252 @ 2.10 GHz	AMD EPYC 7742 @ 2.25 GHz
(procs×cores) /node	2×12	2×24	2×64
Mem/node	120 GB	384 GB	256 GB
Cache hierarchy /proc(L1/L2/L3)	32KB/1MB/16MB L3 shared	32KB/1MB/16MB L3 shared	32KB/512KB/16MB L3 shared
Interconnect	Single Node	Single Node	HPE Cray Slingshot 2×100 Gb/s bi-directional/node
OS	Debian 4.9.210-1	Debian 4.19.208-1	HPE Cray LE (SLES 15)
Compilers	Intel icc 2018	Intel icc 2018	GNU 10.2.0
Flags	`-O3 -fPIC -xHost`	`-O3 -fPIC -xHost`	`-O2 -eF -fPIC`
Memory BW Per Socket	115.212 GB/s	140.8 GB/s	204.763 GB/s

optimal runtime for the SLOPE version. For these tests, we designed an experimental setup where we vary the tile size in increments of 10 up to tile size 100, by 100 up to tile size 1000, by 1000 up to tile size 10000, and by 10000 up to tile size 200000. While presenting the results in graphs, we have selected the tile size region that best demonstrates the runtime variation around the optimal performing tile size. In these tests, we keep the `OP_PART_SIZE`, a tuning parameter in the OP2 library, as 128, which is the default value. We only compare the execution time and subtract the setup or plan time from the total execution time. We use two partitioners to partition the mesh, Chunk [8] and METIS [124], but we present the results with the Chunk partitioner since it gives the best performance during testing. When presenting the results, we take the minimum runtime of at least 5 runs for each test scenario. Runtimes of all the tests performed in this chapter are given in Section B.1 of Appendix B.

3.4.1 Airfoil

Airfoil [82] is an industrial representative CFD benchmark application. This application uses a 2D unstructured grid and solves 2D Euler equations using scalar numerical dissipation. This benchmarking application is used in testing the OP2 library for various platform-specific implementations, code generated with various programming models such as OpenMP, MPI, CUDA, and OpenACC.

```
for (int i = 0; i < ncolors; i++) {
  // for all tiles of this color
  const int ntiles_per_color = exec_tiles_per_color(exec, i);

  #pragma omp parallel for
  for (int j = 0; j < ntiles_per_color; j++) {
    // execute the tile
    tile_t* tile = exec_tile_at(exec, i, j);
    int loop_size;

    // loop adt_calc
    iterations_list& lc2n_0 = tile_get_local_map(tile, 0, "c2n");
    iterations_list& iterations_0 = tile_get_iterations(tile, 0);
    loop_size = tile_loop_size(tile, 0);
    for (int k = 0; k < loop_size; k++) {
      adt_calc(...);
    }

    // loop res_calc
    iterations_list& le2n_1 = tile_get_local_map(tile, 1, "e2n");
    iterations_list& le2c_1 = tile_get_local_map(tile, 1, "e2c");
    iterations_list& iterations_1 = tile_get_iterations(tile, 1);
    loop_size = tile_loop_size(tile, 1);
    for (int k = 0; k < loop_size; k++) {
      res_calc(...);
    }

    // loop bres_calc
    iterations_list& lbe2n_2 = tile_get_local_map(tile, 2, "be2n");
    iterations_list& lbe2c_2 = tile_get_local_map(tile, 2, "be2c");
    iterations_list& iterations_2 = tile_get_iterations(tile, 2);
    loop_size = tile_loop_size(tile, 2);
    for (int k = 0; k < loop_size; k++) {
      bres_calc(...);
    }

    // loop update
    iterations_list& iterations_3 = tile_get_iterations(tile, 3);
    loop_size = tile_loop_size(tile, 3);
    for (int k = 0; k < loop_size; k++) {
      update(...);
    }
  }
}
```

Listing 3.5: Airfoil loop-chain with SLOPE

Datasets

We use NACA (National Advisory Committee for Aeronautics) airfoils for our testing, which are airfoil shapes developed for aircraft wings. The meshes are named considering the number of cells or nodes of the mesh. For instance, the mesh of 720k nodes includes around 720k cells and 1.5 million edges. Airfoil meshes with several sizes were generated to analyze the behavior of the SLOPE library. The generated meshes include node sizes

45000, 180000, 720000, 2880000, 6480000, and 11520000. Meshes ranging from 45k to 2880k nodes were utilized in tests conducted on Scyrus and Telos, while meshes ranging from 720k to 11520k were used on ARCHER2. Mesh sizes were selected for each platform, based on computing node resources to minimize system noise and runtime comparison impact. Test results and graphs of 45k and 180k meshes on Scyrus and Telos, and 720k and 2880k meshes on ARCHER2 are given in Section B.1.1 of Appendix B.

Loop-chain

This application runs for 2000 iterations within a time-stepping loop that consists of five main loops: `adt_calc`, `res_calc`, `bres_calc`, `update`, and `save_soln`. The most compute-intensive loop, `res_calc`, performs approximately 100 floating-point operations per mesh edge and is executed 2000 times throughout the application's total execution.

The first two loops, `adt_calc` and `res_calc`, perform most computations. The `adt_calc` loop iterates over the cells, reads information from adjacent nodes and updates a dataset after performing calculations. The `res_calc` loop, on the other hand, iterates over the edges and computes the flux over interior edges. The third loop, `bres_calc`, iterates over boundary edges and computes flux over them. The fourth loop, `update`, iterates over the cells and updates values based on current calculations. Finally, the `save_soln` loop saves the previous solution by iterating over the cells of the mesh.

Out of these five loops, `save_soln` and `update` are direct loops, while the others use indirect mappings to read, write, or increment. The loops in Listing 3.5 are chained, and all calculations are performed using double-precision floating-point arithmetic.

Results and Analysis

Figure 3.5, Figure 3.7, and Figure 3.9 display single socket Airfoil runs on Scyrus, Telos and ARCHER2, respectively. Figure 3.6, Figure 3.8, and Figure 3.10 show dual socket Airfoil runs on Scyrus, Telos, and ARCHER2, respectively. In these graphs, the runtime of OP2 OpenMP Airfoil is marked for reference and it does not change with the tile size. In Table 3.2, we can see a summary of the best performance gains for the SLOPE Airfoil over the OP2 Airfoil for both single and dual socket runs on all three platforms. We observed that the dual socket runs provide better performance gains on Scyrus. However, on Telos and ARCHER2, the better-performing memory channel configuration (single socket or dual socket) varies depending on the mesh size. The SLOPE Airfoil attained peak performance of 44% for the 2880k mesh on Scyrus, 45% for the 2880k mesh on Telos, and 49% for the 11520k mesh on ARCHER2. This insight indicates that the given meshes have less impact on NUMA issues for the Airfoil application. Since airfoil meshes are 2D and have less complexity than multi-grid 3D meshes, the connectivity information of the mesh is not complex enough to disrupt its performance with NUMA. However, careful analysis shows that in some tile size configurations, the SLOPE Airfoil performed worse than the OP2 Airfoil due to poor cache performance with cache misses.

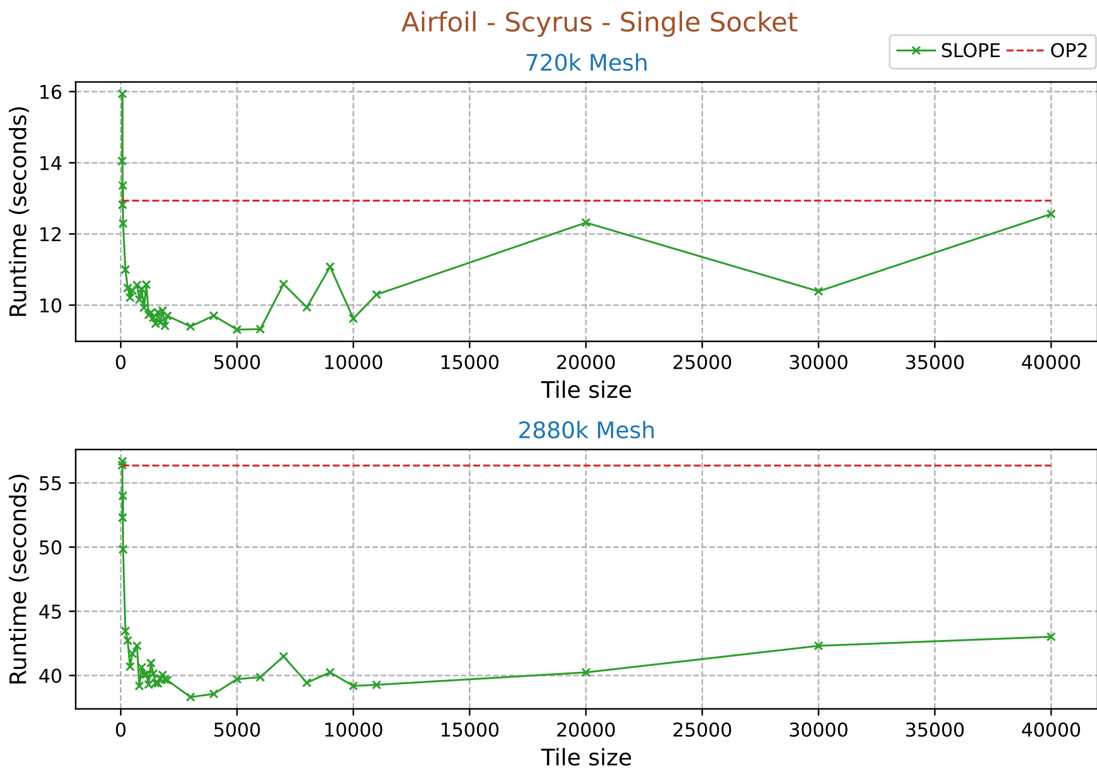


Figure 3.5: Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: single socket, 12 threads)

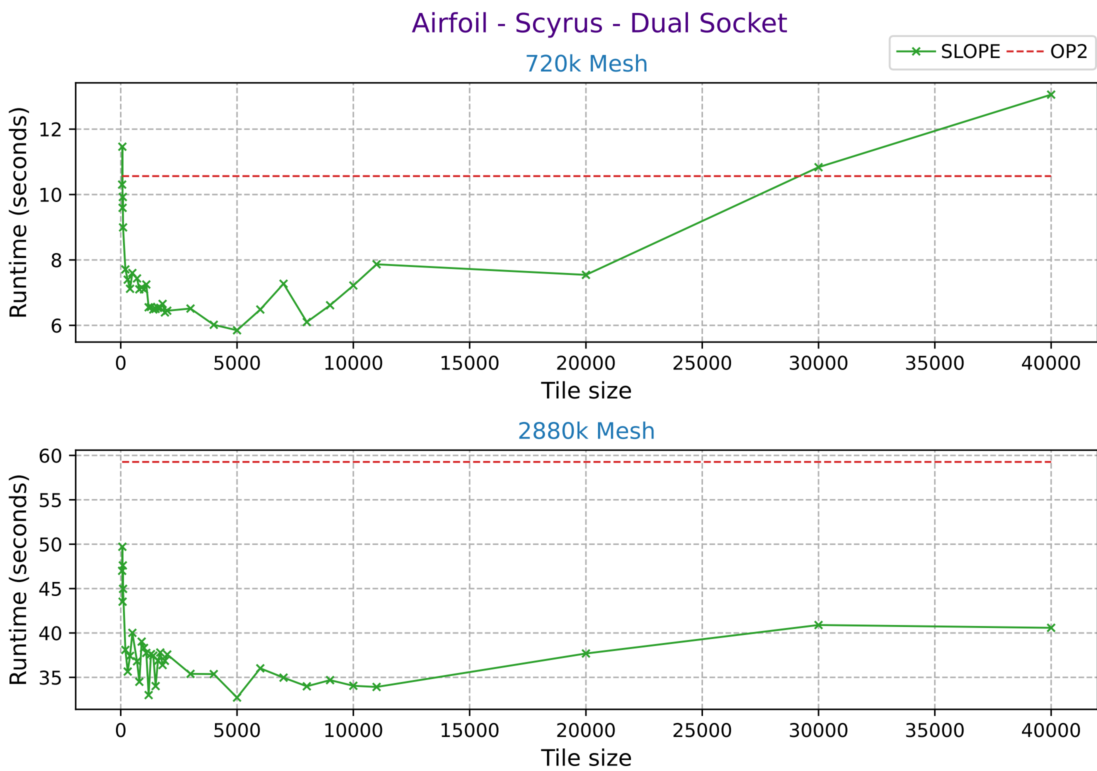


Figure 3.6: Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: dual socket, 24 threads)

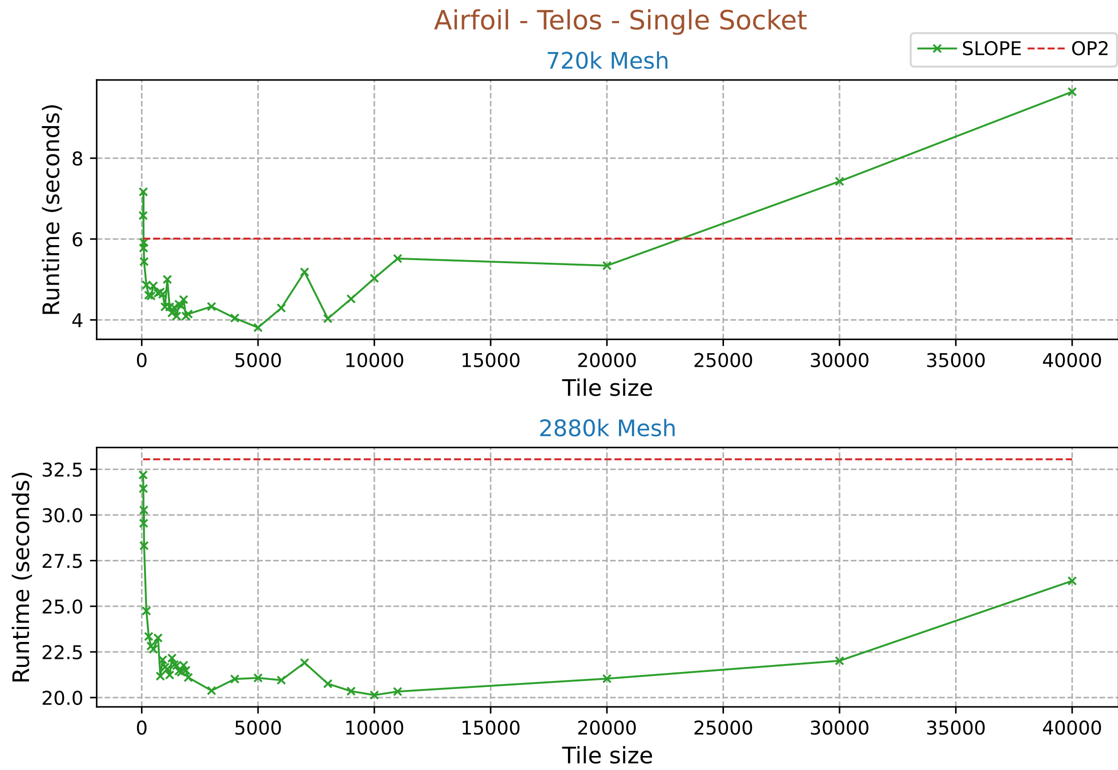


Figure 3.7: Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: single socket, 24 threads)

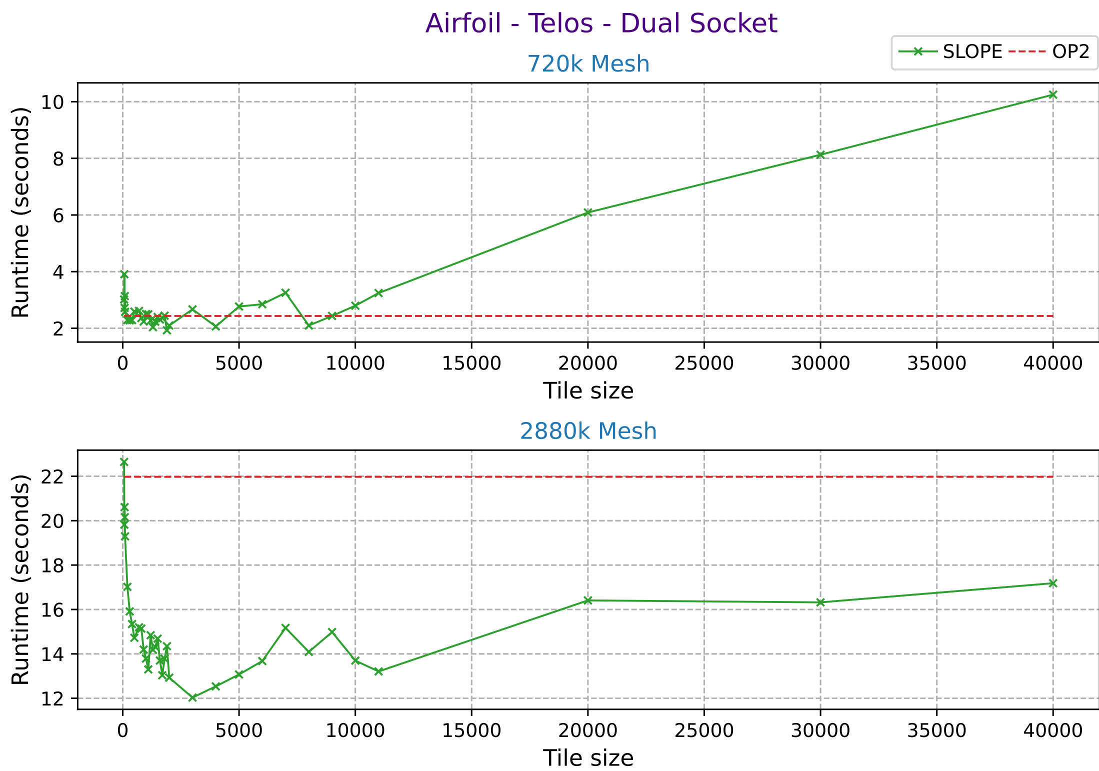


Figure 3.8: Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: dual socket, 48 threads)

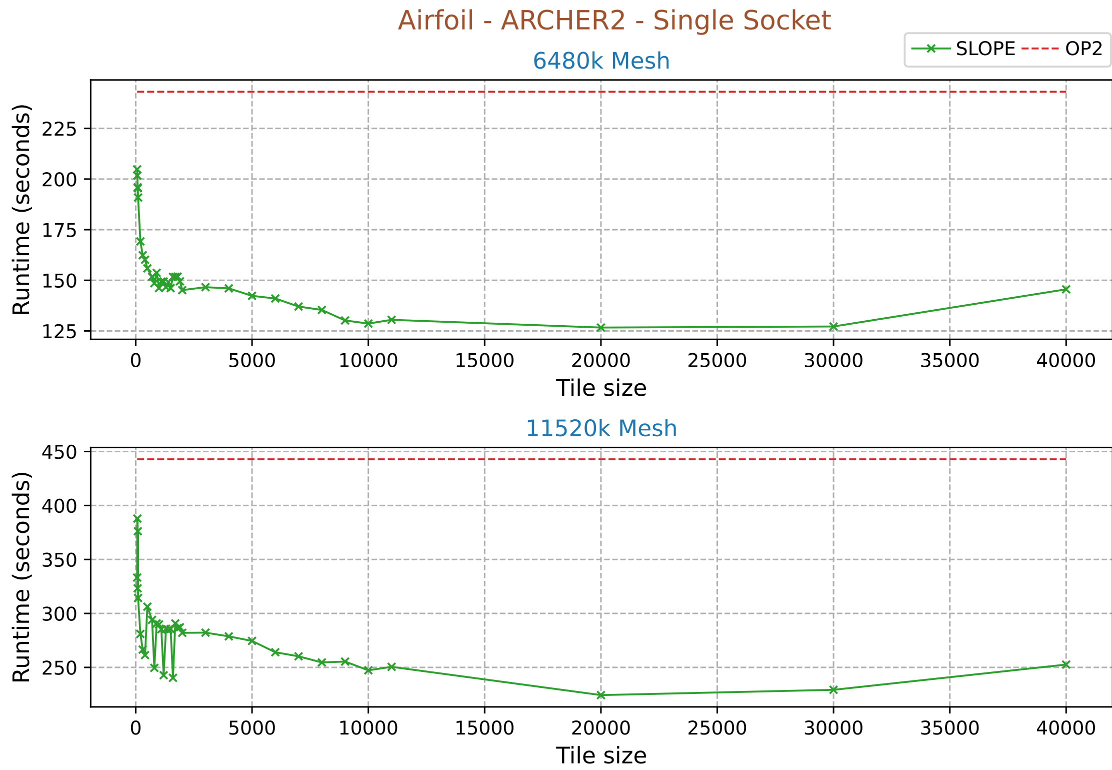


Figure 3.9: Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: single socket, 64 threads)

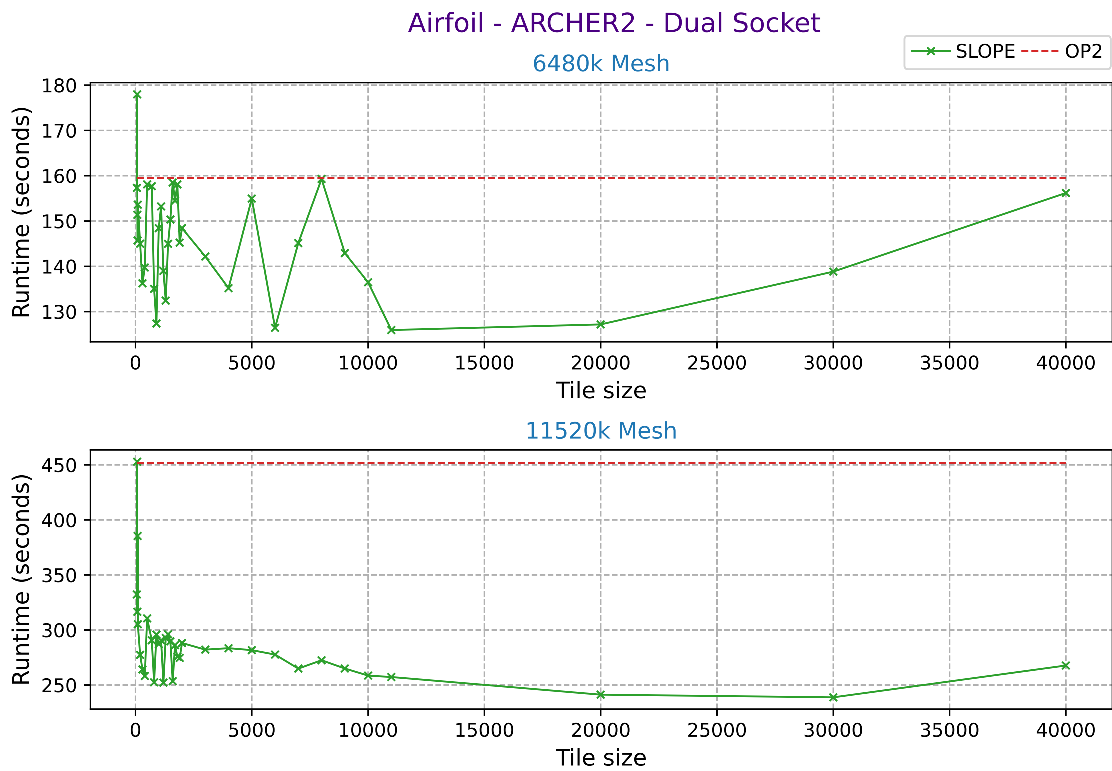
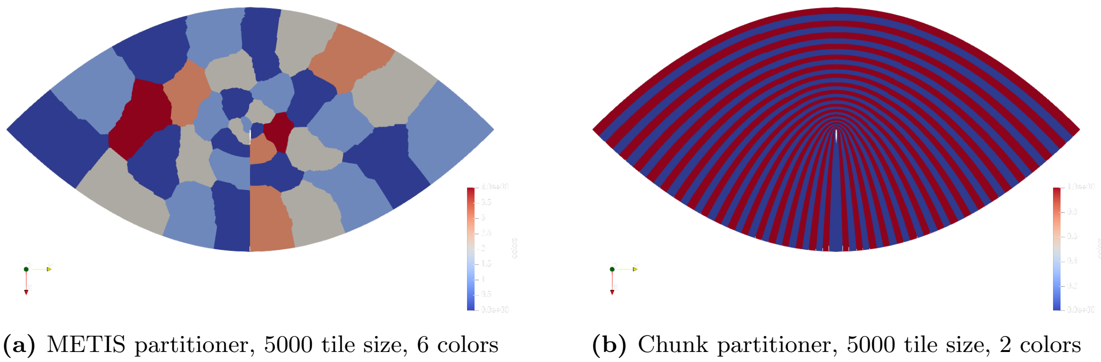


Figure 3.10: Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads)

Table 3.2: Best SLOPE Airfoil performance on Skylake, Telos, and ARCHER2

System	Dataset	Single Socket		Dual Socket	
		Tile Size	Gain %	Tile Size	Gain %
Scyrus	720k	5000	28.00	5000	44.63
	2880k	3000	32.01	5000	44.78
Telos	720k	5000	36.60	1900	20.73
	2880k	10000	39.07	3000	45.24
ARCHER2	6480k	20000	47.90	6000	20.71
	11520k	20000	49.33	30000	47.11



(a) METIS partitioner, 5000 tile size, 6 colors

(b) Chunk partitioner, 5000 tile size, 2 colors

Figure 3.11: 180k airfoil mesh partitioning and coloring

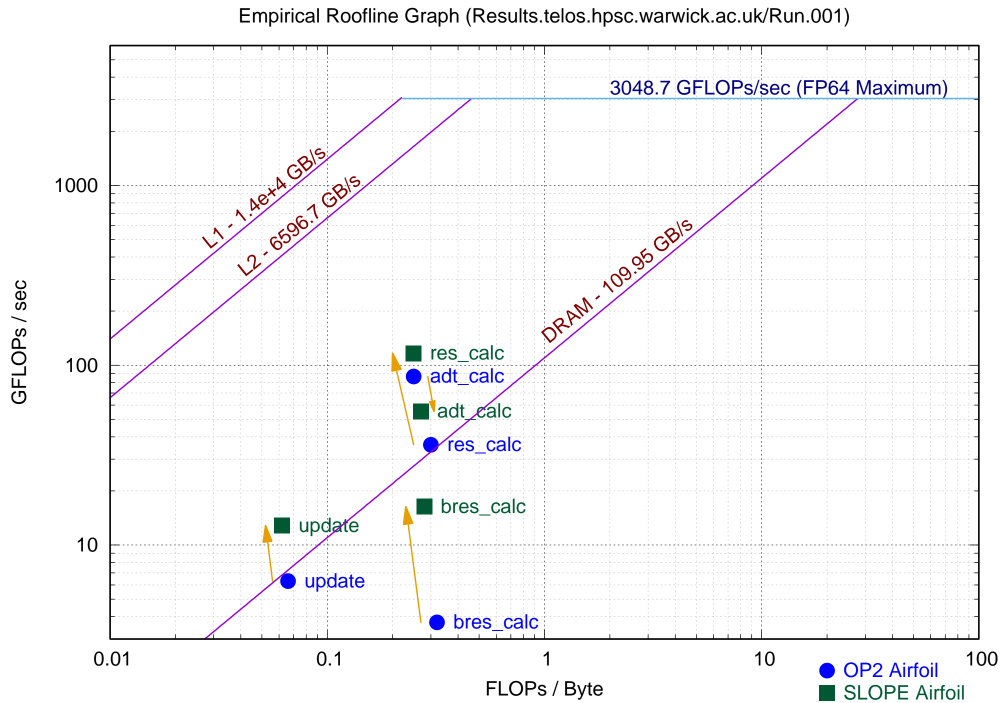


Figure 3.12: Roofline graph of SLOPE Airfoil and OP2 Airfoil (Configurations: Telos, 24 threads, single socket, 2880k mesh)

By using the Chunk partitioner instead of METIS, we are able to achieve higher performance gains. In Figure 3.11a, we can see the 180k airfoil mesh partitioned with a 5000 tile size and colored using six colors with the METIS partitioner. In comparison to that, Figure 3.11b displays the same mesh partitioned with the Chunk partitioner and colored using two colors. The nature of the partitions has a significant impact on the performance of the cache-blocking tiled execution of the application. A more detailed explanation of the impact of the mesh partitioner is given in Section 3.5.2.

Further, we perform a roofline analysis for both OP2 *omp* Airfoil and SLOPE Airfoil, of which the graph is shown in Figure 3.12. According to the roofline graphs, all the loops of SLOPE Airfoil except `adt_calc` are utilizing the memory of the platform better than the OP2 version. The graph establishes the performance gains that we witnessed in our tests and shows the fact that the more loops are memory-bound, the more performance gains from SLOPE incorporation [8].

3.4.2 MG-CFD

MG-CFD [83] is a 3D unstructured multi-grid, finite-volume computational fluid dynamics (CFD) mini-app for inviscid-flow, developed by extending the CFD solver in the Rodinia benchmark suite. In other words, it is a mini version of a commercially sensitive application named Hydra used by Rolls Royce plc. for the simulation of components of next-generation jet engines. This runs on a 3D mesh and employs multi-grid techniques to increase the convergence rate for iterative solvers. The geometry of the coarse levels is derived from the finest grid level. MG-CFD has been converted to use the OP2 API and its performance has been previously benchmarked in [126].

Datasets

For our experiments, we use the NASA Rotor 37 meshes, which represent the geometry of a transonic axial compressor rotor, widely used for validation in CFD. Considering the resources of the testing systems, we have gathered our experiment findings for MG-CFD using meshes containing 1M and 8M nodes.

Loop-chain

We are mainly interested in the four-stage Runge-Kutta (RK) scheme since that sequence of loops contains the features that are suitable to make it a SLOPE loop-chain. Namely, the RK loops are `compute_flux_edge`, `compute_bnd_node_flux`, `time_step` and `unstructured_stream`. The first loop, `compute_flux_edge` iterates over the edges and computes the flux over the interior edges whereas the second loop, `compute_bnd_node_flux` iterates over the boundary edges and computes the flux over them. `time_step` is a direct loop that iterates over the nodes of the mesh and performs some increments and updates values based on a previous status. The `unstructured_stream` loop was introduced as a memory-bound loop that has the same

```
int ncolors = exec_num_colors(exec[level]);

for (int rkcycle = 0; rkcycle < RK; rkcycle++) {
  for (int color = 0; color < ncolors; color++) {
  // for all tiles of this color
  const int n_tiles_per_color = exec_tiles_per_color(exec[level], color);

    #pragma omp parallel for
    for (int j = 0; j < n_tiles_per_color; j++) {
      // execute the tile
      tile_t* tile = exec_tile_at(exec[level], color, j);
      int loop_size;

      // loop compute_flux_edge
      iterations_list& le2n_0 = tile_get_local_map(tile, 0, "e2n");
      iterations_list& iterations_0 = tile_get_iterations(tile, 0);
      loop_size = tile_loop_size(tile, 0);
      for (int k = 0; k < loop_size; k++) {
        compute_flux_edge_kernel(...);
      }

      // loop compute_bnd_node_flux
      iterations_list& lbe2n_1 = tile_get_local_map(tile, 1, "bn2n");
      iterations_list& iterations_1 = tile_get_iterations(tile, 1);
      loop_size = tile_loop_size(tile, 1);
      for (int k = 0; k < loop_size; k++) {
        compute_bnd_node_flux_kernel(...);
      }

      // loop time_step
      iterations_list& iterations_2 = tile_get_iterations(tile, 2);
      loop_size = tile_loop_size(tile, 2);
      for (int k = 0; k < loop_size; k++) {
        time_step_kernel(...);
      }

      // loop unstructured_stream_kernel
      iterations_list& le2n_3 = tile_get_local_map(tile, 3, "e2n");
      iterations_list& iterations_3 = tile_get_iterations(tile, 3);
      loop_size = tile_loop_size(tile, 3);
      for (int k = 0; k < loop_size; k++) {
        unstructured_stream_kernel(...);
      }
    }
  }
}
```

Listing 3.6: MG-CFD RK loop-chain with SLOPE

data access pattern as `compute_flux_edge` with very few FLOPS. Listing 3.6 details the explained RK loops with SLOPE tiling.

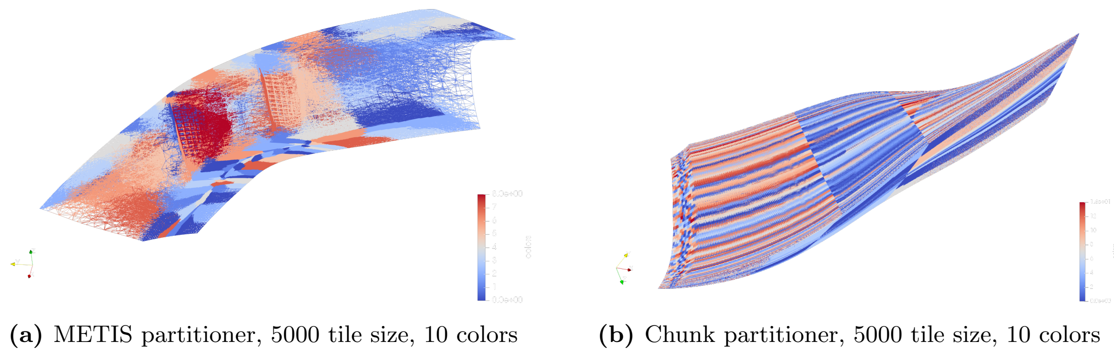
Test Results and Analysis

We conduct a comparison between the SLOPE MG-CFD version and the standard OP2 OpenMP (*omp*) version of MG-CFD. In addition to the common testing strategies mentioned in Section 3.4, we tested MG-CFD with SLOPE using various loop-chain and grid level combinations as outlined in Table 3.3. Like with Airfoil, we present our results using the Chunk partitioner as it yielded the best performance numbers for MG-CFD. The MG-CFD mesh, partitioned with a tile size of 5000 and colored with ten colors using the METIS partitioner, is shown in Figure 3.11a. Figure 3.11b shows the same mesh partitioned with the Chunk partitioner and colored using ten colors.

Table 3.3: SLOPE MG-CFD loop fusion schemes

Fusion Scheme	Grid Level Configuration	Loop Count Configuration	Description
fs1	All four grid levels	Single loop-chain	Adding all RK loops to a single chain and performing calculations for all 4 grid levels.
fs2	All four grid levels	Two loop-chains	Expand the RK loop such that the `time_step` loop is added before and after the `unstructured_stream` loop and create two loop-chains. **loop-chain1** – `compute_flux_edge`, `compute_bnd_node_flux`, `time_step` **loop-chain2** – `unstructured_stream`, `time_step` Perform calculations for all 4 grid levels.
fs3	Fine grid only	Single loop-chain	Adding all RK loops to a single chain and perform calculations only for the fine grid.
fs4	Fine grid only	Two loop-chains	Expand the RK loop such that the `time_step` loop is added before and after the `unstructured_stream` loop and create two loop-chains. **loop-chain1** – `compute_flux_edge`, `compute_bnd_node_flux`, `time_step` **loop-chain2** – `unstructured_stream`, `time_step` Perform calculations only for the fine grid.

We tested all four approaches mentioned in Table 3.3 for the NASA Rotor 37 1M and 8M node datasets. When analyzing the performance results, we understand that the highest performance gains are visible for fs3 and fs4. The results summary in Table 3.4, proves the given statement. This observation indicates that in multi-grid



(a) METIS partitioner, 5000 tile size, 10 colors (b) Chunk partitioner, 5000 tile size, 10 colors

Figure 3.13: 1M MG-CFD mesh partitioning and coloring

systems, SLOPE will give maximum performance for calculations performed on the fine mesh than the coarsened meshes due to the reduced spatial locality of the coarsened grid levels.

Table 3.4: SLOPE MG-CFD single socket performance summary on Scyrus for loop fusion schemes in Table 3.3

Dataset	Fusion Scheme Gain %			
	fs1	**fs2**	**fs3**	**fs4**
Rotor37 1M	6.56	0.48	5.71	24.84
Rotor37 8M	5.74	8.29	11.91	33.30

Table 3.5: Best SLOPE MG-CFD performance on Skylake, Telos, and ARCHER2[1]

System	Dataset	Single Socket		Dual Socket	
		Tile Size	Gain %	Tile Size	Gain %
Scyrus	1M	1800	24.84	4000	37.83
	8M	7000	33.30	5000	50.32
Telos	1M	1800	31.57	4000	47.42
	8M	5000	44.22	1500	-46.22
ARCHER2	1M	20000	-3.12		
	8M	50000	46.07		

Figure 3.14, Figure 3.16 and Figure 3.18 show the runtime variation of the SLOPE MG-CFD with different tile sizes for 1M and 8M meshes on Scyrus, Telos, and ARCHER2 for single socket runs. For dual socket runs, the runtime variation of the SLOPE MG-CFD with different tile sizes on Scyrus and Telos for the same meshes are shown in Figure 3.15 and Figure 3.17, respectively. MG-CFD performance for all three platforms is summarized in Table 3.5. Analysis of the results summary reveals that MG-CFD provides similar performance gains for both single and dual socket runs on Scyrus, while on Telos, we observe negative gains for 8M mesh for the dual socket run, but better gains

[1]Dual socket MG-CFD runs with 128 threads on ARCHER2 have been excluded due to result deviation.

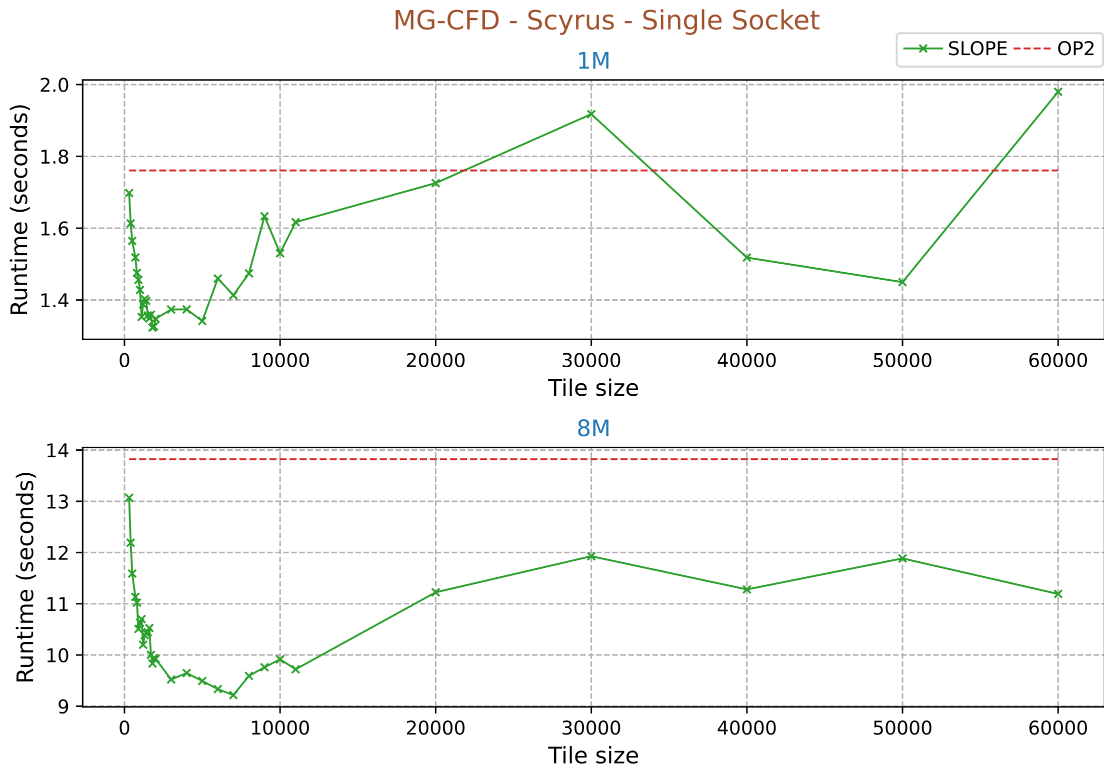


Figure 3.14: Runtime variation of SLOPE MG-CFD with tile sizes on Scyrus (Configurations: single socket, 12 threads)

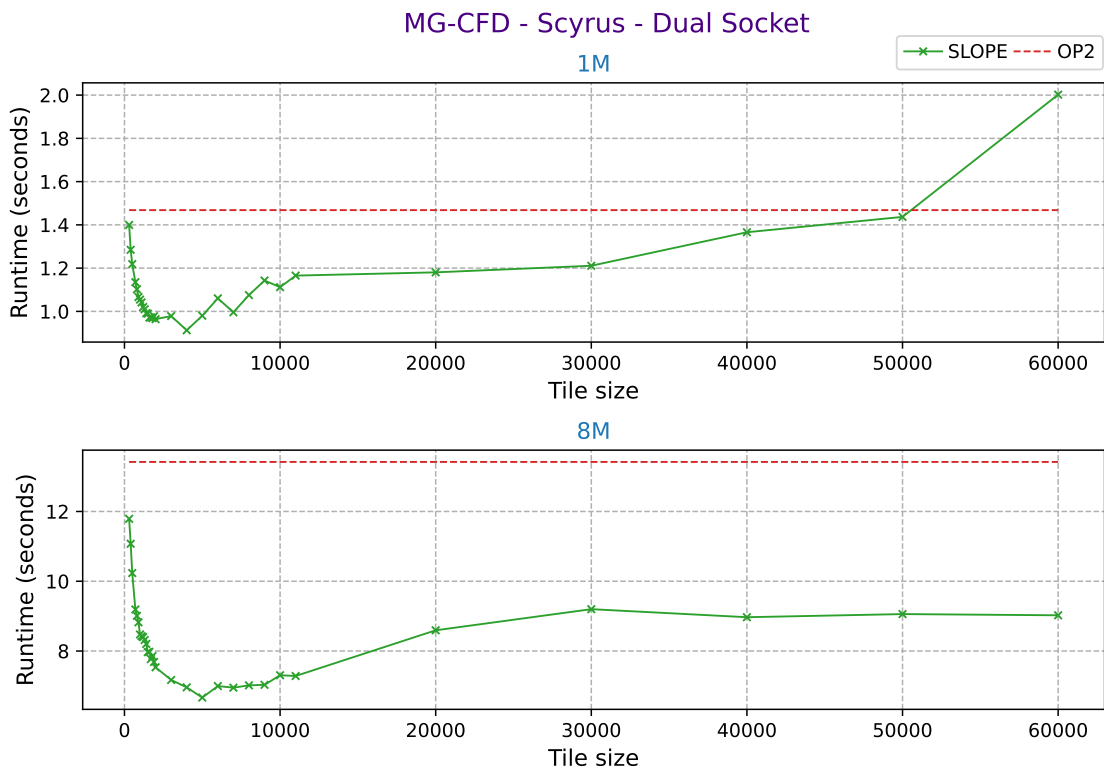


Figure 3.15: Runtime variation of SLOPE MG-CFD with tile sizes on Scyrus (Configurations: dual socket, 24 threads)

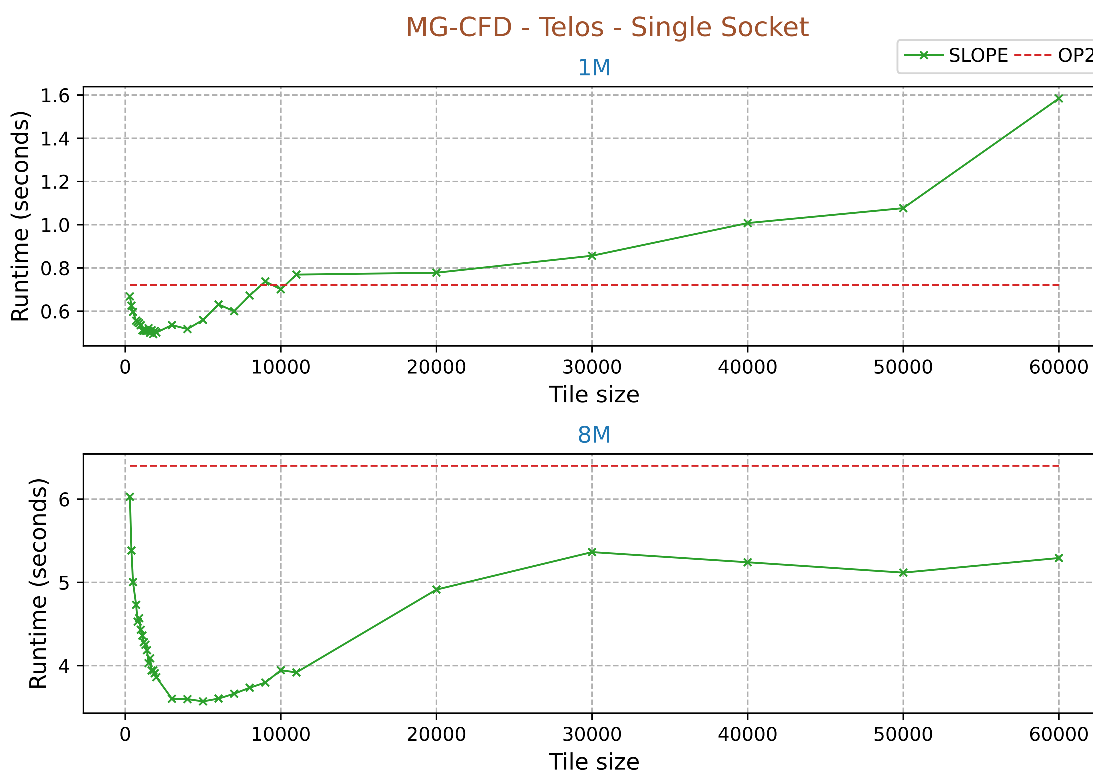


Figure 3.16: Runtime variation of SLOPE MG-CFD with tile sizes on Telos (Configurations: single socket, 24 threads)

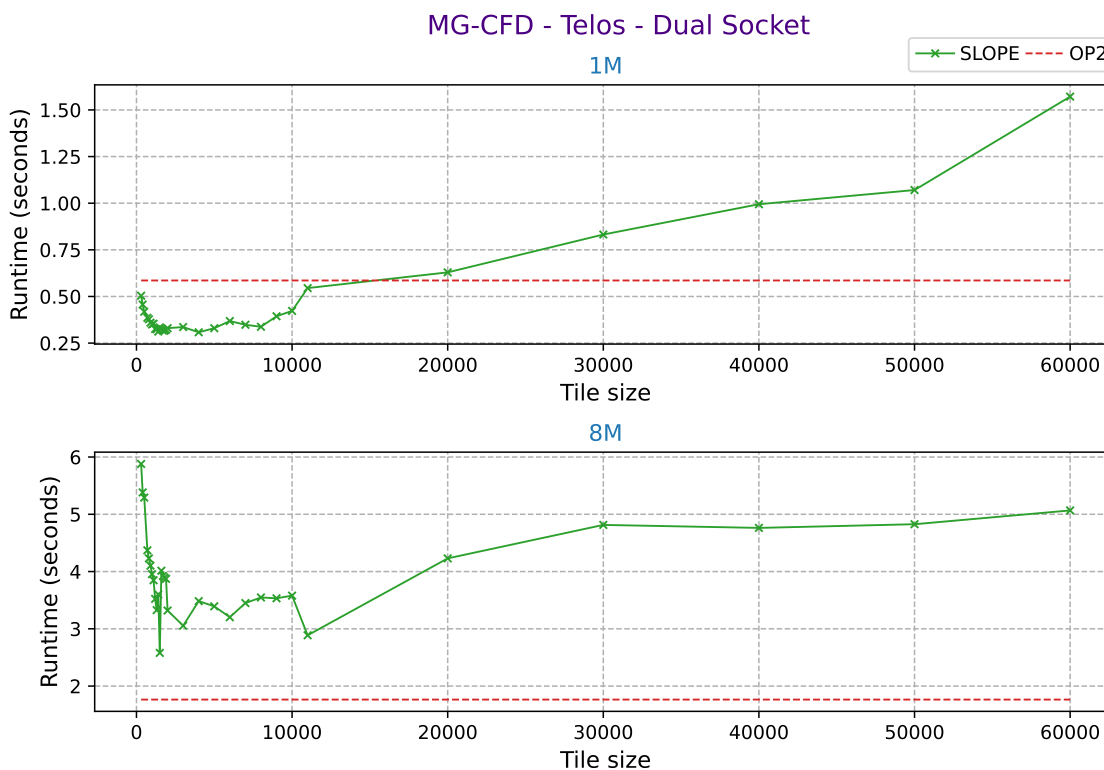


Figure 3.17: Runtime variation of SLOPE MG-CFD with tile sizes on Telos (Configurations: dual socket, 48 threads)

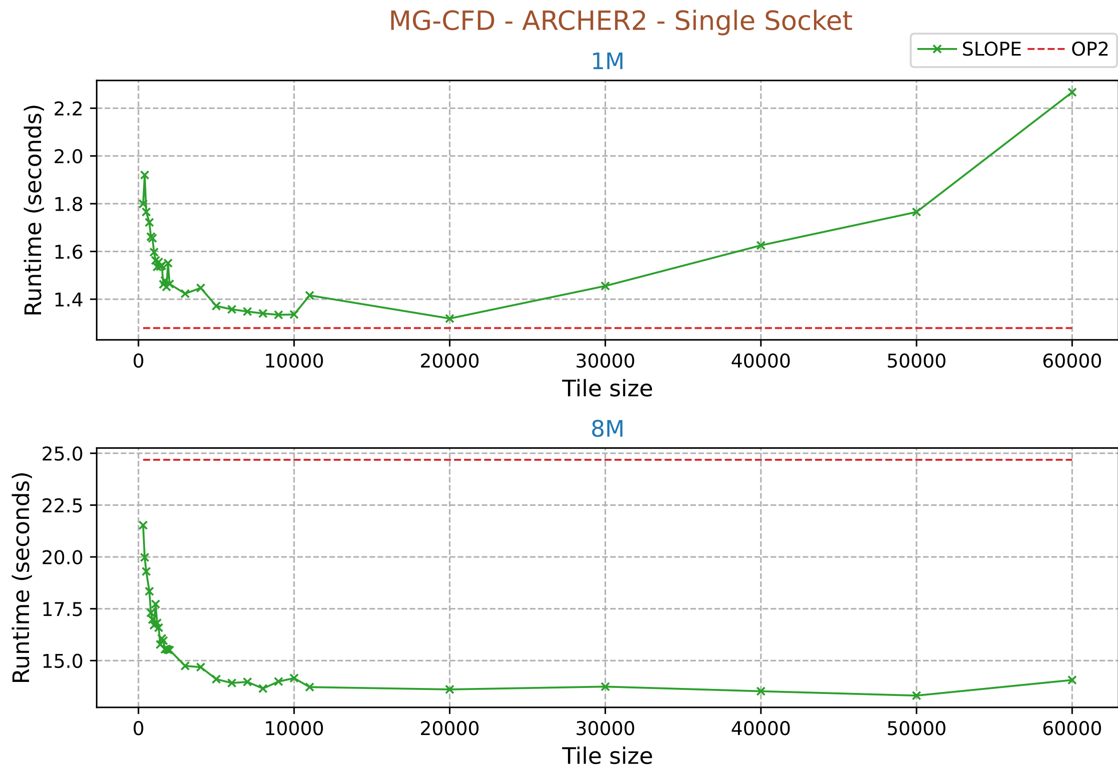


Figure 3.18: Runtime variation of SLOPE MG-CFD with tile sizes on ARCHER2 (Configurations: single socket, 64 threads)

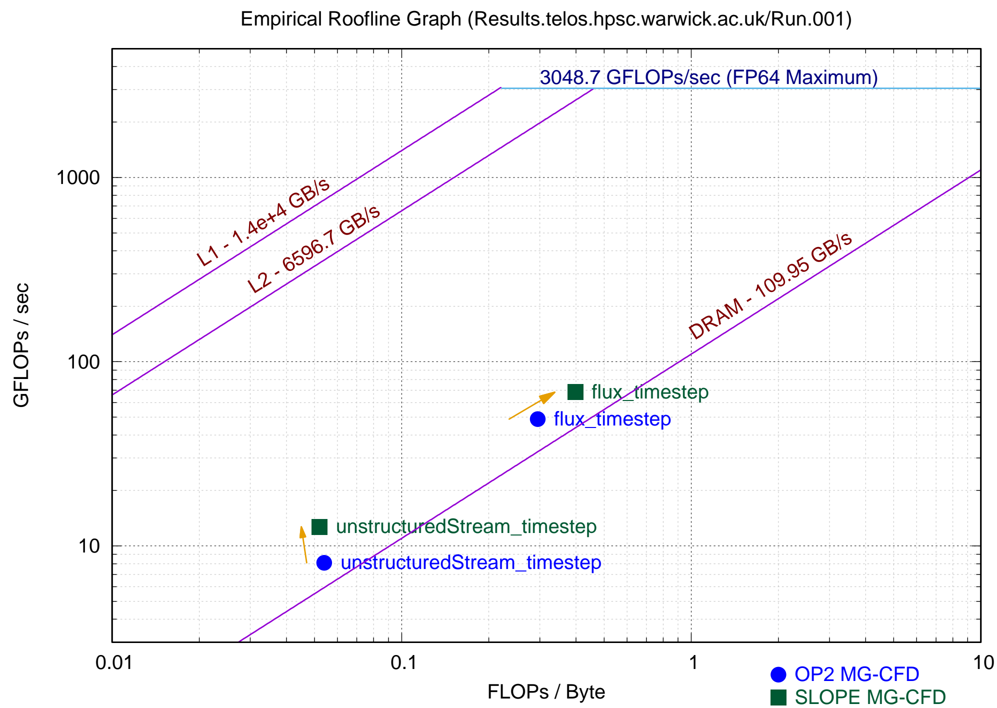


Figure 3.19: Roofline graphs of SLOPE MG-CFD and OP2 MG-CFD (Configurations: Telos, 24 threads, single socket, Rotor37 1M, fs4)

for the 1M mesh for both the single and dual socket runs. Dual socket runs on Telos encounter NUMA issues that adversely impact the 8M mesh run. On ARCHER2, better gains are observed for the 8M mesh with single socket runs. However, within the tile size range of our tests, the 1M mesh runs did not perform well on ARCHER2. The processing power and memory capacity of ARCHER2 nodes are higher compared to those of Scyrus and Telos nodes, as shown in Table 3.1. We presume that the resources in ARCHER2 are more suitable for larger meshes, allowing for performance benefits through cache-blocking tiling.

We conducted roofline tests for OP2 *omp* MG-CFD and SLOPE MG-CFD, and the results are displayed in the graph in Figure 3.19. The roofline graphs indicate that SLOPE MG-CFD's loop-chains are using the platform's memory more efficiently than the OP2 version. The graph also confirms the performance improvements we observed during our tests, demonstrating that the more memory-bound the loops are, the greater the performance gain when utilizing the SLOPE library [8].

3.4.3 Volna

Volna is a novel tool that has been designed to model tsunami waves [84]. Volna acts as a solver that can handle all aspects of a tsunami's life cycle, from its generation, through propagation and finally to the point where it runs up along the coast. It achieves this by applying finite-volume non-linear shallow-water equations (NSWE) to unstructured triangular meshes thus can be run in arbitrary complex domains. Furthermore, an OP2 version of Volna has been developed which separates the scientific code from various parallel implementations [127]. This makes Volna an ideal choice for operational purposes and a tool with enormous potential for modeling tsunami waves and achieving high performance on various platforms.

Datasets

Table 3.6 provides the details of the datasets utilized during our testing.

Table 3.6: Dataset properties

Dataset	#Nodes	#Edges	#Cells
Catalina	49456	147653	98198
NU3	475292	1424856	949565

Loop-chain

We consider the main stages in the Volna simulation for our experiments. These stages make up 90% of the total runtime and are crucial for the program's performance [127]. The OP2 Volna performs the relevant computations of these stages using five parallel loops. In the `computeGradient` and `limiter` loops, the program updates some state

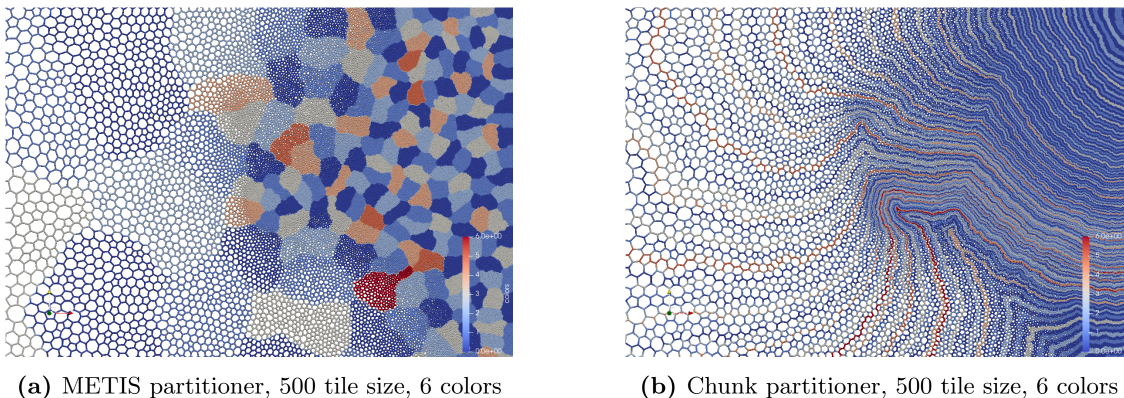
```
1  while (timestamp < ftime) {
2    ...
3    for (int color = 0; color < ncolors; color++) {
4      // for all tiles of this color
5      const int n_tiles_per_color = exec_tiles_per_color(exec[level], color);
6
7      #pragma omp parallel for
8      for (int j = 0; j < n_tiles_per_color; j++) {
9        tile_t* tile = exec_tile_at(exec[level], color, j); int loop_size;
10
11       // loop computeGradient
12       iterations_list& lc2c_0 = tile_get_local_map(tile, 0, "c2c");
13       iterations_list& iterations_0 = tile_get_iterations(tile, 0);
14       loop_size = tile_loop_size(tile, 0);
15       #pragma omp simd
16       for (int k = 0; k < loop_size; k++) {
17         computeGradient(...);
18       }
19
20       // loop limiter
21       iterations_list& lc2e_1 = tile_get_local_map(tile, 1, "c2e");
22       iterations_list& iterations_1 = tile_get_iterations(tile, 1);
23       loop_size = tile_loop_size(tile, 1);
24       #pragma omp simd
25       for (int k = 0; k < loop_size; k++) {
26         limiter(...);
27       }
28
29       // loop computeFluxes
30       iterations_list& le2c_2 = tile_get_local_map(tile, 2, "e2c");
31       iterations_list& iterations_2 = tile_get_iterations(tile,2);
32       loop_size = tile_loop_size(tile, 2);
33       #pragma omp simd
34       for (int k = 0; k < loop_size; k++) {
35         computeFluxes(...);
36       }
37
38       // loop numericalFluxes
39       iterations_list& iterations_3 = tile_get_iterations(tile, 3);
40       loop_size = tile_loop_size(tile, 3);
41       #pragma omp simd
42       for (int k = 0; k < loop_size; k++) {
43         numericalFluxes(...);
44       }
45
46       // loop spaceDiscretization
47       iterations_list& le2c_4 = tile_get_local_map(tile, 4, "e2c");
48       iterations_list& iterations_4 = tile_get_iterations(tile, 4);
49       loop_size = tile_loop_size(tile, 4);
50       for (int k = 0; k < loop_size; k++) {
51         spaceDiscretization(...);
52       }
53     }
54   }
55 }
```

Listing 3.7: Volna loop-chain with SLOPE

variables defined on cells while iterating over the cells of the mesh. The `computeFluxes` loop computes physical properties across the mesh edges and requires large amounts of data to be accessed indirectly through *edges-to-cells*, `e2c` mapping. After initializing the output data array defined over the cells, `SpaceDiscretization` loop applies fluxes to the cell-centered state variables. This loop becomes complex due to the indirect increment access patterns via the *edges-to-cells*, `e2c` mappings. The Volna loop-chain used for performance testing with the SLOPE library is detailed in Listing 3.7.

Test Results and Analysis

During the testing process, the SLOPE Volna version was compared to the standard OP2 OpenMP (*omp*) version of Volna. The testing was conducted on all three platforms mentioned in Table 3.1, using both single and dual socket configurations. The SLOPE version utilized the Chunk partitioner, which resulted in better performance. The partitioning of the Catalina mesh using the METIS partitioner with a tile size of 500 and colored using six colors can be seen in Figure 3.20a. Additionally, the same mesh partitioned using the Chunk partitioner with a tile size of 500 and colored using six colors can be seen in Figure 3.20b.



(a) METIS partitioner, 500 tile size, 6 colors

(b) Chunk partitioner, 500 tile size, 6 colors

Figure 3.20: Volna Catalina mesh partitioning and coloring

Table 3.7: Best SLOPE Volna performance on Skylake, Telos, and ARCHER2

System	Dataset	Single Socket		Dual Socket	
		Tile Size	Gain %	Tile Size	Gain %
Scyrus	Catalina	2000	-13.44	800	-7.90
	NU3	40000	0.001	20000	2.49
Telos	Catalina	1900	-9.69	1500	-12.82
	NU3	20000	-0.09	10000	-17.16
ARCHER2	Catalina	800	27.71	1500	50.90
	NU3	9000	-40.94	9000	-50.65

Figure 3.21, Figure 3.23, and Figure 3.25 display the runtime variation of SLOPE Volna single socket runs for different tile sizes on Scyrus, Telos, and ARCHER2,

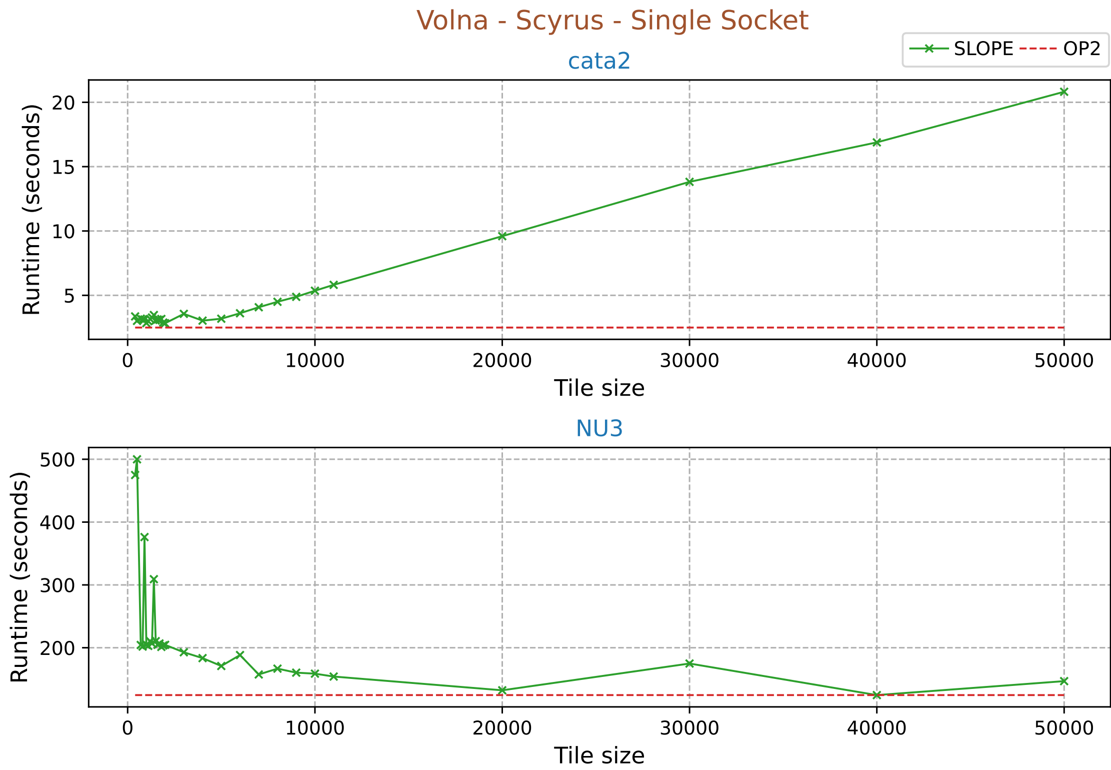


Figure 3.21: Runtime variation of SLOPE Volna with tile sizes on Scyrus (Configurations: single socket, 12 threads)

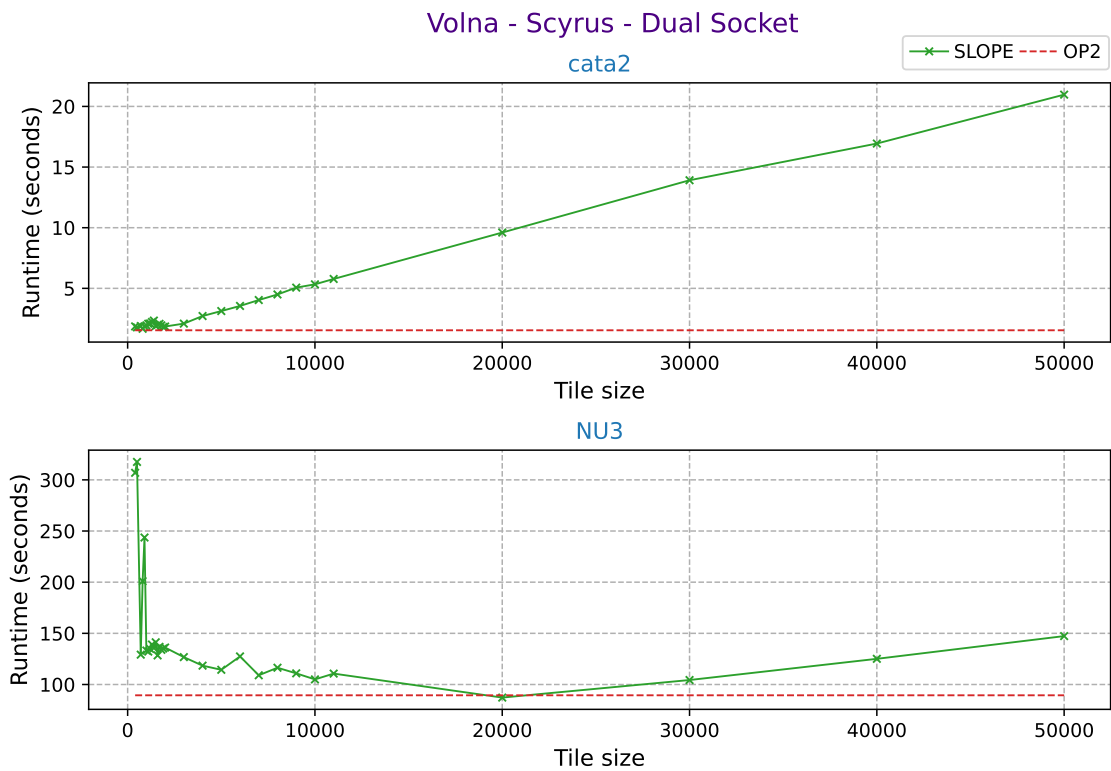


Figure 3.22: Runtime variation of SLOPE Volna with tile sizes on Scyrus (Configurations: dual socket, 24 threads)

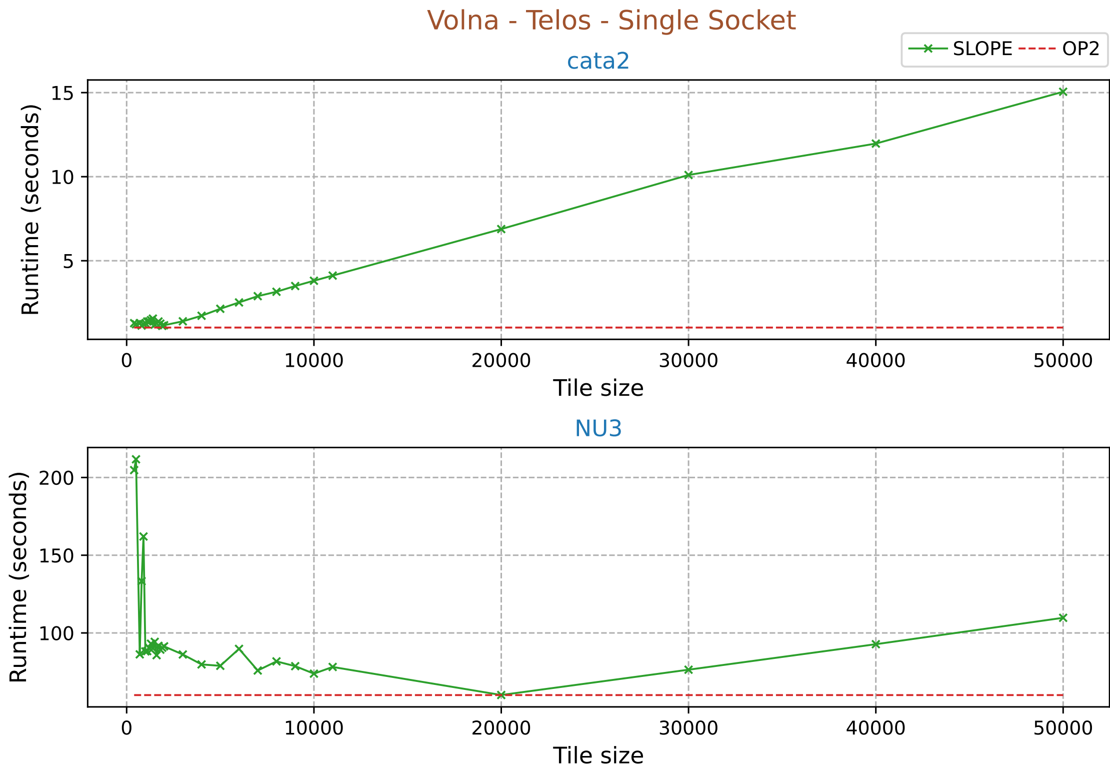


Figure 3.23: Runtime variation of SLOPE Volna with tile sizes on Telos (Configurations: single socket, 24 threads)

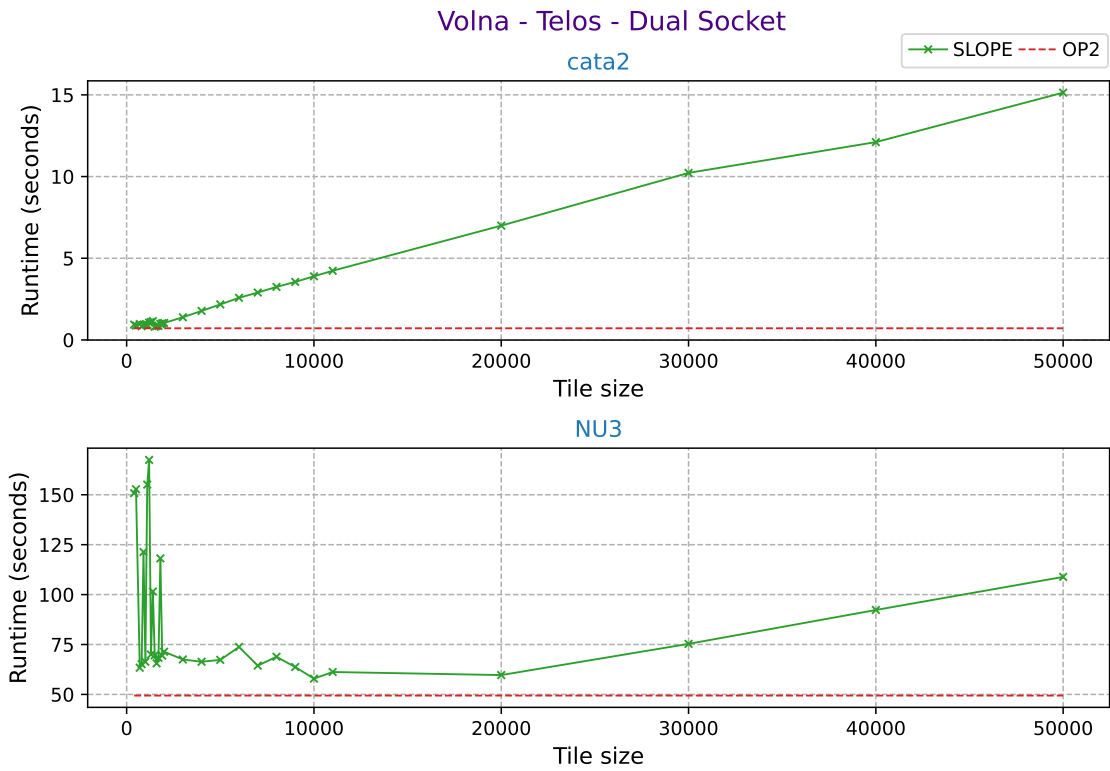


Figure 3.24: Runtime variation of SLOPE Volna with tile sizes on Telos (Configurations: dual socket, 48 threads)

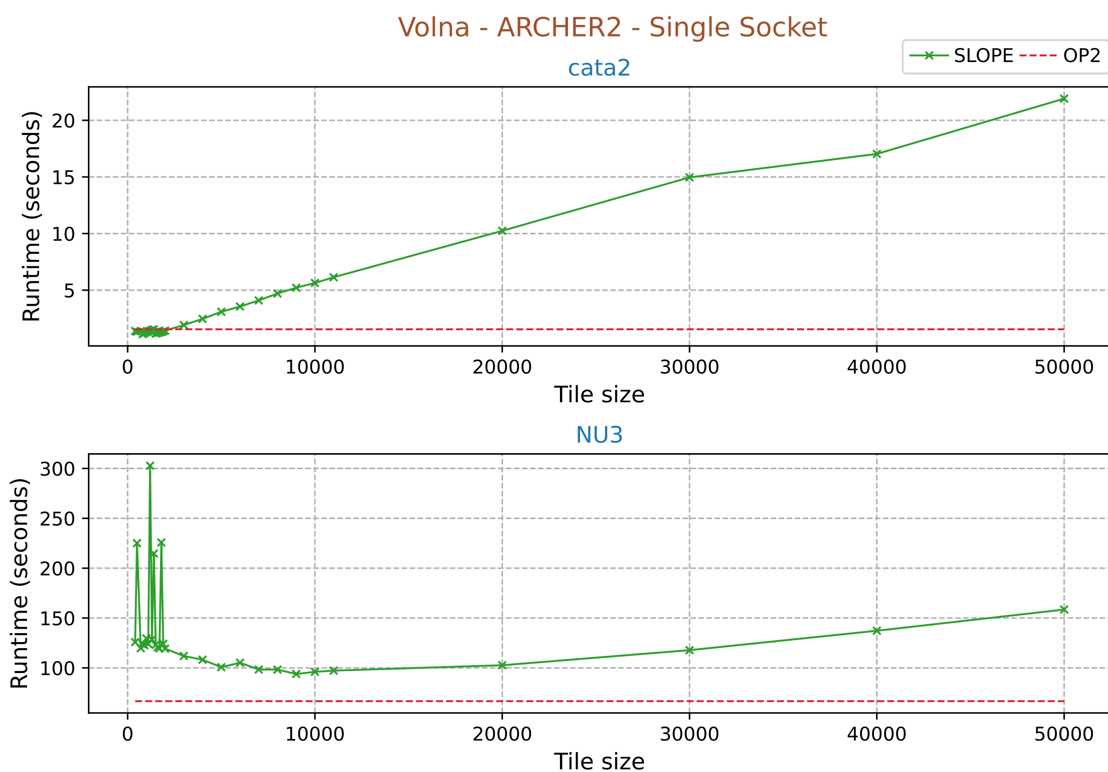


Figure 3.25: Runtime variation of SLOPE Volna with tile sizes on ARCHER2 (Configurations: single socket, 64 threads)

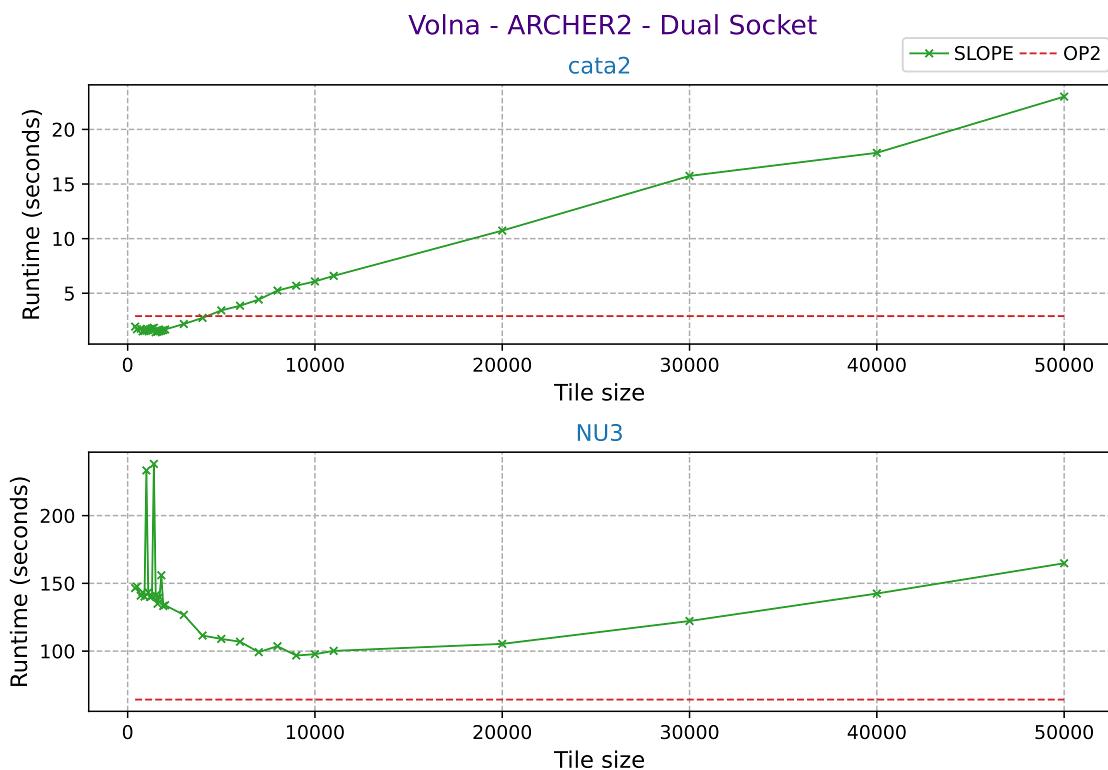


Figure 3.26: Runtime variation of SLOPE Volna with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads)

respectively. These runs were conducted on the datasets outlined in Table 3.6. The corresponding dual socket runs for the same datasets on the same systems are depicted in Figure 3.22, Figure 3.24, and Figure 3.26. Based on the run summary presented in Table 3.7, SLOPE Volna failed to produce significant performance benefits on both Scyrus and Telos, regardless of the number of sockets used. However, for the Catalina dataset, SLOPE Volna performed better on ARCHER2, both for the single and dual socket runs.

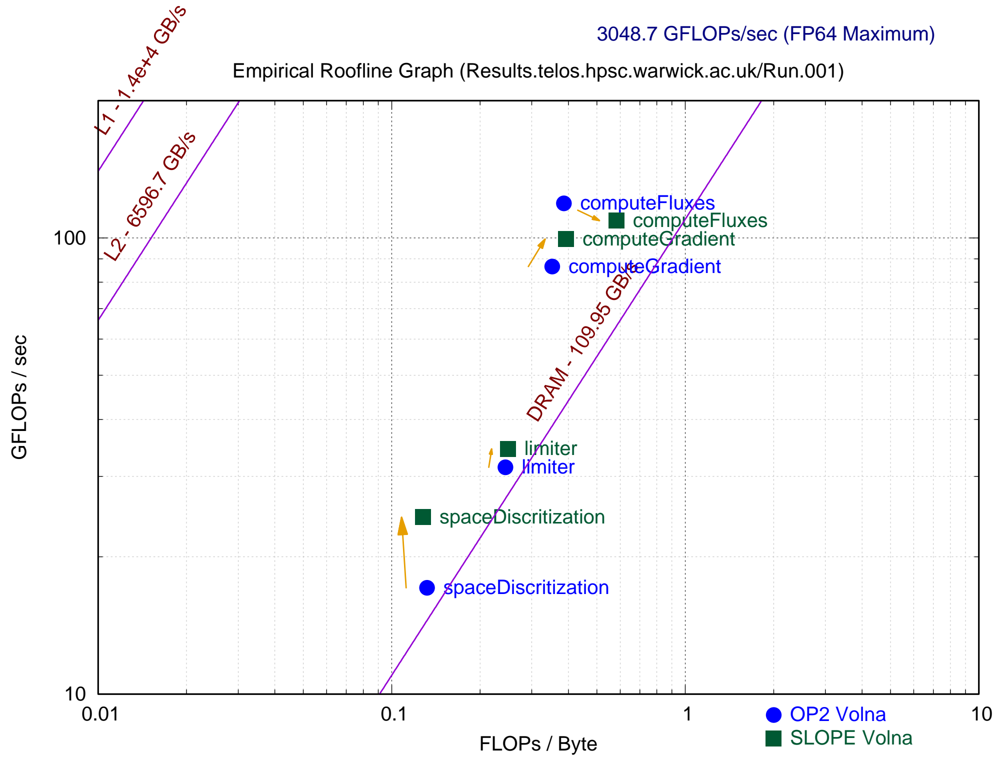


Figure 3.27: Roofline graphs of SLOPE Volna and OP2 Volna (Configurations: Telos, 24 threads, single socket, Catalina)

The roofline test conducted on Telos for the Catalina dataset, as shown in Figure 3.27, suggests that Volna kernels are compute-bound and do not benefit from cache-blocking tiling. Further, computing kernels need to be memory-bound to see performance improvements with SLOPE [8]. However, the complex nature of the Volna meshes utilized during testing and the partitioning nature as shown in Figure 3.20 may have adversely affected the performance of SLOPE Volna.

3.4.4 OP2 Hydra

Figure 3.28 depicts a gas turbine engine consisting of a fan, compressor, combustor, and turbine. The engine's compressor is made up of several blade rows attached to a shaft that rotates at high speeds to compress the cold air entering the engine at atmospheric pressure. The compressed air is then mixed with fuel and ignited inside the combustion chamber. The resulting hot air drives the turbine, which in turn drives both the compressor and the fan. The total thrust needed to propel the aircraft is produced by the fan and the hot air exhaust after combustion. To simulate the full compressor, several models of Reynold's Average Navier-Stokes (RANS) or hybrid RANS/Large eddy simulation (LES) are required. These models represent a simulation of rotor and stator blades that are interlinked with sliding plane interfaces. The Hydra [63] application represents each of these rotors and stators [80].

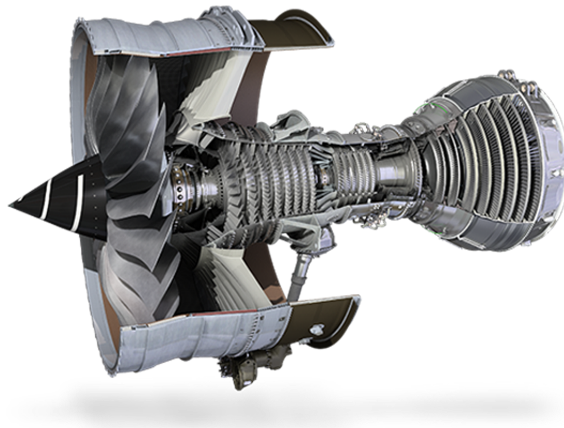


Figure 3.28: RR Trent XWB engine (©Rolls Royce plc. Reproduced with permission.)

Hydra [63] is a full-scale production application developed for modeling various aspects of turbomachinery design. It is an unstructured-mesh finite-volume solver for the compressible RANS equations in their steady and unsteady formulation (RANS/URANS). It uses a 5-step Runge-Kutta method for time-stepping, with multi-grid and block-Jacobi preconditioning. The OP2 version of Rolls Royce's Hydra [80, 81] is developed and it consists of around 500 parallel loops with significantly more complex computations performed on the mesh than the loops in MG-CFD. The SLOPE library does not currently support applications written in Fortran. However, there are many industrial legacy code bases that are Fortran-based, such as Hydra. To address this need, we have developed a Fortran-based API for the SLOPE library, which enables the use of its sparse tiling features in Fortran-based applications.

Datasets

For our experiments, we utilize NASA Rotor 37 meshes containing 1M and 8M nodes. These meshes represent the geometry of a transonic axial compressor rotor, which is widely used for validation in CFD.

Loop-chains

We identified 3 main time-consuming loop-chains in Hydra that come inside the main time-stepping loop. Namely, they are `iflux`, `vflux`, and `jinit`. The loop-chain information is given in Table 3.8.

Table 3.8: Loop-chain information

Loop-chain	Loop count	Loop names	Iteration set
`iflux`	2	`initviscres`	`nodes`
		`iflux_edge`	`edges`
`vflux`	2	`initres`	`nodes`
		`vflux_edge`	`edges`
`jinit`	3	`jac_init`	`nodes`
		`jaca_init`	`nodes`
		`accumedges`	`edges`

One of the main requirements to *slopify* a loop-chain is that the iteration space of the seed loop should represent the whole mesh so that the iterations of the other loops can be linked with the seed loop iteration partitioning. In other terms, the iteration spaces of other loops should be within the mesh elements represented by the seed loop iteration space. Some of the loop-chains in Hydra could not be *slopified* due to not fulfilling this requirement.

Test Results and Analysis

We compare the individual loop-chains of Hydra's SLOPE version and OP2 OpenMP (*omp*) version to evaluate their performance. We vary the tile size according to empirical intervals to measure the loop-chain's performance. The most significant performance improvements observed with the NASA Rotor 37 datasets (1M and 8M) are presented in Table 3.9. Figure 3.29, Figure 3.31, and Figure 3.33 show the single socket runs of Hydra on Scyrus, Telos, and Hydra, respectively. On the other hand, Figure 3.30, Figure 3.32, and Figure 3.34 represent the dual socket runs of the same tests.

Upon analyzing the results, it is evident that the loop-chains, `iflux` and `jinit` display the most significant improvement in performance. Notably, while Scyrus and Telos perform better with the 1M mesh, ARCHER2 outperforms with the 8M mesh. Our testing shows that using a single or dual socket has a minimal impact on performance, indicating no significant issues with NUMA on Hydra within the scope of our testing.

After examining the system specifications outlined in Table 3.1, it appears that Scyrus and Telos have less available bandwidth than ARCHER2. This is an important factor because the SLOPE sparse tiling method achieves optimal results when bandwidth is a limiting factor [8]. As a result, we observe more substantial performance improvements on Scyrus and Telos for smaller meshes, and on ARCHER2 for larger meshes.

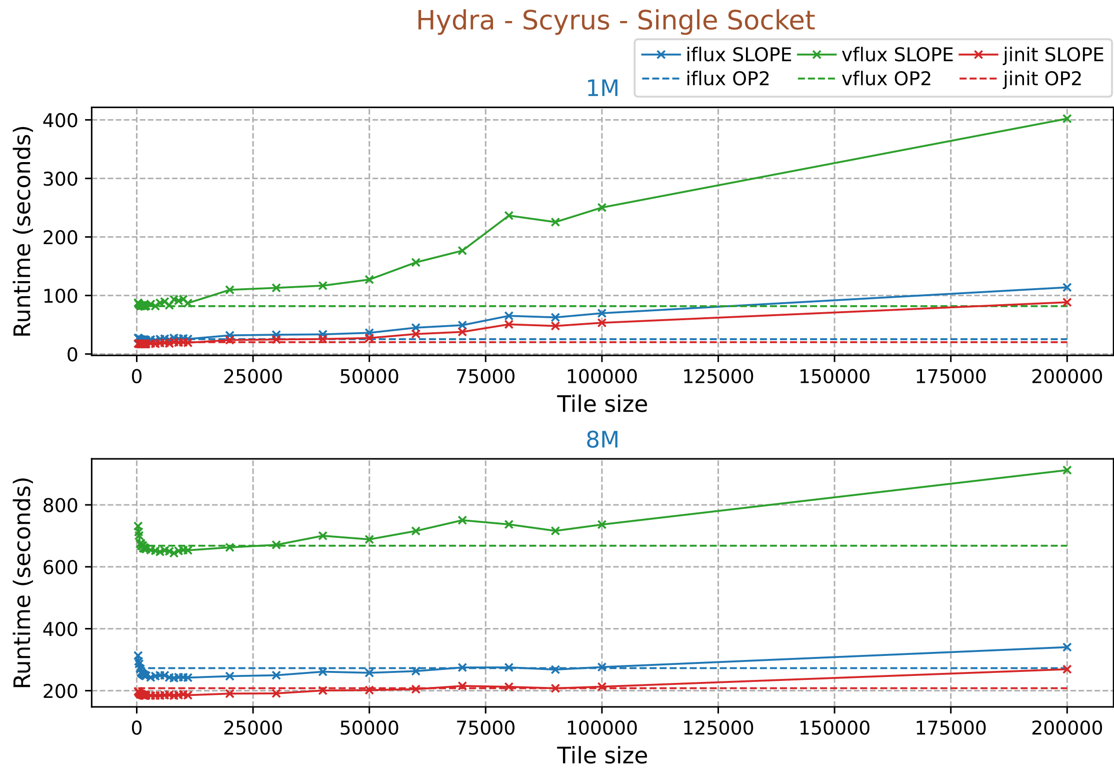


Figure 3.29: Runtime variation of SLOPE Hydra loop-chains with tile sizes on Scyrus (Configurations: single socket, 12 threads)

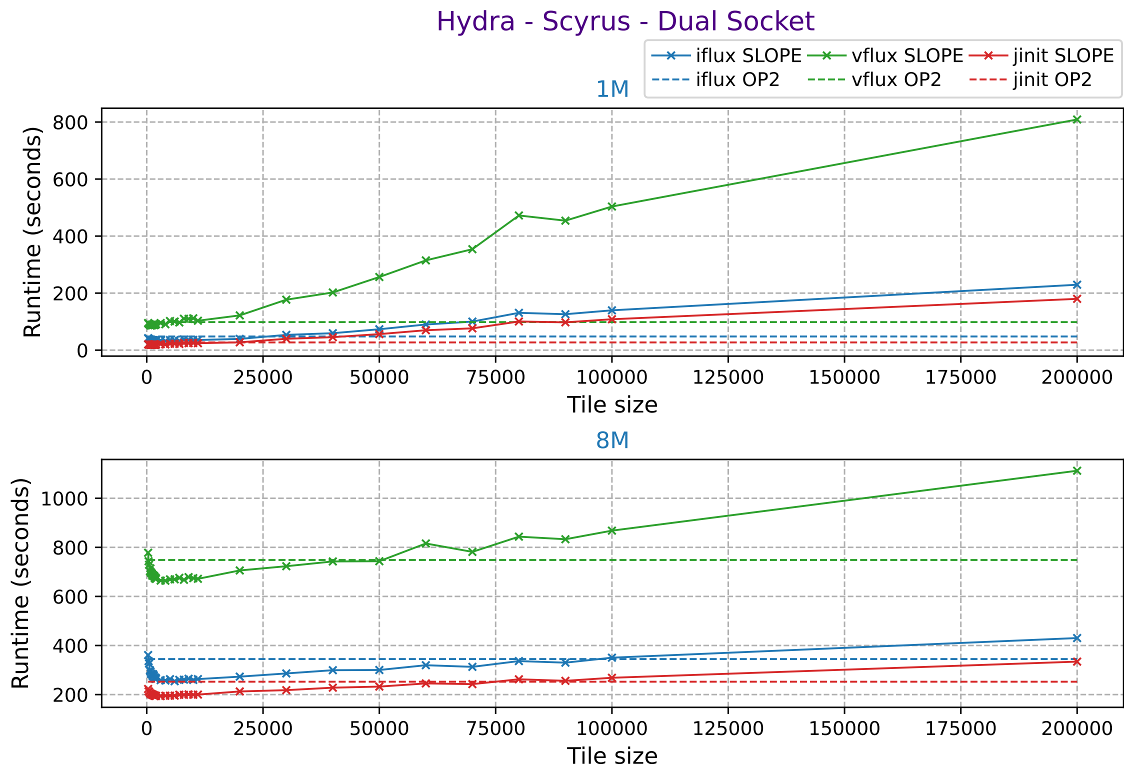


Figure 3.30: Runtime variation of SLOPE Hydra loop-chains with tile sizes on Scyrus (Configurations: dual socket, 24 threads)

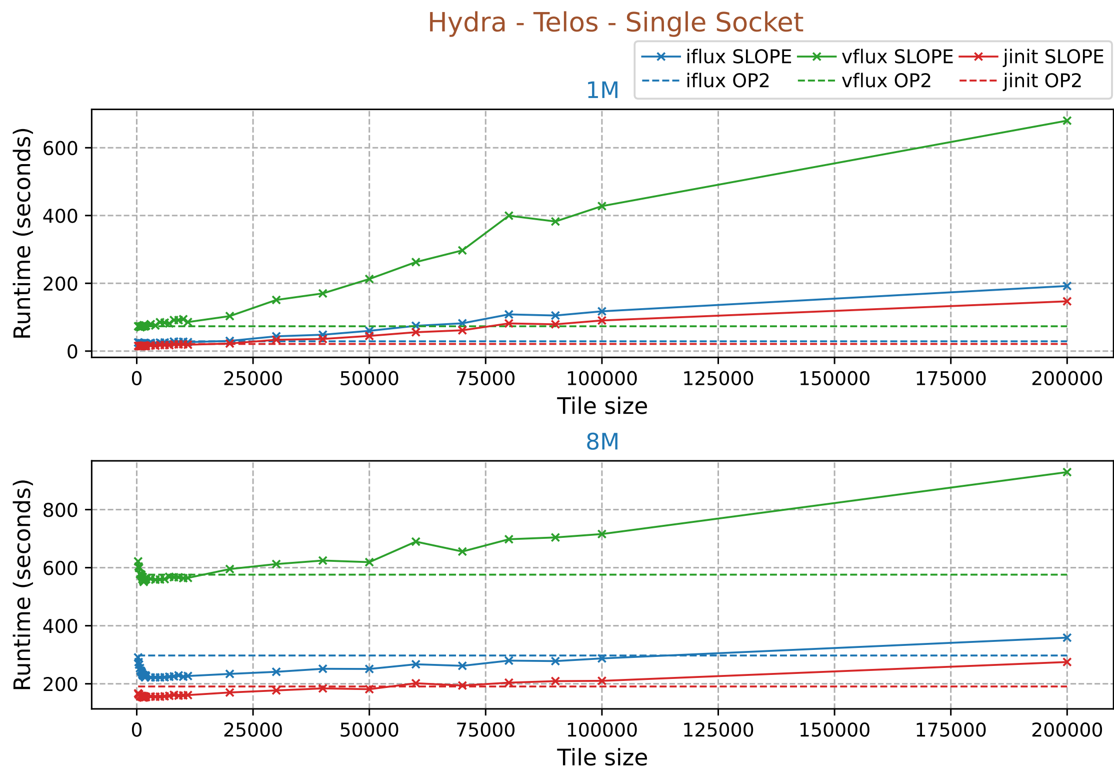


Figure 3.31: Runtime variation of SLOPE Hydra loop-chains with tile sizes on Telos (Configurations: single socket, 24 threads)

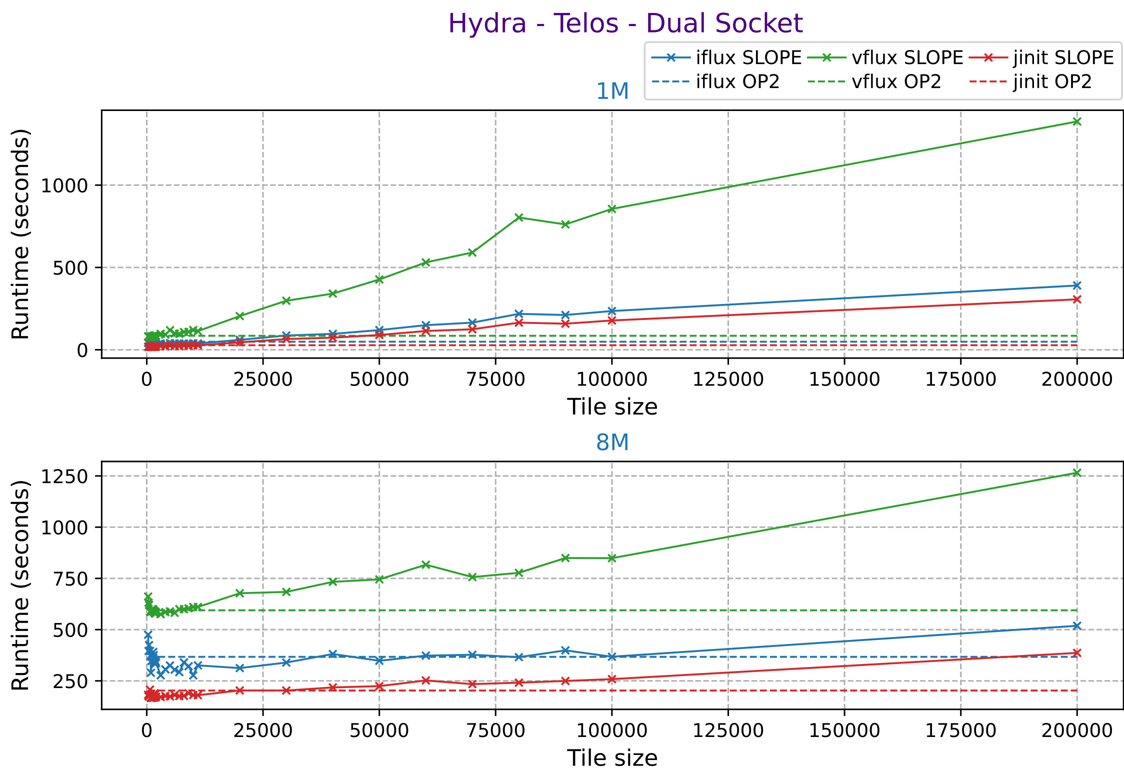


Figure 3.32: Runtime variation of SLOPE Hydra loop-chains with tile sizes on Telos (Configurations: dual socket, 48 threads)

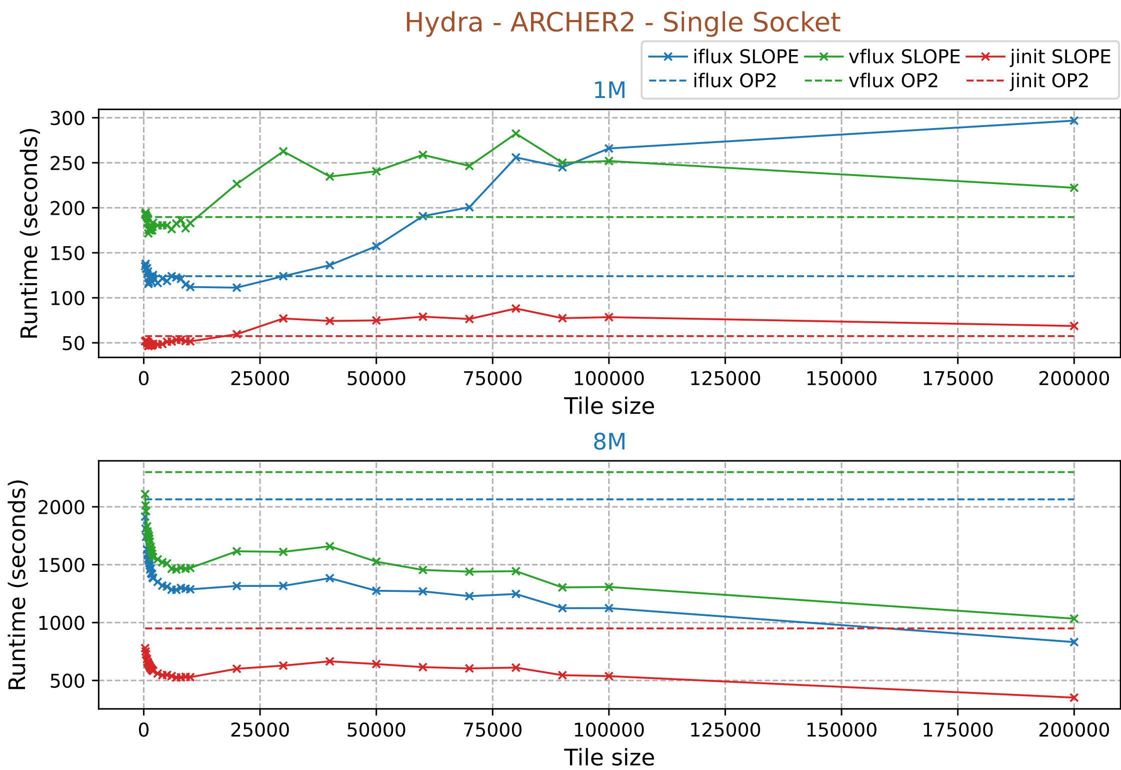


Figure 3.33: Runtime variation of SLOPE Hydra loop-chains with tile sizes on ARCHER2 (Configurations: single socket, 64 threads)

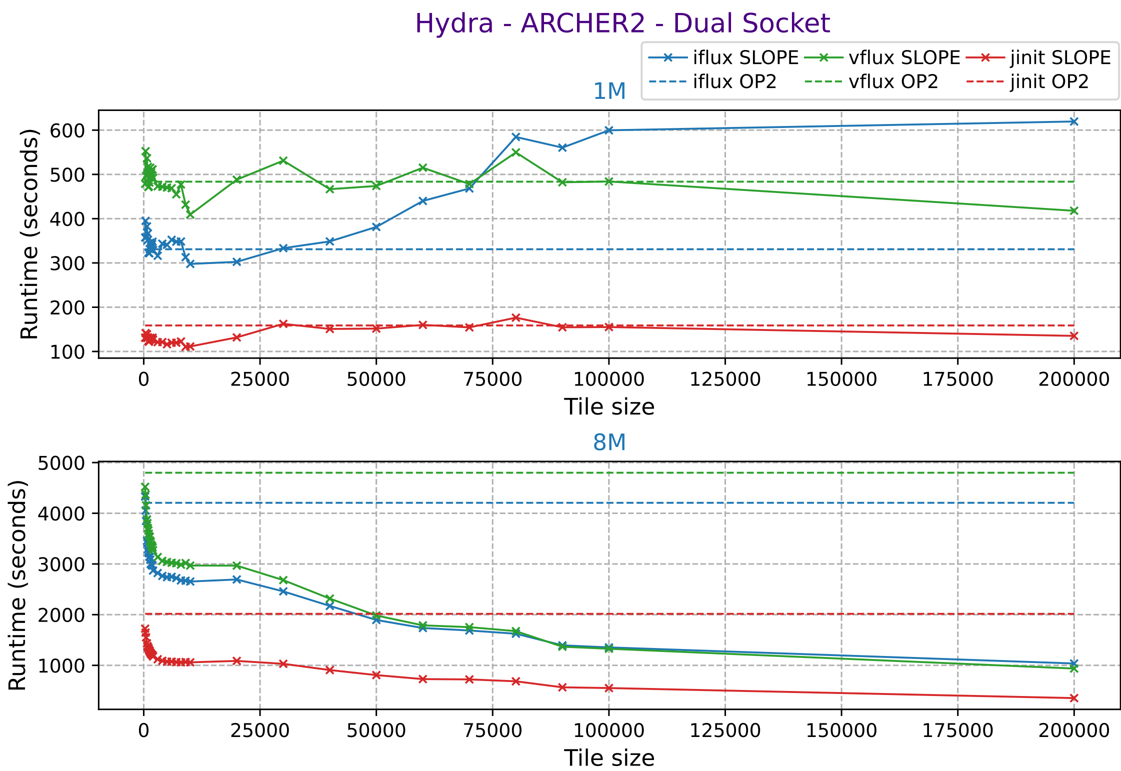


Figure 3.34: Runtime variation of SLOPE Hydra loop-chains with tile sizes on ARCHER2 (Configurations: single socket, 128 threads)

Table 3.9: Best SLOPE Hydra loop-chain performance on Skylake, Telos, and ARCHER2

Dataset	System	Loop-chain	Single Socket		Dual Socket	
			Tile Size	Gain %	Tile Size	Gain %
1M	Scyrus	`iflux`	1700	10.92	4000	33.20
		`vflux`	1700	0.77	800	12.17
		`jinit`	1700	16.41	800	31.42
	Telos	`iflux`	1700	23.74	1400	29.65
		`vflux`	800	3.99	800	8.64
		`jinit`	800	30.47	1200	35.53
	ARCHER2	`iflux`	20000	10.25	10000	10.02
		`vflux`	1000	9.57	10000	15.36
		`jinit`	1000	19.38	9000	30.44
8M	Scyrus	`iflux`	8000	11.73	6000	26.20
		`vflux`	8000	3.57	4000	11.28
		`jinit`	4000	11.50	3000	23.20
	Telos	`iflux`	5000	25.40	10000	24.66
		`vflux`	1500	4.34	3000	3.23
		`jinit`	1500	20.12	900	18.18
	ARCHER2	`iflux`	200000	59.72	200000	75.35
		`vflux`	200000	55.04	200000	80.47
		`jinit`	200000	62.95	200000	82.42

In summary, the analysis highlights the complex relationship between system architecture, mesh size, and optimization techniques. Understanding these dynamics provides valuable insights for future efforts to optimize computational workloads on similar systems.

3.5 Factors Impacting Performance

3.5.1 Tile Size

The tile size refers to the dimensions of the subproblem that a processor or thread works on. The variation in tile sizes can significantly impact performance, affecting the balance between computation and memory access, as well as the potential for parallelism.

To achieve the benefits of cache-blocking tiling, it is crucial to balance computation and communication or data movement between various memory hierarchies of the processor. The tile size should be chosen carefully to ensure an optimal balance between computation and communication. Smaller tile sizes can reduce memory access latency by using smaller data chunks, which may result in a higher proportion of computation overhead relative to memory access time. Meanwhile, larger tile sizes can amortize memory access latency by fetching more data simultaneously, but this can increase memory pressure and result in more cache misses.

Increasing the application's parallelism while reducing the Non-Uniform Memory Access (NUMA) issues can further improve performance. Smaller tile sizes can improve parallelism, allowing more threads or processors to work on different tiles simultaneously. On the other hand, larger tile sizes may reduce parallelism as fewer tiles can fit into the available compute resources.

Optimizing cache usage can also immensely help the application's performance. Smaller tile sizes may lead to better cache efficiency as they are more likely to fit entirely within the cache. On the other hand, larger tile sizes may cause cache thrashing if they exceed the cache capacity, resulting in more cache misses.

Optimizing Tile Size

The right tile size must be chosen to ensure optimal performance when utilizing sparse tiling methods. Factors such as empirical analysis, problem characteristics, and target hardware must be considered when determining the optimal tile size. Empirical analysis and performance profiling should be conducted by experimenting with different tile sizes until the one that provides the right balance between computation and memory access is found. The nature of the problem and data structures should also be considered. Smaller tiles may be better for applications with higher sparsity, while larger tiles can help reduce memory overhead. Finally, the characteristics of the target hardware, such as cache sizes, memory bandwidth, and the number of available cores/threads, should be taken into account. It is important to keep in mind that the optimal tile size may vary across different hardware platforms.

In conclusion, the optimal tile size plays a crucial role in performance when utilizing sparse tiling methods. It influences the balance between computation and communication or memory access, parallelism, and cache efficiency. Careful analysis of the specific application, problem characteristics, and target hardware is necessary to achieve optimal performance gains.

3.5.2 Mesh Partitioner

When working with sparse matrices and tiling strategies in shared-memory parallel computing, the choice of partitioner can greatly affect performance. Two common partitioning approaches used in this context are METIS partitioning and Chunk partitioning.

METIS Partitioner

METIS [124] is a widely used tool for partitioning graphs, especially for dividing large sparse matrices into smaller submatrices or tiles. METIS uses advanced algorithms to create partitions that balance the workload and minimize communication between partitions. This helps reduce data movement between partitions and improve cache and memory efficiency. METIS is particularly useful for tackling large-scale problems and is utilized in many scientific and engineering applications.

Chunk Partitioner

Dividing a matrix into fixed-size blocks or chunks, regardless of its graph structure, is known as Chunk partitioning. Each Chunk is handled separately, which can result in uneven workloads if the matrix's sparsity pattern is highly irregular. Despite this, Chunk partitioning is easy to implement and does not require much computational power. It also provides consistent chunk sizes, which can be useful in certain situations.

The decision to use either METIS partitioning or Chunk partitioning depends on the specific application and the characteristics of the sparse matrices. As explained by Luporini et al. [37], the METIS partitioner may not perform well with SLOPE-based applications due to a rise in TLB (Translation Lookaside Buffer) misses and less effective hardware prefetching caused by more irregular tile expansion. Ultimately, the choice should be based on the performance requirements and characteristics of the specific sparse matrix computation problem. Experimentation and benchmarking with representative datasets and workloads can help determine the most effective partitioning strategy for the application.

3.5.3 Loop Fusion Scheme

Loop fusion is a compiler optimization technique which we have discussed in detail in Section 2.4.1. We tested several loop-fusion schemes for the MG-CFD application when testing with SLOPE. We found that one particular loop fusion scheme yielded better performance gains than the others. This is a crucial factor when chaining a set of loops. Rather than adding all the loops to a single chain, we can consider having multiple loop-chains. Such an approach could help to:

1. Reduce loop overheads such as loop setup and loop termination conditions;
2. Turn data reuse into data locality by chaining loops that access similar datasets together;
3. Improve opportunities for vectorization; and
4. Improve code maintainability and understandability by having smaller and logical loop-chains.

However, the effectiveness of these approaches depends on the nature of the application. Therefore, we should search for avenues that can offer effective loop fusion schemes to improve performance through better data locality.

3.6 Conclusion

We conducted an analysis of the SLOPE library's performance on four different applications: Airfoil, MG-CFD, Volna, and Hydra. Our goal was to identify the benefits of using sparse tiling and determine any possible improvements. We used Intel VTune and Advisor profiling tools to investigate the sparse tiling behavior in Intel processor-based systems. Through roofline analysis, we were able to reason about the positive and negative gains of using sparse tiling. The computing kernels that did not perform well with the SLOPE library are more compute-bound and memory enhancements with tiling are less likely to improve their performance. The better-performing loop fusion schemes also suggest that the performance improvements are from the improved cache performance with tiling. The profiling tool analysis suggests that the data reuse with the adapted loop fusions schemes has turned it into data locality. We conducted single socket and dual socket runs to analyze how NUMA affects the performance of cache-blocking tiling. It was observed that in some cases, NUMA issues can reduce the performance of the cache-blocking tiling version. The extent to which NUMA affects the application's performance depends on the characteristics of the dataset and the behavior of the computing kernel, as well.

To optimize unstructured-mesh-based applications, sparse tiling is employed to increase data locality through the efficient reuse of data in a sequence of loops. The SLOPE library is a powerful tool that facilitates the creation of loop-chains and sparse tiling, thereby improving the performance of these applications. The OP2 library is currently used to generate code for parallel execution on different platforms, and we strive to further enhance the performance by implementing the sparse tiling and loop fusion mechanisms from the SLOPE library, which can reduce communication requirements. Our experiments have revealed that the performance of the SLOPE library is influenced by the nature and size of mesh partitions, as well as the loop fusion scheme utilized. By optimizing these parameters, we can achieve even better performance improvements. Our roofline tests have confirmed that the most significant speedups are attained when kernels are bandwidth-bound.

Chapter 4

Communication-Avoiding Optimizations on Distributed-Memory Systems

In the previous chapter, we examined how cache-blocking tiling and other shared-memory parallelization techniques can improve the performance of unstructured-mesh-based applications. However, shared-memory computing has its limitations. Real-world problems are often too big for a single node to handle and the demand for processing power has increased exponentially in scientific, engineering, and data-intensive fields. There are also challenges associated with handling intricate memory hierarchies, including NUMA issues. Furthermore, considerations related to synchronization, cache coherency, and race conditions present additional complexities. Therefore, to address these challenges and tackle large-scale problems, distributed-memory parallelism is essential. Distributed systems offer several key benefits that are crucial for HPC, as explained below.

One of the primary benefits of distributed parallelism is scalability. As computational demands increase, a distributed architecture allows for seamless expansion to accommodate more extensive simulations, datasets, and computations. This scalability ensures that HPC systems can handle growing workloads without sacrificing performance. Distributed parallelism also provides high computational power, which is crucial for many computationally intensive and time-consuming HPC problems. By spreading the workload across multiple nodes, distributed parallelism harnesses the collective power of numerous processors, accelerating computation and enabling the resolution of complex simulations in feasible timeframes.

Furthermore, distributed parallelism facilitates processing large datasets by dividing them into smaller chunks and distributing them across nodes for concurrent analysis. This feature is especially crucial as the usage of big data continues to grow in various HPC applications. Distributed parallelism empowers researchers to simulate and analyze phenomena with greater fidelity and precision, leading to more accurate models and simu-

lations. It also enables the simulation of complex systems by distributing the calculations across nodes, collectively tackling various aspects of the simulation.

In summary, distributed-memory parallelism is essential to HPC as it empowers researchers, scientists, and engineers to solve complex problems efficiently. It leads to breakthroughs in science and technology, accelerates innovation across a wide range of domains, and enables the exploration of new frontiers.

4.1 Need for Communication-Avoidance

When it comes to distributed-memory systems, there are some challenges that require communication-avoidance techniques as a solution. One of the biggest issues is the latency and overhead involved in inter-node/inter-process communication. This means that when data has to travel across network boundaries, it can cause significant delays and consumes valuable system resources. Limited bandwidth in distributed networks can also lead to congestion and contention, making it difficult to exchange information efficiently. Additionally, too much communication can cause some nodes to become overloaded with data transfers, which can negatively impact the overall system performance. To combat these obstacles, it is important to implement communication-avoidance strategies that minimize communication, optimize resource utilization, and improve the efficiency and scalability of distributed-memory systems.

The OP2 library [6] enables distributed-memory parallelism, utilizing its MPI back-end to execute unstructured-mesh based applications written according to the OP2 API. The current halo structure and loop execution model involve multiple message exchanges among the application's processes to achieve distributed execution. However, communication bandwidth limitations in modern computing systems can hinder performance gains from parallel execution. To fully exploit the computing power of these massively parallel systems, it is crucial to address issues such as multiple synchronizations during loop execution and message exchange delays. Increasing computations instead of communication can be more beneficial in these distributed systems. Thus, adding a communication-avoidance (CA) back-end to the OP2 library has become a pressing need.

4.2 Communication-Avoidance Back-End

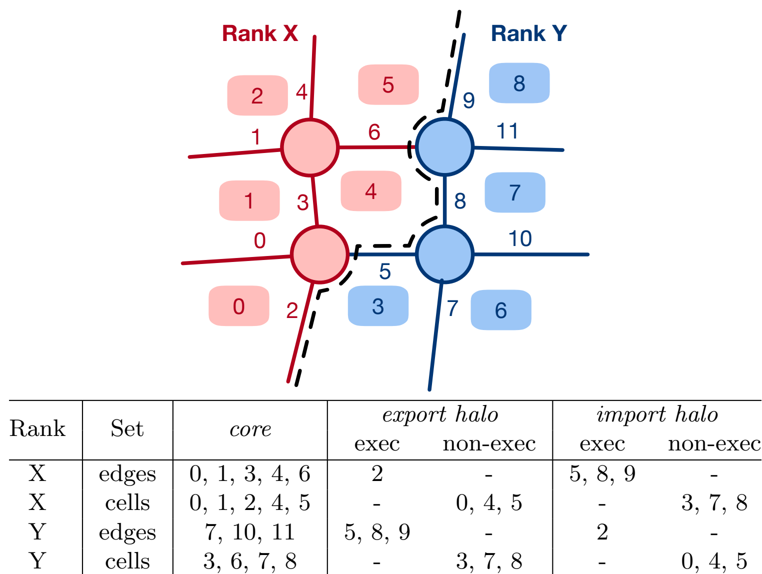
On a distributed-memory parallel level, as we will demonstrate, the idea is to move all communications to the beginning of the loop-chain, eliminating per-loop halo exchanges between neighboring processes, in place of a larger aggregated message. In this case, each mesh partition held by an MPI process can be thought of as a single "tile", which will be executed without halo exchanges within the loop-chain. On a shared-memory multi-threaded parallelization level, the idea would be to select sufficiently sized tiles allowing us to keep a working set of data in the fast cache memory of processors reducing the number of accesses from/to slower main memory per loop. The implementation of

these ideas within OP2, creating a new communication-avoiding back-end and applying it to large-scale unstructured-mesh applications, investigating performance, form the main contribution of this chapter. We will (1) initially codify sparse tiling for distributed-memory execution and then (2) extend it for GPU clusters integrating the back-end to work with CUDA code generated by OP2 in a later chapter.

This section is dedicated to explaining the development of the new communication-avoidance (CA) back-end for the OP2 library and its integration with relevant OP2 library features.

4.2.1 OP2 Distributed-Memory Parallelism

OP2 uses an *owner-compute* model for parallelizing computations on a distributed-memory parallel system [6]. In this model, the unstructured-mesh, defined by mappings and data, is partitioned among a number of processes so that each process *owns* some of the set elements. A process will only perform computations to update elements in their own partitions but will require data from elements in other partitions held in separate processes, specifically at the boundaries of the partitions. Thus, copies of data held in foreign partitions need to be communicated following the standard "halo" exchange mechanisms when using a message passing parallel implementation.



Rank	Set	*core*	*export halo*		*import halo*	
			exec	non-exec	exec	non-exec
X	edges	0, 1, 3, 4, 6	2	-	5, 8, 9	-
X	cells	0, 1, 2, 4, 5	-	0, 4, 5	-	3, 7, 8
Y	edges	7, 10, 11	5, 8, 9	-	2	-
Y	cells	3, 6, 7, 8	-	3, 7, 8	-	0, 4, 5

Figure 4.1: OP2 partitioning over two MPI ranks and resulting halos on each rank [6]

Figure 4.1 illustrates this halo setup and configuration in OP2, where the back-end separates the iteration space such that it is segmented into

(i) *core* – set of iterations that do not need to access halo data

(ii) *export halo* – mesh data to be sent from the local process to some foreign process

(iii) *import halo* – mesh data received from some foreign process

The elements in the import and export halos are further separated into two groups depending on whether redundant computations will be performed on them. For example, in the mesh in Figure 4.1, a loop over edges updating cells will require edges 5,8, and 9 executed on rank X to update cells 4 and 5 (i.e., an import execute halo, *ieh*). However, a loop over edges reading data on cells will require cells 0,4, and 5 to be imported onto rank Y (i.e., an import non-execute halo, *inh*). The *inh* is essentially a read-only halo. These then have a corresponding export execute (*eeh*) and export non-execute (*enh*) halos on each of the local processes.

This section is built on those concepts and primarily compares the implementation of OP2 MPI with the newly developed CA version. In order to explain and formulate the communication-avoidance framework, we utilize the loop-chain abstraction introduced in Section 3.2.4.

4.2.2 Halo Exchanges

Algorithm 4.1: Algorithm to determine a halo exchange and dirty bit management [9]

```
1  foreach indirect op_arg do
2      if ((op_arg.access = OP_READ) || (op_arg.access = OP_RW)) &&
         (op_arg.dat.dirty_bit = 1) then
3          do_halo_exchange(op_arg.dat);
4          clear_dirty_bit(op_arg.dat);
5      end if
6  end foreach

7  loop_size ← set_size;

8  foreach indirect op_arg do
9      if (op_arg.access ≠ OP_READ) then
10         loop_size ← set_size + ieh;
11         break;
12     end if
13 end foreach

14 execute_loop (loop_size);
```

During the execution of an OP2 application under MPI, a data exchange is initiated among the processes depending on the type of the `op_par_loop` and its arguments, `op_arg`s, when a call to `op_par_loop` is made. If the call goes to an *indirect loop* and any of the `op_arg`s refers to an indirect read of a dataset that was previously modified, the relevant *execute* and *non-execute* halos of that dataset must be exchanged before performing calculations on elements not belonging to the *core* region and in the *ieh* region of the relevant dataset. OP2 uses a *dirty_bit* to identify the datasets modified during previous calculations. The halo exchange process and dirty bit management are explained in detail in Algorithm 4.1.

```
1 MPI_Isend(&grp_send_buffer[buf_start],
2   (arg_size - buf_start), MPI_CHAR,
3   exp_common_list->ranks[r], grp_tag, OP_MPI_WORLD,
4   &grp_send_requests[r]);
5 ...
6 MPI_Irecv(&grp_recv_buffer[imp_disp],
7   imp_size, MPI_CHAR,
8   imp_common_list->ranks[i], grp_tag, OP_MPI_WORLD,
9   &grp_recv_requests[i]);
```

Listing 4.1: MPI halo send and recv

When a halo exchange is triggered, it is handled by a call to `op_halo_exchange_chained(int nargs, op_arg *args, int exec_flag)` inside `op_mpi_rt_support.c` using non-blocking MPI sends and receives, as explained in Listing 4.1.

4.2.3 OP2 Loop Execution

In OP2, the mesh data in `op_maps` and `op_dats` are held in 1D arrays with *core*, *export*, and *import* halos structured as illustrated in Figure 4.2(a). This ordering then enables latency hiding where the loop iterations in an `op_par_loop`, corresponding to the *core* can be carried out while halo exchanges are in-flight as these iterations do not access halo data. This latency-hiding algorithm is detailed in Algorithm 4.2. OP2 exchanges the *ieh* and *inh* in separate messages. After all the messages have been sent/received, execution over *execute halos* can be performed. In an `op_par_loop`, halos for an `op_dat` are exchanged only if (1) it is indirectly accessed as a read (`OP_READ`) or a read/write (`OP_RW`) and (2) a preceding loop has modified it, i.e., the halos need updating. A *dirty-bit* is used to keep track of when an `op_dat` is updated (by an `OP_RW`, `OP_WRITE`, or an increment (`OP_INC`)) in a loop, as already explained in Algorithm 4.1.

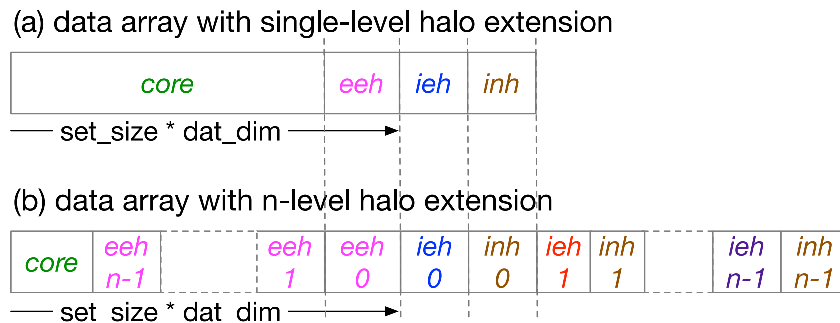


Figure 4.2: `op_dat` and `op_map` data structures with (a) single and (b) multiple halo levels

Algorithm 4.2: OP2 loop execution

Input: `op_par_loop` l over set S_l
Result: Execute loop

1 `MPI_Isend(`*eeh, enh*`);`
2 `MPI_Irecv(`*ieh, inh*`);`

3 **foreach** *iteration* $I \in S_l^c$ **do**
4 │ `execute_iteration(`I`);`
5 **end foreach**

6 `MPI_Wait(`*eeh, enh, ieh, inh*`);`

7 **foreach** *iteration* $I \in S_l^e, S_l^i$ **do**
8 │ `execute_iteration(`I`);`
9 **end foreach**

4.2.4 Multi-Layered Halo Data Structure

The OP2 halos described above essentially maintain the data dependencies required to execute each process's partition independently in parallel. Explicit messages are exchanged to update/sync the halos when carrying out computations in each `op_par_loop`. The dependency neighborhood for a single loop, therefore, can be viewed as a halo layer with a depth of 1 (see Figure 4.3). As shown in [8], in a loop-chain with n loops, syncs per loop can be eliminated if a large dependency neighborhood, maximally a layer with depth of n can be communicated at the start of the loop-chain and computed over, redundantly to update the mesh elements to satisfy the dependencies, that would have otherwise been updated as a halo exchange. Thus for the loop-chain with 2 loops detailed in Listing 3.2, a halo depth of 2 needs to be maintained as illustrated in Figure 4.4.

The maximum depth of n is required only when in a loop-chain, $L_0, \ldots, L_{n-1}$, each loop L_i updates an `op_dat` d and the next loop L_{i+1} read or read/writes to d. This leads to iteration spaces that decrease in size for each loop in the loop-chain. Specifically, to compute I iterations of the last loop in the chain, L_{n-1}, the loops $L_{n-1}, L_{n-2}, \ldots, L_0$ should be iterating over I plus halo depths of $1, 2, \ldots, n$ respectively.

4.2.5 Loop-chains with CA

Algorithm 4.3 details the steps to follow until the loop-chain is executed with the CA framework. In the main inspection/setup phase, `halo_exch_dats` identifies the `op_dat`s for halo exchange based on their access modes and dirty-bit values. Then, `calc_halo_layers` compute the number of halo layers required for each loop in the loop-chain, $\mathbb{L}$. The analysis is detailed in Algorithm 4.4. Given a loop-chain, Algorithm 4.4 calculates, the minimum halo extension required for each `op_dat` in each loop according to its individual data access patterns. Then, the maximum halo extension required for a loop is obtained

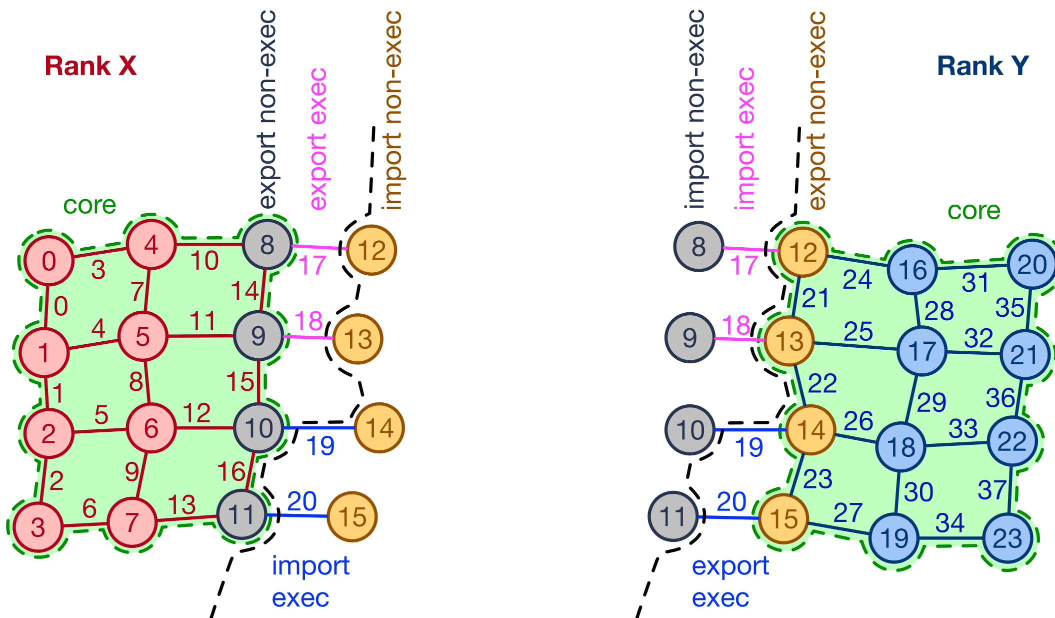


Figure 4.3: Halo layer with depth 1

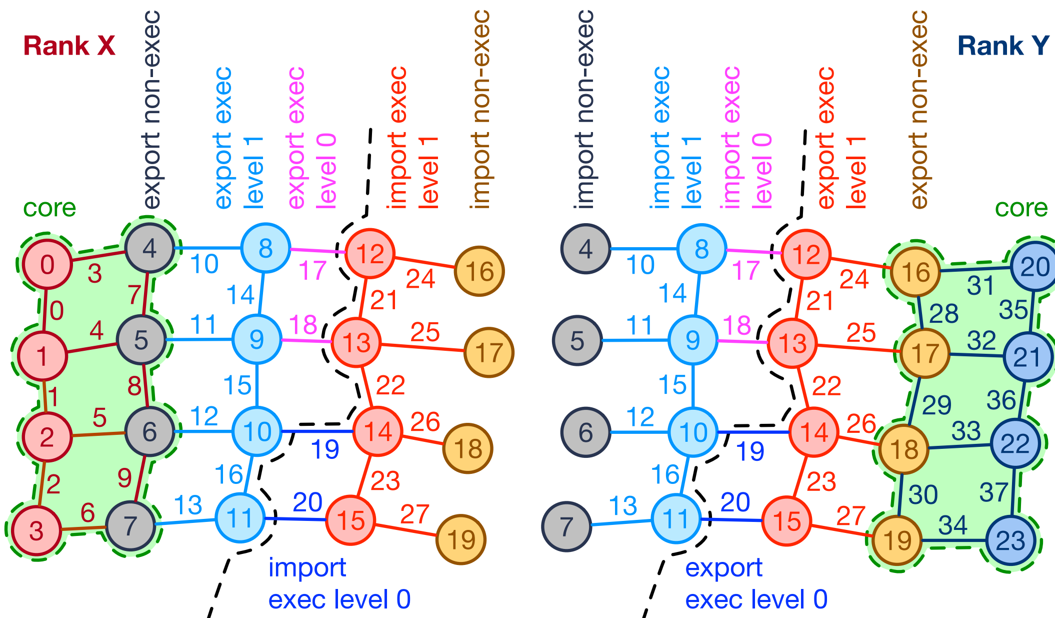


Figure 4.4: Halo layer with depth 2

based on the halo extensions calculated for each individual `op_dat` in the loop, finally making the loop's halo extension effective for all the `op_dats` in the loop. `calc_iters` (in Algorithm 4.3) computes the iteration counts within the *core* and halo layers, while `restructure_elements` separates each halo layer into *core* and *eeh* as illustrated in Figure 4.2(b) and renumbers mappings of *core*, *eeh*, and *enh* accordingly. OP2's halo data structure was extended to support this multi-layered halo setup. The loop-chain executes *core* followed by eeh_{n-1} to eeh_0, then all the import halos ieh_0 to ieh_{n-1} (See lines 8-18 in Algorithm 4.3.).

Algorithm 4.3: Loop-chain execution with CA

```
    Input: Loop-chain, 𝕃 = {L_0, ..., L_{n-1}}, op_dats, 𝔻 used in 𝕃
    Result: Execute loop-chain with CA

    // Find op_dats requiring halo syncs:  𝔻^h ⊆ 𝔻
 1  𝔻^h ← halo_exch_dats(𝔻, < M, mode >, 𝕃);

    // Compute #halo layers required for each loop in 𝕃
 2  ℍ𝕃_l ← calc_halo_layers(𝕃, 𝔻^h);

    // Find core, exec, & non-exec halo #iters for loops in 𝕃
 3  𝕊_l^c, 𝕊_l^h, 𝕊_l^n ← calc_iters(𝕊_l, ℍ𝕃_l);

    // Rearrange and renumber multi-layered core, eeh, & enh for each
       op_dat d ∈ 𝔻^h
 4  𝕊^eeh,𝕊^enh ← restructure_elements(𝔻^h, ℍ𝕃_l, 𝕊_l^c, 𝕊_l^h, 𝕊_l^n);

 5  m^{eeh+enh} ← create_grouped_msg(𝔻^h, 𝕊^eeh,𝕊^enh);

 6  MPI_Isend(m^{eeh+enh});
 7  MPI_Irecv(m^{ieh+inh});

 8  foreach loop l ∈ 𝕃 do
 9  |   foreach iteration I ∈ S_l^c do
10  |   |   execute_iteration(I);
11  |   end foreach
12  end foreach

13  MPI_Wait(m^{eeh+enh}, m^{ieh+inh});

14  foreach loop l ∈ 𝕃 do
15  |   foreach iteration I ∈ 𝕊_l^h do
16  |   |   execute_iteration(I);
17  |   end foreach
18  end foreach
```

These steps will be elaborated below in detail to have a better idea about the communication-avoidance framework.

4.2.6 Inspection Phase

Before the loop-chain execution, the OP2 library's communication-avoidance back-end needs to complete several stages known as the *inspection* phase. Some of these stages are shared with the existing MPI back-end, which is explained in the 'OP2 Developers Guide - Distributed-Memory (MPI) Parallelisation' by Mudalige et al. [9]. However, we have improved it to enable our CA back-end to create multiple halo layers. In this section, we will discuss the main stages of the inspection phase in the communication-avoidance back-end which are detailed in lines 1-4 in Algorithm 4.3.

```
op_dat dat0 = op_decl_dat(nodes, 2, "double", dat0_val, "dat0");
op_dat dat1 = op_decl_dat(nodes, 2, "double", dat1_val, "dat1");

// loop over edges, updating node values
op_par_loop(L0_kernel, "L0", edges,
  op_arg_dat(dat0, 0, e2n, 2, "double", OP_INC),
  op_arg_dat(dat0, 1, e2n, 2, "double", OP_INC),
  op_arg_dat(dat1, 0, e2n, 2, "double", OP_READ),
  op_arg_dat(dat1, 1, e2n, 2, "double", OP_READ));

// loop over edges, updating node values
op_par_loop(L1_kernel, "L1", edges,
  op_arg_dat(dat0, 0, e2n, 2, "double", OP_READ),
  op_arg_dat(dat0, 1, e2n, 2, "double", OP_READ),
  op_arg_dat(dat1, 0, e2n, 2, "double", OP_WRITE),
  op_arg_dat(dat1, 1, e2n, 2, "double", OP_WRITE));

// loop over cells, updating node values
op_par_loop(L2_kernel, "L2", cells,
  op_arg_dat(dat0, 0, c2n, 3, "double", OP_WRITE),
  op_arg_dat(dat0, 1, c2n, 3, "double", OP_WRITE),
  op_arg_dat(dat0, 2, c2n, 3, "double", OP_WRITE),
  op_arg_dat(dat1, 0, c2n, 3, "double", OP_READ),
  op_arg_dat(dat1, 1, c2n, 3, "double", OP_READ),
  op_arg_dat(dat1, 2, c2n, 3, "double", OP_READ));
```

Listing 4.2: Loop-chain ($\mathbb{L}$) example written using OP2 API for CA

1. **Find `op_dats` requiring halo synchs in $\mathbb{L}$**

   By analyzing a loop-chain, we can distinguish the data access patterns of each `op_dat` separately, allowing us to identify the `op_dats` that require a halo exchange.

   Requirement for halo exchange: Indirect read access (`OP_READ` or `OP_RW`) to a previously modified `op_dat` (with `OP_WRITE`, `OP_INC`, or `OP_RW`) inside a loop.

   In the example provided in Listing 4.2, `dat0` is modified (`OP_INC`) in `L0` and indirectly read (`OP_READ`) in `L1` using *edges-to-nodes* (`e2n`) mappings. This requires a halo exchange before executing `L1`. Similarly, `dat1` is modified (`OP_WRITE`) in `L1` and indirectly read (`OP_READ`) in `L2` using *cells-to-nodes* (`c2n`) mappings. This also requires a halo exchange before executing `L2`.

   Therefore, in this example, two halo exchanges need to be triggered when executing the code using standard loop execution. This leads to multiple synchronization points in the code, which negatively affects the loop execution performance. However, for the CA back-end, we identify these locations beforehand and avoid multiple synchronization points during the execution of a sequence of loops.

2. **Compute #halo layers required for each loop in $\mathbb{L}$**

   Once the `op_dats` that require a halo exchange have been identified, it is essential to determine the minimum halo extension needed for each `op_dat` in each loop. This

Algorithm 4.4: `calc_halo_layers`

Input: Loop-chain, $\mathbb{L} = \{L_{n-1}, \ldots, L_0\}$, `op_dats` requiring halo syncs, $\mathbb{D}^h$, their access descriptors, <M, `mode`>, loops where `op_dats` ($\in \mathbb{D}^h$) are accessed, $\mathbb{A}_{\mathbb{D}^h}$

Output: Halo extensions, $\mathbb{HE}$ for loops in loop-chain, $\mathbb{L}$

```
   // Calculate halo extension for individual op_dats in loops
1  foreach op_dat D ∈ D^h do
2  |   halo_ext ← 0;
3  |   ind_rd ← false // True for indirect read
4  |   foreach loop l ∈ L do // Iterate from loop n-1 to 0
5  |   |   HE_{D_l} ← 1;
6  |   |   if D_l<M, mode> ≠ NULL && l ∈ A_{D_l} then
7  |   |   |   if ind_rd && (mode = OP_WR || mode = OP_INC || mode = OP_RW) then
8  |   |   |   |   HE_{D_l} ← halo_ext + 1;
9  |   |   |   |   halo_ext ← 0;
10 |   |   |   |   ind_rd ← false;
11 |   |   |   |   continue;
12 |   |   |   end if
13 |   |   |   if M ≠ ID && (mode = OP_RD || mode = OP_RW) then
14 |   |   |   |   halo_ext ← halo_ext + 1;
15 |   |   |   |   HE_{D_l} ← halo_ext;
16 |   |   |   |   ind_rd ← true;
17 |   |   |   |   continue;
18 |   |   |   end if
19 |   |   |   if M = ID && (mode = OP_RD || mode = OP_RW) then
20 |   |   |   |   HE_{D_l} ← 1;
21 |   |   |   |   halo_ext ← 0;
22 |   |   |   |   ind_rd ← false;
23 |   |   |   |   continue;
24 |   |   |   end if
25 |   |   end if
26 |   end foreach
27 end foreach
   // Calculate effective halo extension for loops
28 foreach loop l ∈ L do
29 |   HE_l = max(HE_{D_l^h})
30 end foreach
```

calculation is based on the unique data access patterns of each `op_dat`. After the minimum halo extension has been determined for each `op_dat`, it is necessary to determine the maximum halo extension required for the loop. This maximum halo extension is based on the halo extensions calculated for each individual `op_dat` in the loop. Ultimately, the halo extension of the loop is made effective for all `op_dats`

```
1  typedef struct op_halo_info_core {
2    int *nhalos;            //array of number of halos, nhalos
3    int *nhalos_bits;       //bitmask to track required halos
4    int *nhalos_calc_bits;  //bitmask to track required halos for calculation
5    int max_nhalos;         //max number of halos
6    int max_calc_nhalos;    //max number of halos used for calculation
7    int nhalos_count;       //number of elements in nhalos array
8    int nhalos_cap;         //capacity of nhalos array
9  } op_halo_info_core;
10 typedef op_halo_info_core* op_halo_info;
```

Listing 4.3: `op_halo_info_core` data structure

in the loop. This process is explained in Algorithm 4.4, and the data structure, `op_halo_info_core` as in Listing 4.3 is used to keep track of halo information.

According to the algorithm in `calc_halo_layers`, it appears that `dat0` will require a halo extension of 2 in `L0` and 1 in `L1`, while `dat1` will need a halo extension of 2 in `L1` and 1 in `L2`. Therefore, the effective halo extensions for the loops will be 2 in `L0`, 2 in `L1`, and 1 in `L2`.

It is important to note that in a different loop-chain, `dat0` and `dat1` may require a different maximum halo extension than the calculated halo extension of 2. The library and data structure provided in Listing 4.3 supports halo extensions of more than 2. We have tested the library for a halo extension of 32 in our experiments.

3. **Find core, exec, & non-exec halo #iters for loops in $\mathbb{L}$**

Once the loop-chain information and their halo extensions have been finalized, the sizes of the *core*, *exec*, and *non-exec* components are calculated. To create the *import* and *export* halo lists, we utilize the `halo_list_core` data structure outlined in Listing 4.4. We have improved the data structure to support information on multiple halo layers.

The steps required for this main stage are explained below. We are utilizing the current OP2 MPI back-end as our reference, so these steps have some common functionalities with what is explained in [9]. However, there are some differences: these steps have been enhanced to support multiple halo layers and they are called in an outer loop until we generate the required number of halos. Additionally, there are new steps added to facilitate our multiple halo generation process. The enhancements and new steps are easily identifiable by comparing them to 'OP2 Developers Guide - Distributed-Memory (MPI) Parallelisation' by Mudalige et al. [9].

(a) **Generate export lists for execute set elements**

The program will iterate through all the set elements allocated to each MPI process. It will check if any mappings related to those set elements refer to set elements of the to-set of the mappings that belong to foreign MPI processes. For instance, if edge 17 is made up of nodes 8 and 12, and node 12 belongs

```
typedef struct {
  op_set set;       // set related to this list
  int size;         // number of elements in this list
  int *ranks;       // MPI ranks to be exported to or imported from
  int ranks_size; // number of MPI neighbors to be exported to/imported from
  int *disps;       // displacement of the starting point of each rank's
                    // element list
  int *sizes;       // number of elements exported to/imported from each ranks
  int *list;        // the list of all elements

  // added to support multiple halo levels
  int num_levels;             // number of halo levels
  int *ranks_disps_by_level; // displacement of each rank's element list
                              // inside a halo level
  int *disps_by_level;        // displacement of each halo level's element list
  int *level_sizes;           // total halo level sizes
} halo_list_core;
typedef halo_list_core *halo_list;

// halo lists to support standard OP2 halos
halo_list *OP_export_exec_list; // eeh list
halo_list *OP_import_exec_list; // ieh list

halo_list *OP_import_nonexec_list; // inh list
halo_list *OP_export_nonexec_list; // enh list

// halo lists to support extra halo layers for communication-avoidance
halo_list **OP_aug_export_exec_lists; // eeh lists
halo_list **OP_aug_import_exec_lists; // ieh lists

halo_list **OP_aug_export_nonexec_lists; // inh lists
halo_list **OP_aug_import_nonexec_lists; // enh lists
```

Listing 4.4: `halo_list_core` data structure and its use in the CA back-end

to a foreign MPI process, edge 17 will be identified as a mesh element to be exported. These identified set elements will be added to an export execute halo (*eeh*) list. The halo list will also keep track of the ranks to which the set element should be exported. Before adding an element to the *eeh* list, it will verify that the same element was not previously exported to the same MPI process for an inner halo layer.

(b) **Generate import lists for mappings and execute sets**

Each MPI process exchanges the *eeh* lists with the neighboring MPI processes. They will create the relevant import execute halo (*ieh*) lists to use in exchanging mappings and data elements.

(c) **Exchange mapping entries using the import/export execute halo lists**

All the mappings related to the from-set elements in the *eeh* layers will be exchanged with the relevant MPI processes using the created *eeh* and *ieh* lists. The received mapping entries will be appended to the existing `map` array in the `op_map` data structure in the process.

(d) **Generate import lists for mappings and non-execute sets**

Each MPI process will iterate through all its mappings, including the mappings of the imported halo layers. It will check for any unavailable elements referred to by the mappings (i.e., elements in the to-set of the mappings) that are not in the *ieh* lists. These unavailable elements will be added to an *inh* list which will be sorted according to the local index of the foreign process.

(e) **Generate export lists for mappings and non-execute sets**

Each MPI process will then exchange its *import non-execute* halos with its neighbors. It will use that to create *export non-execute* halos. After this stage, all the halo lists will be completed and they will be used to exchange data among the MPI processes.

(f) **Exchange data defined on execute set elements using the import/-export lists**

The data defined on execute set elements will be exchanged with the processes' neighbors by using the created *eeh* and *ieh* lists. This data will be appended to the `data` array of the `op_dat` data structure of the receiving process.

(g) **Exchange data defined on non-execute set elements using the import/export lists**

The data defined on non-execute set elements will be exchanged with the processes' neighbors by using the created *enh* and *inh* lists. This data will also be appended to the `data` array of the `op_dat` data structure of the receiving process.

(h) **Update set attributes for the generated halo layer**

After creating the halo layer, the attributes in the `op_set` will be updated with *core* and halo sizes (*execute* and *non-execute*). These sizes are utilized in halo exchanges and further halo layer generation.

(i) **Create and exchange augmented partition ranges**

After creating a halo layer, the partition boundary will expand and the next halo layers will have to be created based on the latest expanded boundary of the partition with imported halos. To facilitate this, the new partition range of the process, which is now augmented, will be exchanged among all the available processes of the system to support further halo expansion if there is any.

(j) **Renumber mapping tables**

Having exchanged all the extra mapping information of the added halo layers, they are renumbered according to the local set element arrangement as illustrated in Figure 4.2 for single and multiple halos.

(k) **Create MPI send buffers**

At this stage, data buffers to exchange halo information of `op_dat`s during the application execution will be created. A common buffer will also be created considering the maximum possible size for the grouped halo exchanges.

4. **Rearrange and renumber multi-layered *core*, *eeh*, & *enh* for each** `op_dat`

(a) **Separate core elements**

Once all the *import* and *export* halo lists, both *execute* and *non-execute*, have been created, each process can determine the number of computations it can perform without waiting for information from other processes. These *core* computations can be carried out while the halo information is still in-flight. To enable computation and communication to overlap, the data elements will be rearranged or the core elements will be separated, as shown in Figure 4.2.

(b) **Save the original set element indices**

Since the set elements are rearranged, it is necessary to keep track of the original data arrangement when formulating the final answer for the application. Saving the original set element indices supports that purpose.

(c) **Clean up and compute rough halo size numbers**

Temporary data arrays are released after these steps. Only the halo lists (*import/export* and *execute/non-execute*), which are used during calculations of the application, are kept in memory. Especially when it comes to multiple halo layers, information on the intermediate halo layers which are used to generate the final required halo layer, will only be kept if they are used in the calculations.

4.2.7 Execution Phase

In our applications, we only create loop-chains for loops that result in multiple halo exchanges during execution. The remaining loops are executed using the standard OP2 loop execution mechanism. With the newly developed CA version, both loop-chains and individual loops can be executed without any effect on their performance.

This section will outline the primary steps involved in executing a loop-chain using the CA back-end, utilizing the information on extra halos generated during the inspection phase. These steps are detailed in lines 5-18 in Algorithm 4.3.

1. **Create grouped halo message**

In the communication-avoidance framework, we aim to minimize the need for multiple synchronization points in a loop-chain. To achieve this, we identify and exchange the necessary halos for distributed loop execution before moving on to the loop-chain execution. This eliminates the need to exchange halos between each loop. During this process, we must also add any extra halos identified during the inspection phase.

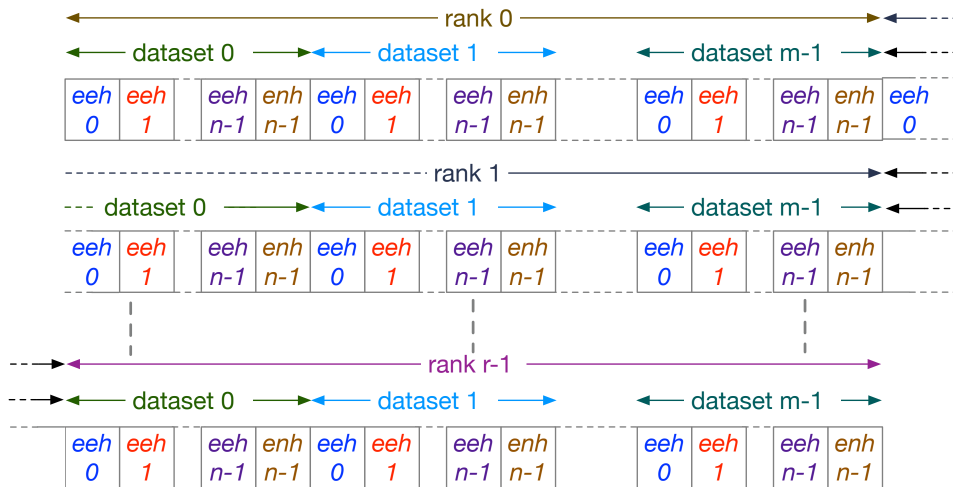


Figure 4.5: Grouped halo array

To further reduce the number of messages exchanged for the entire loop-chain, we organize the halos from multiple loops and `op_dats` into the same data structure. This allows us to construct a single message per MPI rank to be sent/received (lines 5-7, 13 in Algorithm 4.3), as shown in Figure 4.5.

Algorithm 4.3 also does latency hiding. This means that the iterations of each loop in the chain that involve the *core* are executed first while the halos are in-flight. Once the send/receive operations are completed, the iterations related to *non-core* elements, including the additional iterations for the extra halos, are executed.

2. **Core iteration execution**

   During the inspection phase, we separate the *core* iterations as outlined in Step 4a of Section 4.2.6. These iterations can be executed without the halo information from neighboring processes. To save the time spent on communicating halos, we start executing the *core* iterations while the halos are in-flight. The *core* iterations consist of iterations from all the loops in the loop-chain, which are executed in the loop-order.

   For example, in the loop-chain presented in Listing 4.2, the *core* sections of $L0$, $L1$, and $L2$ are executed in the loop order before their *non-core* iterations. The *core* sizes for these loops generally decrease gradually from the first loop to the last loop. In the example given in Listing 4.2, the *core* sizes are $S^c_{L0} > S^c_{L1} > S^c_{L2}$.

3. **Message reception and unpacking**

   Line 13 of Algorithm 4.3 includes the `MPI_Wait` command, which indicates that the message exchange between processes is complete. However, in the CA version, the message content cannot be directly received by the data buffer of the `op_dat` as in the OP2 standard halo message exchange. This is because we have added halos of

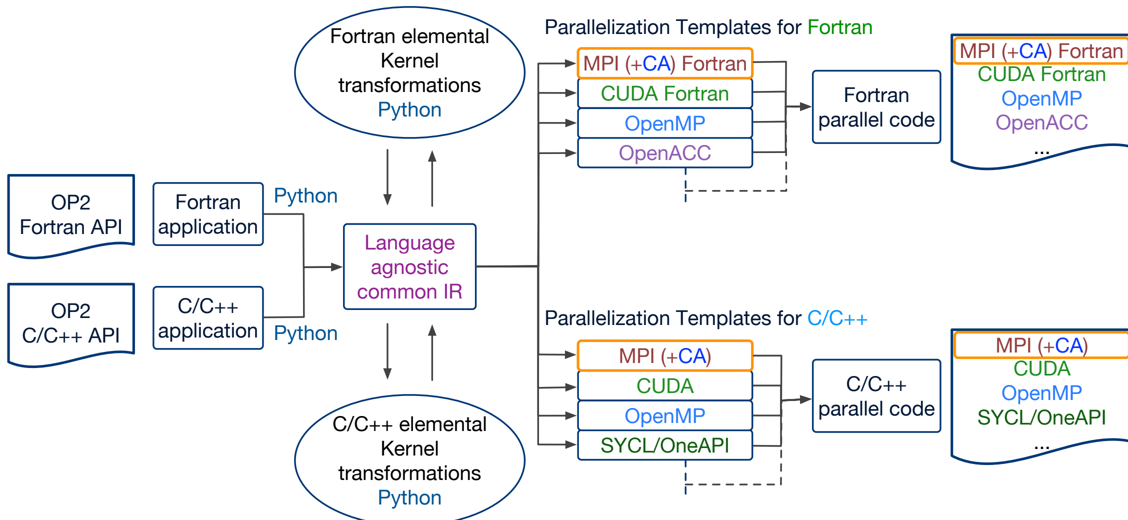


Figure 4.6: OP2 code generation/translation process

multiple `op_dats` to a single message. To continue with further calculations, this grouped halo message must be properly unpacked for relevant `op_dat data` buffers.

4. **Non-core iteration execution**

   Once the halo exchange is finished, *non-core* iterations are executed. These involve executing the *export-exec* halos (*eeh*) and *import-exec* halos (*ieh*). If the loop has n halos, the execution will occur from eeh_{n-1} to eeh_0 and from ieh_0 to ieh_{n-1}.

4.2.8 Automatic Code Generation

The OP2 framework is designed to generate optimized code for various platforms with different programming models, using a highly effective code generation process as shown in Figure 4.6. It is especially useful when dealing with complex mathematical computations on unstructured-mesh data. OP2 supports two major programming languages, namely Fortran and C/C++. The biggest advantage of OP2 is its ability to abstract parallel programming complexities and platform-specific optimizations. As a result, scientists and researchers can focus on their scientific objectives.

The OP2 code generation process can handle code written in either Fortran or C/C++, converting it into a common intermediate representation (IR). This IR acts as a connector, allowing high-level, easy-to-read code to be turned into low-level, machine-executable code that can be optimized for different hardware architectures. The IR includes loop descriptions and elemental kernels that are shared by both supported languages. A major benefit of OP2 code generation is that the code is still understandable to humans, meaning developers and scientists can easily view and comprehend it. This human-readable IR is beneficial when trying to understand platform-specific optimizations applied during the code generation process.

The OP2 code generation process allows specific scientific simulations to be optimized and parallelized using annotations and directives. These directives determine whether computations should be performed on CPUs or GPUs and which parallel programming models to use, such as CUDA, OpenMP, OpenACC, SYCL/OneAPI, and HIP. The newest addition to the code generation process is the communication-avoidance (CA) enhancement generation which we have described in this chapter. To further improve efficiency, OP2 offers parallelization templates for both Fortran and C/C++, which generate optimized code for various platforms to utilize their massive parallelism. These enhancements are applied to computational kernels to ensure that they run efficiently on designated hardware. The code generation process utilizes the Python language for code analysis, transformations, and generating optimizations. This process has no opaque-/black box layers and is easy to understand.

4.2.9 OP2-CA Integration

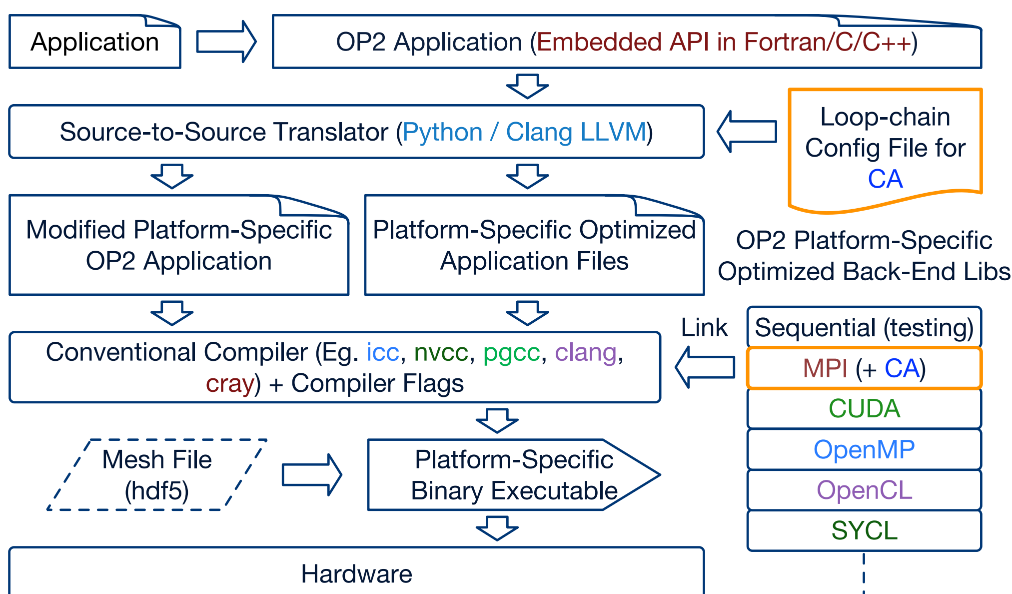


Figure 4.7: OP2 code generation with CA

After developing the CA back-end we added automatic code generation support for communication-avoidance optimizations to the OP2 library. These optimizations have been integrated into OP2 in a way that enables applications developed with OP2 API to use the new features without changing the high-level source. The only change to OP2's original architecture explained in Section 2.6.6 is the addition of a *loop-chain configuration file* that specifies loop names, loop count, and maximum halo extension of loops to be chained within the application. The OP2 code-generator has been modified to generate the loop-chain execution template in Algorithm 4.3 for the selected loops. The generated code for CA optimizations is human-readable and can be compiled with a traditional compiler that links the new CA back-end library. This results in an executable that can run on the chosen hardware.

4.3 Analytic Model for Loop-chain Performance

In the previous section, we introduced a new communication-avoidance back-end for the OP2 library. Our next step is to determine if a loop-chain will benefit from this enhancement. To do so, we can create an analytic performance model to analyze the runtimes of OP2 loops and compare them to equivalent loop-chains executed with the communication-avoiding setup. This can be achieved by parameterizing loop characteristics, communication patterns, and machine properties.

4.3.1 OP2 Loop Execution

Considering the execution of a single OP2 loop, l as detailed in Algorithm 4.2, its runtime can be modeled by (i) the time taken for computing S_l^c (core) and $S_l^e + S_l^i$ (execute halo) number of iterations, (ii) time taken to sync/communicate halos, minus any overlap of computation and communication. If we note $S_l^e + S_l^i = S_l^h = S_l^1$ to indicate that this is execution over a single halo layer, then the time taken by an OP2 loop is given by:

$$T_{op2,l} = MAX \left[\ g_l S_l^c, \ 2 d_l p_l (L + m_l^1 / B) \ \right] + g_l S_l^1 \tag{4.1}$$

Here, g_l is the *compute* time for one iteration of the loop body, d_l is the number of `op_dats` with halos to be synced, p_l is the maximum number of neighboring processes to communicate halos with per MPI process. m_l^1 is the maximum message size (in bytes) sent to a neighbor for either *eeh* or *enh* halos. The superscript 1 indicates the number of halo layers exchanged (1 is the default for OP2 loops). The multiplier 2 accounts for the time for sending a separate message for *eeh* and *enh*. The maximum message size and number of neighbors per MPI process are only known at runtime after the mesh partitioning [6]. The maximum is used for each of the components above to model the critical path of the runtime, where for example we assume that the halo exchange cost between processes only completes as the slowest exchange between a pair of processes. L and B are the latency and bandwidth of the network respectively. Then, the time taken for n number of OP2 loops in a loop-chain, $\mathbb{L}$ is simply the sum of the time taken by the individual loops:

$$T_{op2,\mathbb{L}} = \sum_{l=0}^{n-1} T_{op2,l} \tag{4.2}$$

4.3.2 CA Loop-chain Execution

When the loop-chain is executed with the communication-avoidance setup using the multi-layered halo data structures, a single grouped halo message is exchanged per neighboring process. This and the execution steps in Algorithm 4.3 lead to a total runtime of the full loop-chain:

$$T_{ca,\mathbb{L}} = MAX \left[\sum_{l=0}^{n-1} g_l S_l^c, \ p(L + m^r / B + c) \right] + \sum_{l=0}^{n-1} g_l S_l^h \tag{4.3}$$

As detailed in Algorithm 4.3, S_l^h includes iterations from the execute halos of multiple levels for each loop. The message size, m^r is the maximum grouped message size sent to each neighbor, which combines both the *eeh* and *enh* into a single message, as discussed before. r (where $r \leq n$) indicates the maximum number of halo layers required to carry out the CA algorithm for the loop-chain. Thus, m^r is given by:

$$m^r = \sum_{l=0}^{n-1} \Big(\sum_{d=0}^{d_l-1} (S_d^{eeh,h_l} + S_d^{enh,h_l}) \times \delta \Big) \tag{4.4}$$

Here, d_l is the number of halo syncing dats in loop l, h_l is the halo extension for loop l, S_d^{eeh,h_l} - *eeh* size up to level h_l of the set on which the dataset d is defined, S_d^{enh,h_l} - *enh* size of level h_l. Not all *enh* levels up to h_l are packed into the message; only the levels updated are included. Given that a larger number of messages are grouped for communication, an additional compute cost (a packing and unpacking cost) c is added to communication per neighbor. Finally, p is the maximum number of neighbors communicated by a process when exchanging the grouped message, and δ is the size of a data element of the `op_dat` d in bytes.

4.3.3 Insights from Performance Model

The execution time of an OP2 parallel loop is determined by the slowest of the core iteration execution time and the halo message exchange time, as shown in Equation (4.1). When executing a series of OP2 parallel loops, each loop is executed one after the other, resulting in a total execution time for all loops, as demonstrated in Equation (4.2). In the CA version, the loops in the loop-chain are combined, and the core iterations of all the loops are executed first before executing the non-core iterations. This is different from the OP2 version, where the non-core iterations of the loops are executed right after the respective loop's core iterations. Moreover, Equations (4.2) and (4.3) illustrate that the CA version is more efficient since it sends a single message to a specific neighboring process instead of sending multiple messages to the same neighbor as in the OP2 version.

When there are more loops in the loop-chain, the CA version executes more core iterations while the message exchange is happening, resulting in faster communication compared to core computations. However, according to Equations (4.1) and (4.2), in the OP2 version, message exchange time can become the dominant factor compared to core iteration execution time at higher loop counts due to increased message exchanges. Therefore, we can assume that for longer loop-chains, the CA version will be more beneficial in terms of saving time on communication and yielding higher gains in both lower and higher node/process counts than the OP2 version.

When we increase the number of processes for a fixed problem size, the problem size allocated for each process decreases. Because of this, a single process executes fewer core iterations. Therefore, when we scale the problem, the communication time or the halo message exchange time becomes dominant for both the OP2 and CA versions, as described

in Equations (4.2) and (4.3). At this point, the version that performs the communications faster becomes the winner. The CA version reduces the number of messages sent, which results in lower communication time compared to the OP2 version even at lower loop counts, especially at higher node/process counts. Therefore, it can be hypothesized that CA performance gains will appear at higher node/process counts (i.e., with larger machine sizes) in a strong scaling scenario, particularly when the number of core iterations per partition (one partition is assigned per process in OP2) is smaller.

Furthermore, the communication time depends on the maximum number of halo layers (r) determining the message size (m^r). If r is significantly smaller than n, then the gains of the CA version are likely to be large. However, any performance gains from faster communications can be diminished if the sum of the times to execute over the multiple halos given by $\sum_{l=0}^{n-1} g_l S_l^h$ in Equation (4.3), is significantly larger than the sum of times to execute over a single halo region, given by $g_l S_l^1$ in Equation (4.2). Therefore, the maximum number of halo layers involved in the CA execution of the loop-chain has a significant impact on performance.

4.3.4 Challenges in Performance Modeling

Modeling the performance of unstructured-mesh applications presents various challenges arising from the nature of unstructured-meshes and the complexities they introduce into the computational simulations.

- Mesh Complexities

  When working with unstructured-meshes, the application needs to be provided with explicit connectivity information. This is due to the irregular node connectivity and element shapes. During application execution, accessing data and mesh elements requires the use of indirection maps, which can make predicting the application's behavior challenging.

- Load Imbalance

  When an unstructured-mesh application is executed in a distributed environment, the mesh may not be partitioned and allocated equally to the system's processes. This inefficient load balancing can lead to poor performance prediction and resource allocation.

- Communication Overhead

  Irregular mesh structures cause irregular communication patterns among the system's processes. Message sizes, message senders, and receivers are unpredictable at compile time in an unstructured-mesh application. Modeling the impact of uneven data movement and synchronization overhead can be a challenging task.

- Cache Performance

  The indirect memory accesses in these applications cause non-contiguous memory accesses. Mesh renumbering and data rearrangement steps taken to enhance the cache performance are not predictable during the compile time of the application. Considering the caching effect in performance modeling has become another challenge that researchers face.

- Application Complexities

  Most real-world unstructured-mesh applications consist of thousands to millions of lines of code and are written using various programming languages such as Fortran and C/C++ integrating multiple libraries including MPI, OpenMP, CUDA, and SYCL. The computations in them are even more complex to understand. Modeling the performance of such complex and coupled code requires multiple other interactions to be considered.

Addressing these challenges requires a comprehensive understanding of computational science, unstructured-mesh complexities, and performance modeling techniques. To effectively capture the complexities of unstructured-mesh simulations and account for their impact on different aspects of performance, such as computation, memory usage, communication, and scalability, sophisticated models must be developed. This level of precision is essential to ensure optimal results.

4.4 Performance

We now investigate the performance of the communication-avoiding back-end, applying it to two existing applications developed with OP2: (1) a representative CFD mini-app, MG-CFD [83] used for benchmarking and co-design and (2) the recently developed OP2 version of Hydra, OP2 Hydra [80], a large-scale production CFD application used at Rolls Royce plc. The OP2 code generator was extended so that it can automatically generate the loop-chain execution template in Algorithm 4.3 for selected loops. Then, the OP2 MPI back-end can be swapped with the new CA back-end and linked at compile-time. The high-level science source remains unmodified.

Performance is benchmarked on the ARCHER2 supercomputer, an HPE Cray EX system located at EPCC. Table 4.1 briefly details the key hardware and setup of the system. Each ARCHER2 node consists of two AMD EPYC 7742 processors, each with 64 cores (128 total cores) arranged in an 8×NUMA regions per node (16 cores per NUMA region) configuration [125]. Each node is equipped with 256 GB of memory. The nodes are interconnected by an HPE Cray Slingshot, 2×100 Gb/s bi-directional per node network. The GNU compiler collection version 10.2.0 was used on ARCHER2 with compiler flags noted in the table. For the MPI communication, Cray MPICH 8.1.23 was used. Runtimes of all the tests performed in this chapter can be found in Section B.2 of Appendix B.

Table 4.1: System specifications

System	**ARCHER2** HPE Cray EX [125]
Processor	AMD EPYC 7742 @ 2.25 GHz
(procs×cores)/node	2×64
Mem/node	256 GB
Interconnect	HPE Cray Slingshot 2×100 Gb/s bi-directional/node
OS	HPE Cray LE (SLES 15)
Compilers	GNU 10.2.0
Flags	`-O2 -eF -fPIC`
MPI	Cray MPICH 8.1.23

Unstructured-mesh-based applications require explicit connectivity information of mesh elements, which limits their ability to perform compile-time performance enhancements and static performance analysis. Before making any code changes for CA enhancements, we can only execute the inspection phase of the CA back-end linked to the application and gather the information required for the analytical model to predict the profitability of applying CA enhancements to the identified loop-chains. To support this, we have enhanced our CA back-end so that we can gather required information such as message sizes, and core and halo sizes by only executing the inspection phase of the CA back-end. This gathered information is used to analyze and establish the performance results of the thesis.

4.4.1 MG-CFD

MG-CFD [83] is a 3D unstructured multi-grid, finite-volume computational fluid dynamics (CFD) mini-app for inviscid-flow, developed by extending the CFD solver in the Rodinia benchmark suite. More information about MG-CFD can be found in Section 3.4.2. The code for MG-CFD is available as open-source software and can be accessed through [128]. To conduct our experiments, we used the NASA Rotor 37 meshes with 8M and 24M nodes. These meshes are commonly used for CFD validation and represent the geometry of a transonic axial compressor rotor.

Synthetic Loop-chains

As discussed in Section 4.2.2, an OP2 loop will exchange MPI halos for an `op_dat` if it is to be indirectly read (`OP_READ`) in a loop but has been modified (`OP_WRITE`, `OP_INC`, or `OP_RW`) in a preceding loop. In this case, we note the `op_dat`'s halos are *dirty* at the start of the loop, triggering an MPI halo exchange before accessing halo values for computation. As such, two consecutive loops, the first, modifying an `op_dat` followed by a second, reading the same data indirectly, will be our target access pattern (i.e., access descriptor in the loop-chain abstraction), for applying sparse tiling. However, consecutive loops with the above access descriptor do not exist in MG-CFD. Nevertheless, the relatively small

```
1  for (int i = 0; i < nchains; i++) {
2    // loop which modifies dataset var
3    op_par_loop(ca_write_kernel, "ca_write_kernel", op_edges[l],
4      op_arg_dat(var[l][i], 0, en[l], 5, "double", OP_INC),
5      op_arg_dat(var[l][i], 1, en[l], 5, "double", OP_INC));
6
7    // loop which reads the modified(dirtied) dataset var
8    op_par_loop(ca_read_kernel, "ca_read_kernel", op_edges[l],
9      op_arg_dat(var[l][i],  0, en[l], 5, "double", OP_READ),
10     op_arg_dat(var[l][i],  1, en[l], 5, "double", OP_READ),
11     op_arg_dat(ew[l]    , -1, OP_ID, 3, "double", OP_READ),
12     op_arg_dat(fluxes[l],  0, en[l], 5, "double", OP_INC),
13     op_arg_dat(fluxes[l],  1, en[l], 5, "double", OP_INC));
14 }
```

Listing 4.5: Expandable synthetic *2-loop-chain*

size and simplicity of MG-CFD means that we could add a *synthetic* sequence of loops to create the required setup. With such a configuration, we are then able to create arbitrarily extendable loop-chains with the above target access pattern allowing to examine the limits of a single loop-chain and observe its consequent performance, reasoning with the performance model. The new loops introduced to MG-CFD consist of two loops [16] as detailed in Listing 4.5. The first, `ca_write_kernel` modifies an `op_dat` named, `var` through an indirect increment, `OP_INC`, operation while iterating over the `edges`. Thus, `var` becomes dirty at the end of this loop. The second loop, `ca_read_kernel`, indirectly reads `var`. `ca_read_kernel` is in fact a copy of the most time-consuming loop in MG-CFD, the `compute_flux_edge` loop that has the same access modes. This enables us to make an effective comparison between reducing communications, i.e., no halo exchanges in the second loop, and the consequent increase in the (redundant) computations over the larger depth halos in the first loop.

This *2-loop-chain* is enclosed within an outer loop whereby setting its iteration count, `nchains`, we can create longer loop-chains to explore performance. For instance, setting `nchains` $= 8$ expands the existing loop-chain with 2 loops in Listing 4.5, 8 times to generate a loop-chain with 16 loops. However, the number of halo layers needed by the loop-chain will be decided by considering the data access patterns of the datasets in the expanded loop-chain according to Algorithm 4.4. With `nchains` $= 1$, the execution of the loop-chain, according to Algorithm 4.3, will not reduce the number of MPI messages exchanged. However, for `nchains` > 1, taking the resulting sequence of loops as a single loop-chain and applying Algorithm 4.3, we can see how multiple halo exchanges can be combined into a single larger message. We use this configuration to explore the performance on the ARCHER2 supercomputer.

ARCHER2 Results

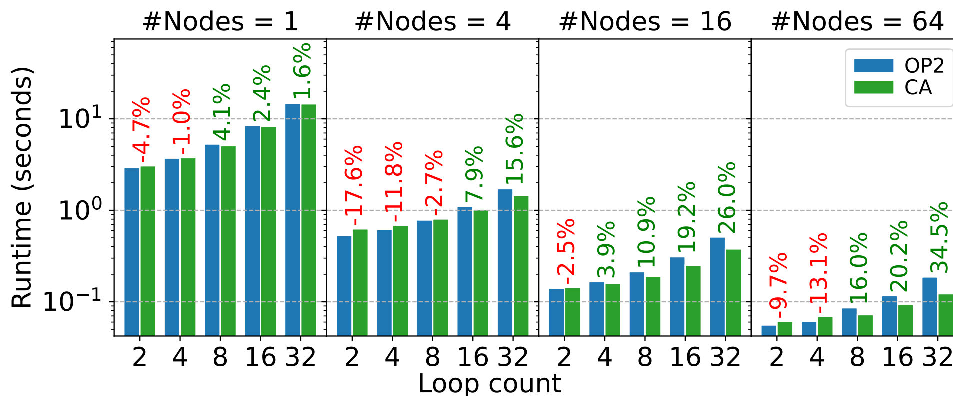


Figure 4.8: MG-CFD CA performance with 8M mesh on ARCHER2

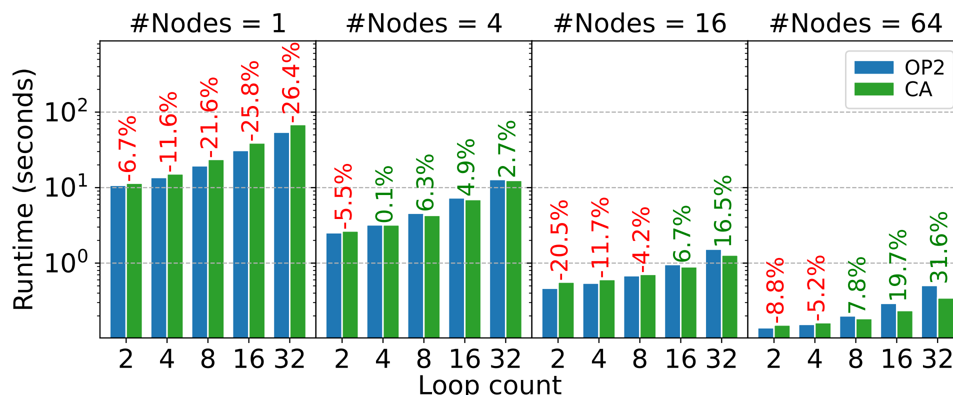


Figure 4.9: MG-CFD CA performance with 24M mesh on ARCHER2

Figure 4.8 and Figure 4.9 present execution times (min. of at least five runs each, CoV < 0.01) of MG-CFD on ARCHER2 for a mesh size of 8M and 24M, respectively. The reported runtime is the time taken by the main iteration loop. We have not included the constant cost for the inspection phase, which gets amortized (and is negligible) for larger numbers of main iterations, as is typical for real-world applications. Note the log scale of the y-axis. For both cases, we compare the original OP2 and the CA runtimes for loop-chains with loop counts $n = 2, 4, \ldots, 32$. For each run, we utilized the full 128 cores/node (128 MPI procs/node). Additionally, to obtain the best partitions per process, i.e., the smallest MPI halos and the least number of neighbors per process, we used the k-way partitioner routine from the ParMETIS library [124]. Increasing n from 2 to 32 will result in the original OP2 loops exchanging 16× more messages per neighbor. Only a single message is exchanged in the CA version. However, the grouped halo message size, m^r for CA can potentially contain a maximum of n halo layers. According to the data access patterns of the synthetic loop-chain, r is set to 2 for our benchmarking, making the message size m^2. We investigate the performance for the case where the number of `op_dats` exchanged remains constant at 2 per loop-chain. While this scenario is synthetic, as shown by Reguly et al. [36], for structured-mesh codes, it is a prevalent case we see in

Table 4.2: MG-CFD on ARCHER2 - 8M Mesh - Model Components: OP2 comms ($\sum(2dpm^1)$) - CA comms (pm^r) in bytes, OP2 core iterations ($\sum(S^c)$) - CA core iterations ($\sum(S^c)$), OP2 halo iterations ($\sum(S^1)$) - CA halo iterations ($\sum(S^h)$), and performance gain% of CA over OP2

#Nodes	#Loops	8M Mesh						
		OP2			CA			Gain%
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$	
1	2	2088960	395966	30622	4141880	381001	96763	-4.74
	4	4177920	791932	61244	4141880	706526	193526	-1.02
	8	8355840	1583864	122488	4141880	1223225	387052	4.08
	16	16711680	3167728	244976	4141880	1842929	774104	2.37
	32	33423360	6335456	489952	4141880	2110935	1548208	1.59
4	2	877600	96494	14408	1763200	90460	42292	-17.56
	4	1755200	192988	28816	1763200	161214	84584	-11.75
	8	3510400	385976	57632	1763200	258408	169168	-2.65
	16	7020800	771952	115264	1763200	341095	338336	7.90
	32	14041600	1543904	230528	1763200	351631	676672	15.62
16	2	365400	23148	5850	747600	20962	17485	-2.50
	4	730800	46296	11700	747600	34634	34970	3.91
	8	1461600	92592	23400	747600	47264	69940	10.85
	16	2923200	185184	46800	747600	49646	139880	19.16
	32	5846400	370368	93600	747600	49646	279760	26.00
64	2	168360	5658	2260	356160	4783	7277	-9.70
	4	336720	11316	4520	356160	7125	14554	-13.08
	8	673440	22632	9040	356160	8285	29108	15.98
	16	1346880	45264	18080	356160	8295	58216	20.20
	32	2693760	90528	36160	356160	8295	116432	34.45

real-world applications. The same was observed for Hydra, an unstructured-mesh code, but within much smaller loop-chains (see Section 4.4.2).

For both the 8M and 24M meshes, better runtimes can be seen at higher node counts with CA. The performance gains are also larger for higher loop counts. This aligns with the insights from the model, where the CA version saves on the number of messages sent without an increase in the message size. Up to 35% faster runtimes can be seen with CA, compared to the original OP2. Empirical measurements for the above runs provide further evidence for the performance trends. Table 4.2 and Table 4.3 detail the computation and communication model component factors for the 8M and 24M mesh execution with MG-CFD on ARCHER2. These were obtained by recording the message sizes and number of MPI neighbors for each tested configuration and substituting these values into the analytic model.

In Table 4.2 and Table 4.3, we can observe that the number of core computations ($\sum(S^c)$) performed during the halo exchange increases as the loop count increases for a given node count. The CA version performs fewer total core computations in the loop-chain than the OP2 version. However, in the CA version, all the core computations are performed together before executing the non-core computations. In contrast, the non-core computations of a given loop are executed immediately after its core iteration execution in

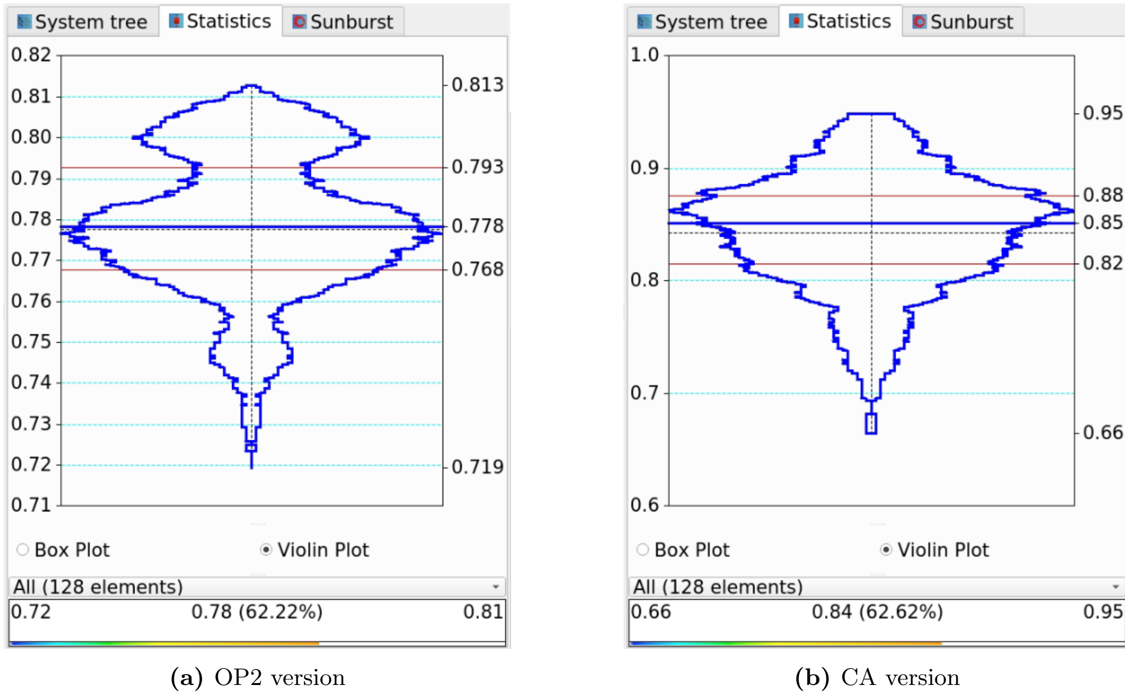
Table 4.3: MG-CFD on ARCHER2 - 24M Mesh - Model Components

#Nodes	#Loops	24M Mesh						
		OP2			CA			Gain%
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$	
1	2	4476960	1383744	78678	8791200	1346416	241975	-6.71
	4	8953920	2767488	157356	8791200	2562399	483950	-11.63
	8	17907840	5534976	314712	8791200	4662827	967900	-21.60
	16	35815680	11069952	629424	8791200	7772310	1935800	-25.76
	32	71631360	22139904	1258848	8791200	11029298	3871600	-26.35
4	2	2776200	340060	32220	5497800	325105	99731	-5.47
	4	5552400	680120	64440	5497800	601197	199462	0.11
	8	11104800	1360240	128880	5497800	1029981	398924	6.30
	16	22209600	2720480	257760	5497800	1513264	797848	4.93
	32	44419200	5440960	515520	5497800	1681473	1595696	2.66
16	2	1006720	82868	13344	2039840	77673	38976	-20.50
	4	2013440	165736	26688	2039840	137854	77952	-11.72
	8	4026880	331472	53376	2039840	217854	155904	-4.20
	16	8053760	662944	106752	2039840	277656	311808	6.73
	32	16107520	1325888	213504	2039840	283708	623616	16.47
64	2	406080	20154	5348	825600	18125	16183	-8.76
	4	812160	40308	10696	825600	29980	32366	-5.24
	8	1624320	80616	21392	825600	41653	64732	7.83
	16	3248640	161232	42784	825600	44690	129464	19.68
	32	6497280	322464	85568	825600	44690	258928	31.65

the OP2 version. This allows the CA version to hide latency for the halo message exchange better than the OP2 version for higher loop counts in lower node counts, providing better gains for the CA version, as explained in Section 4.3.3.

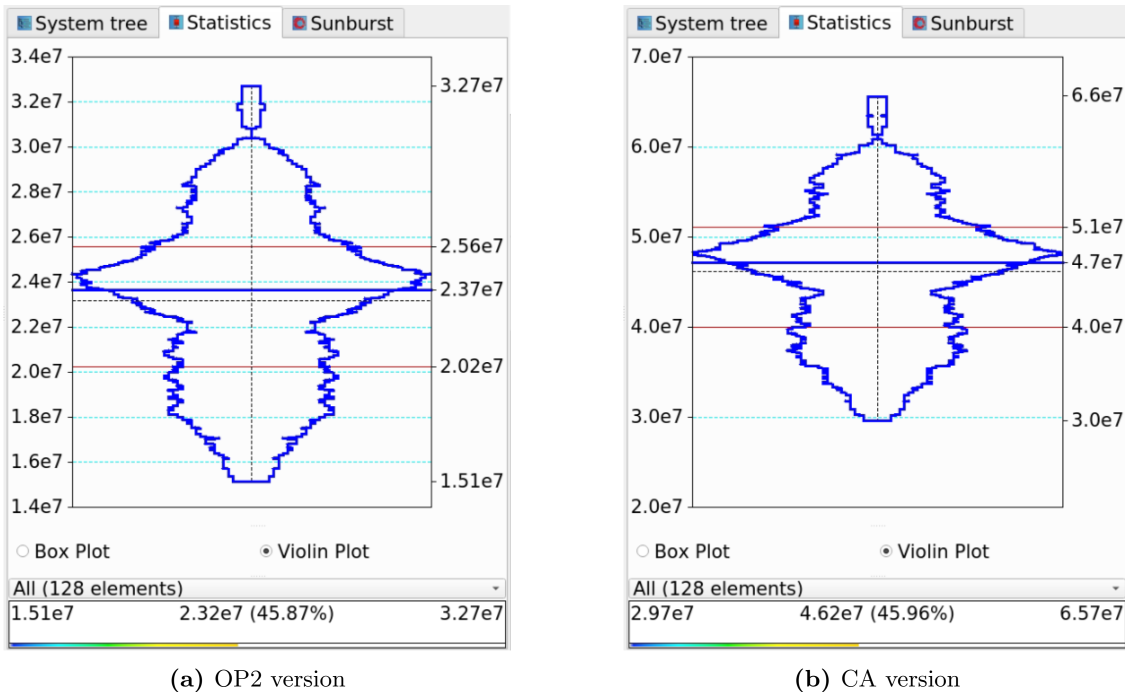
For instance, let us consider the performance gains for the 8M mesh in node 1 from Table 4.2. The number of core computations ($\sum(S^c)$) for the CA version increases from 381001 to 2110935 for loop counts 2 to 32 while keeping the exchanged halo message size (pm^r) constant at 4141880 bytes. In the OP2 version, for the same configurations, core iterations increase from 395966 to 6335456, and the message size ($\sum(2dpm^1)$) increases from 2088960 to 33423360 in bytes. Although the core computations ($\sum(S^c)$) increase in the OP2 version, they are not utilized together for latency hiding in the same way as the CA version. Hence, from loop count 8, the CA version starts to give performance benefits, due to latency hiding, even though it has extended halo computations ($\sum(S^h)$) compared to the OP2 version. Based on the performance model data presented in Table 4.2, we can observe that the OP2 version outperforms the CA version in terms of the total amount of communications (measured in bytes), starting from loop count 8. In other terms, this indicates that the communication reduction strategy implemented in the CA version begins to have a significant impact on the loop-chain at this point. This behavior is consistent across all other node counts as well.

We utilized Scalasca [129] and Score-P [130] to verify our analytical model's analysis of the results. Figure 4.10a and Figure 4.10b show the mean runtime of 0.78 seconds



(a) OP2 version (b) CA version

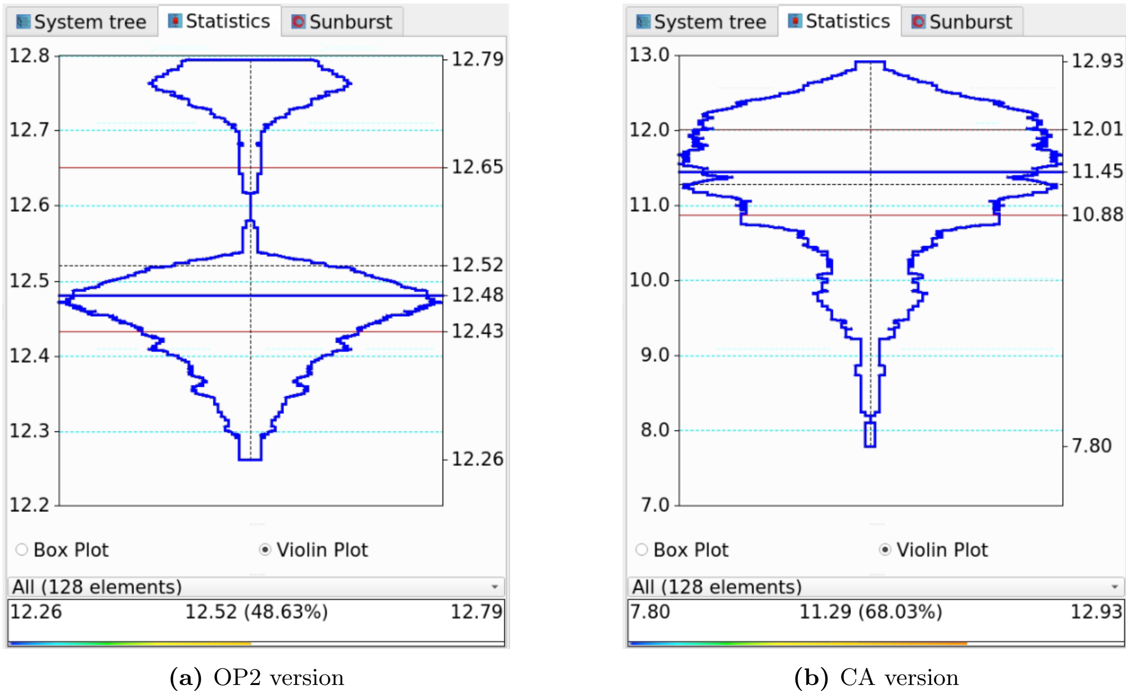
Figure 4.10: MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 2)



(a) OP2 version (b) CA version

Figure 4.11: MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 2)

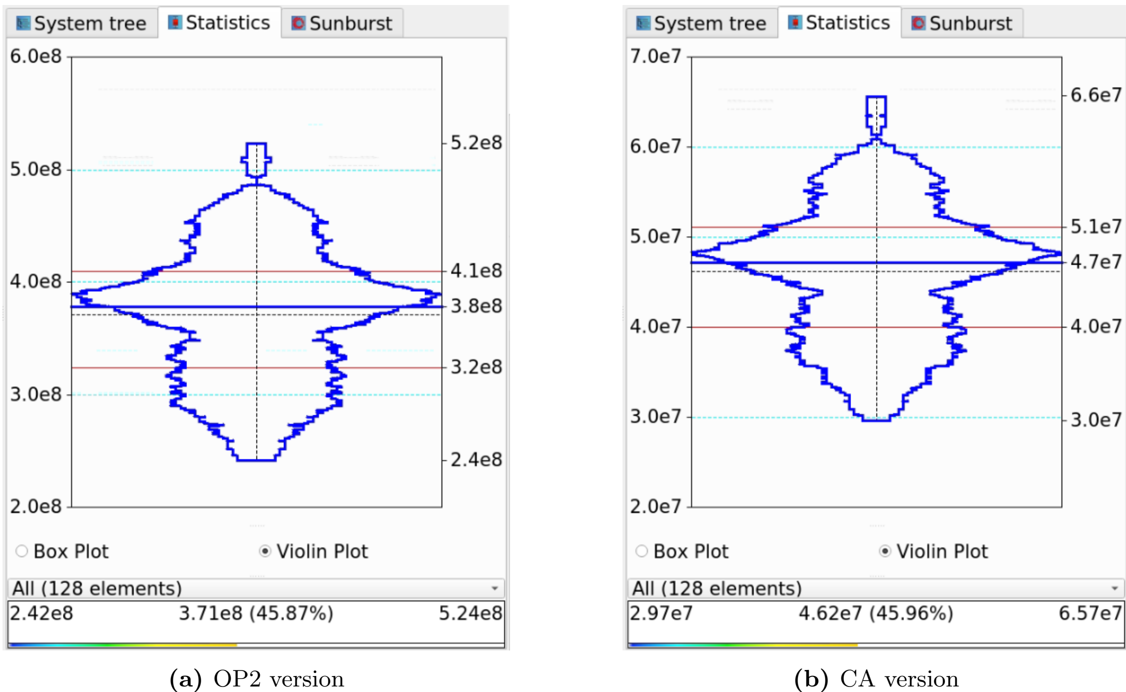
for the OP2 version and 0.84 seconds for the CA version, respectively, for loop count 2 in node count 1. Similarly, Figure 4.11a and Figure 4.11b show the exchanged mean data size of 2.32×10^7 bytes for the OP2 version and 4.62×10^7 bytes for the CA ver-



(a) OP2 version

(b) CA version

Figure 4.12: MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 32)

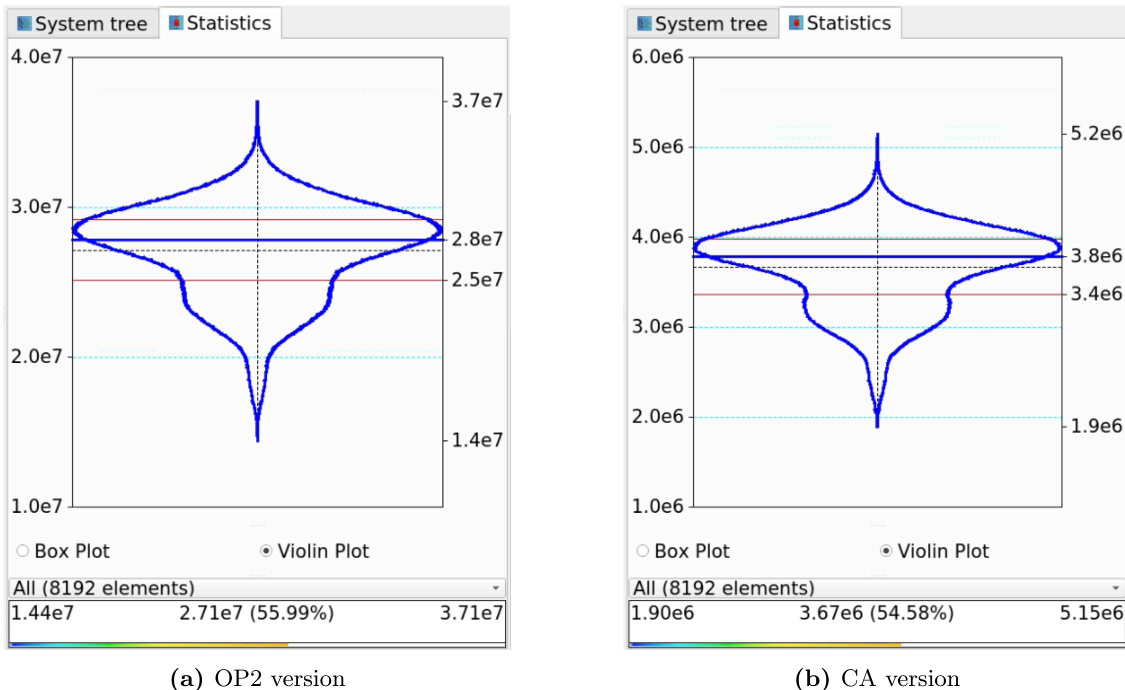


(a) OP2 version

(b) CA version

Figure 4.13: MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 1, #Loops - 32)

sion, respectively, for the same configurations. This profile data indicates that the OP2 version performs better than the CA version at loop count 2 in node count 1 with lower communication which aligns with our performance model insights.



(a) OP2 version **(b)** CA version

Figure 4.14: MG-CFD data exchange in bytes for synthetic loop-chain (Configurations: #Nodes - 64, #Loops - 32)

Furthermore, Figure 4.12a and Figure 4.12b show the mean runtime of 12.52 seconds for the OP2 version and 11.29 seconds for the CA version, for loop count 32 in node count 1. Furthermore, Figure 4.13a and Figure 4.13b show the exchanged mean data size of 3.71×10^8 bytes for the OP2 version and 4.62×10^7 bytes for the CA version, for the same configurations. This profile data demonstrates that the CA version outperforms the OP2 version at loop count 32 in node count 1 with lower communication which can also be inferred from the performance model data in Table 4.2, as explained earlier.

These profiling results support our arguments based on the analytical model, where the OP2 version performs better for lower loop counts in lower nodes, whereas the CA version performs better for higher loop counts in lower node counts, provided that a sufficiently large number of core computations are available for latency hiding, considering the aggregated message size in the CA version. Looking at the data presented in Table 4.3, it can be observed that for 24M, the CA version does not outperform the OP2 version for any of the loop counts tested on a single node. This indicates that the CA version lacks adequate core computations to mask the latency that comes with the aggregate message size increase in the CA version.

Per node, as the loop count increases, the amount of data communicated among the neighbors increases in the OP2 version ($\sum(2dpm^1)$), while it remains constant in the CA version (pm^r). The amount of core computations performed during the halo exchange is always smaller for the CA execution compared to the OP2 version. However, the amount of computations over the halos that are performed after the halo exchange is always higher in the CA version. As the node count increases, the number of core computations reduces

for both the OP2 and the CA versions, and communication costs become dominant. For instance, in Table 4.2, at node count 64, the amount of data exchanged in the OP2 version increases from 168360 to 2693760 in bytes, while the core computations increase from 5658 to 90528. In the CA version, the message size remains constant at 356160 bytes, while core computations increase from 4783 to 8295, which is not sufficient for complete latency hiding of message exchange. In both versions, communication becomes dominant, and the version that performs less communication with a lesser number of messages, which is the CA version, gives better performance gains. Insights from Scalasca and Score-P profiling data in Figure 4.14 show that the amount of communication is significantly lower in the CA version (mean - 3.67×10^6 bytes) compared to the OP2 version (mean - 2.71×10^7 bytes). Through this, we can confirm that our performance model prediction of a potential performance gain with lower communication at higher node and loop counts is accurate.

To summarize, we observed that for the 8M mesh, the benefits of reduced communication with CA become evident from 1 node onwards at the 8 loop count configuration. Similarly, for 4 nodes, this becomes apparent at 16 loop count. However, in the case of the 24M mesh, this advantage begins at a higher node count of 4 with a 4 loop count configuration.

4.4.2 OP2 Hydra

Our second application is the OP2 version of Rolls Royce's Hydra CFD application [80, 81] which we previously examined in Chapter 3. Hydra is a comprehensive production application designed for modeling various aspects of turbomachinery design, and its details can be found in Section 3.4.4. We are currently testing the same 8M and 24M node NASA Rotor 37 meshes that were previously tested with MG-CFD. OP2 Hydra consists of roughly 500 parallel loops, with more complex computations performed on the mesh than in the loops of MG-CFD.

A number of loop-chains were identified to target CA optimizations. Table 4.4 and Table 4.5 detail six loop-chains selected for our benchmarking. The constituent loops, their iteration set, access modes of `op_dats` that require halo exchanges, and required max halo layers for each loop are detailed in the tables. The loop-chains in Table 4.4 (`vflux`, `iflux`, and `jacob`) require only a single layer of halos, while the loop-chains in Table 4.5 (`weight`, `period`, and `gradl`) require multiple layers of halos for execution. However, these loop-chains consist of the most time-consuming loops in Hydra [81]. The relative costs or the proportional contributions of the loop-chains to the total runtime of Hydra are `vflux` 18%, `iflux` 5%, `gradl` 8%, and `jacob` 2%. The loop-chains, `weight` and `period` are inside the setup phase and outside the main time-stepping loop.

Table 4.4: OP2 Hydra loop-chains with single halo layer ($HE_l = 1$).

loop-chain: `iflux` (loop count = 2)

Parallel loop (l)	Iteration set (S)	Halo exchanged datasets	HE_l
`initviscres`	`nodes`	-	1
`iflux_edge`	`edges`	`qrg`	1

loop-chain: `vflux` (loop count = 2)

Parallel loop (l)	Iteration set (S)	Halo exchanged datasets	HE_l
`initres`	`nodes`	-	1
`vflux_edge`	`edges`	`qp, xp, ql, qmu, qrg`	1

loop-chain: `jacob` (loop count = 3)

Parallel loop (l)	Iteration set (S)	Halo exchanged datasets	HE_l
`jac_period`	`pedges`	`jac, jaca`	1
`jac_centreline`	`cbnd`	-	1
`jac_corrections`	`bnd`	`jac`	1

ARCHER2 Results

Performance of each loop-chain on ARCHER2 up to 128 nodes (16k cores) is detailed in Figure 4.15 and Figure 4.16 (CoV < 0.08). This is the cumulative time taken by each loop-chain for 20 iterations of the main time-stepping loop. Hydra's default partitioner based on the recursive inertial bisection of the mesh is used in all these experiments. We collected information on message sizes, core and halo sizes, the number of neighboring processes, and other parameters through the CA back-end inspection phase to analyze the results. The data in Table 4.6 and Table 4.7 is used for our analysis with the analytical model.

In Table 4.6, we can observe that the loop-chains, `weight`, `period`, and `jacob` exhibit performance improvements with Hydra for the 8M mesh on ARCHER2. The `weight` loop-chain specifically shows the highest gain of 14% at the 128 node run, where it has one of the highest communication reductions of 53% compared to the OP2 version, with a relatively low computation increase compared to other node counts. However, in Table 4.7, for the 24M mesh run, we do not see any performance gains for the `weight` loop-chain. This is primarily due to the higher aggregated message size, with relatively low core iterations for latency hiding, and a comparatively high increase in halo computations.

For the `period` loop-chain, we observe that the highest performance gain of 42% occurs at the 64 node run where the communication reduction is the highest in comparison to all other runs, as in Table 4.6. Additionally, there is only a relatively low increase in computation for the 8M mesh. A similar trend is also observed for the 24M mesh in Table 4.7 for the `period` loop-chain. However, for the `iflux` and `vflux` loop-chains,

Table 4.5: OP2 Hydra loop-chains with multiple halo layers ($HE_l >= 1$)

loop-chain: `weight` (loop count = 5)

Parallel loop (l)	Iter. set (S)	Access modes and halo ext.		HE_l
		`mode`$_{qo}$	HE_{qo}	
`sumbwts`	`bnd`	INC	2	2
`periodsym`	`pedges`	RW	1	1
`centreline`	`cbnd`	WRITE	2	2
`edgelength`	`edges`	RW	2	2
`periodicity`	`pedges`	RW	1	1

loop-chain: `period` (loop count= 6)

Parallel loop (l)	Iter. set (S)	Access modes and halo ext.				HE_l
		`mode`$_{qo}$	HE_{qo}	`mode`$_{vol}$	HE_{vol}	
`negflag`	`pedges`	-	1	RW	2	2
`limxp`	`edges`	RW	2	READ	1	2
`periodicity`	`pedges`	RW	1	-	1	1
`limxp`	`edges`	RW	2	READ	1	2
`periodicity`	`pedges`	RW	1	-	1	1
`negflag`	`pedges`	-	1	RW	1	1

loop-chain: `gradl` (loop count = 2)

Parallel loop (l)	Iter. set (S)	Access modes and halo ext.				HE_l
		`mode`$_{qp}$	HE_{qp}	`mode`$_{ql}$	HE_{ql}	
`edgecon`	`edges`	INC	2	INC	2	2
`period`	`pedges`	RW	1	RW	1	1

there are no performance gains with the CA version for both the 8M and 24M mesh runs, even though there is neither communication reduction nor computation increase in the CA version when compared to the OP2 version. Despite sending an aggregated message for halo exchanges instead of multiple messages for individual datasets, this message aggregation did not help to hide the latency in the CA version due to a low number of core iterations. As a result, no significant performance gains were observed.

In the `jacob` loop-chain, we avoid sending a redundant message in the CA version with no halo increase. The data in both Table 4.6 and Table 4.7 shows that the `jacob` loop-chain only experiences a reduction in communication with no increase in computation in the CA version. This scenario is always favorable for the CA version as it avoids extra computations and communication. However, for the `gradl` loop-chain, we observe both an increase in communication (i.e., negative communication reduction) and an increase in computation for the CA version. Hence, the `gradl` loop-chain does not provide any performance benefits for either the 8M or 24M mesh runs.

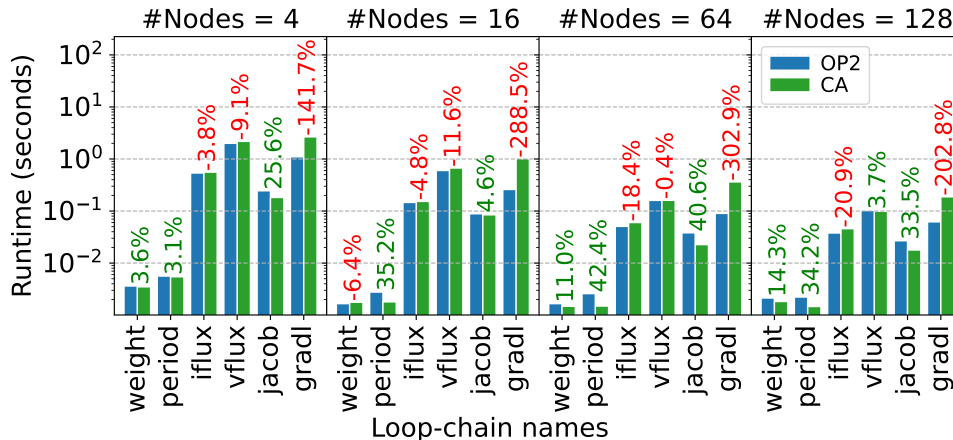


Figure 4.15: Hydra CA performance with 8M mesh on ARCHER2

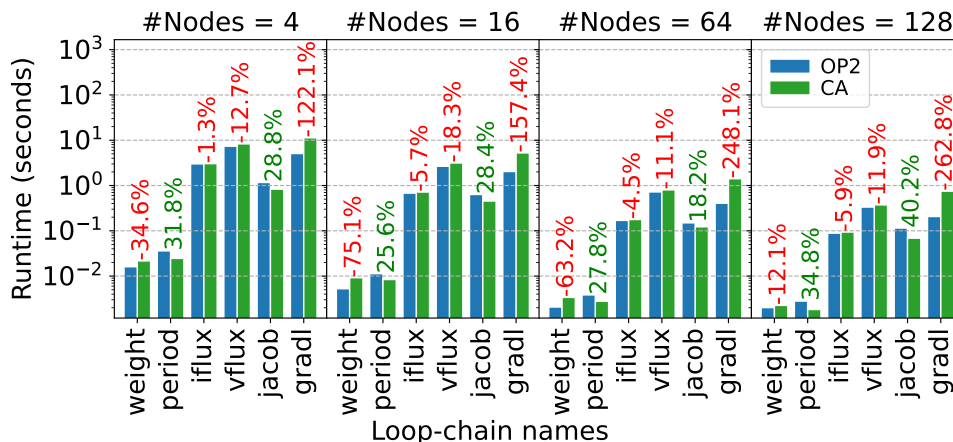


Figure 4.16: Hydra CA performance with 24M mesh on ARCHER2

In summary, results indicate that the loop-chains with the highest communication reduction, `period` and `jacob` showed performance improvements with CA – 42% and 40% on 64 nodes for the 8M problem, respectively. The `weight` loop-chain showed performance gains only with 8M mesh with a maximum of 14% on 128 nodes due to its communication reduction not being adequate enough to outperform the computation increase with 24M mesh. Other loop-chains, executing them as individual OP2 loops gave the best performance. Again the insights from the model as in Table 4.6 and Table 4.7, align with these results where loop-chains with higher communication reduction preferably with large loop counts tend to break-even the balance of computation vs communications performance.

4.5 Conclusion

Loop-chains with a higher communication reduction compared to the increased redundant computation due to added halo extensions show performance gains in Hydra. The communication reduction and the total core size for latency hiding of `period` is significantly higher than that of `weight`, giving it a higher performance gain as indicated through the

Table 4.6: Hydra loop-chains (LCs) on ARCHER2 - 8M Mesh - Model Components

LC(#Loops)	#Nodes	8M Mesh								
		OP2			CA			Loop-chain Gain%	Comm Reduction %	Comp Increase %
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$			
weight(5)	4	31694400	50206	37538	21051360	50184	135189	3.57	33.58	72.23
	16	16387200	13045	19346	8610624	13045	62767	-6.42	47.46	69.18
	64	5121792	3458	6616	2154240	3458	23373	11.01	57.94	71.69
	128	2482272	1800	3612	1150560	1800	14121	14.29	53.65	74.42
period(6)	4	51063200	93122	75076	17542800	93089	271347	3.11	65.64	72.33
	16	26401600	22652	38692	7175520	22652	128252	35.16	72.82	69.83
	64	8251776	5434	13232	1795200	5434	47115	42.35	78.24	71.92
	128	3999216	2622	7224	958800	2622	27821	34.25	76.03	74.03
iflux(2)	4	5282400	62842	16674	5282400	62842	16674	-3.84	0.00	0.00
	16	2731200	15403	7360	2731200	15403	7360	-4.75	0.00	0.00
	64	853632	3737	2686	853632	3737	2686	-18.37	0.00	0.00
	128	413712	1821	1572	413712	1821	1572	-20.93	0.00	0.00
vflux(2)	4	59867200	62842	16674	59867200	62842	16674	-9.11	0.00	0.00
	16	30953600	15403	7360	30953600	15403	7360	-11.58	0.00	0.00
	64	9674496	3737	2686	9674496	3737	2686	-0.44	0.00	0.00
	128	4688736	1821	1572	4688736	1821	1572	3.68	0.00	0.00
jacob(3)	4	89800800	3658	10432	45780800	3658	10432	25.59	49.02	0.00
	16	46430400	1719	5993	23670400	1719	5993	4.57	49.02	0.00
	64	14511744	741	1965	7398144	741	1965	40.61	49.02	0.00
	128	7033104	489	1020	3585504	489	1020	33.48	49.02	0.00
gradl(2)	4	52824000	46548	27106	105256800	46537	123131	-141.72	-99.26	77.99
	16	27312000	11326	13353	43053120	11326	55039	-288.51	-57.63	75.74
	64	8536320	2717	4651	10771200	2717	20065	-302.88	-26.18	76.82
	128	4137120	1311	2592	5752800	1311	11983	-202.80	-39.05	78.37

model in Table 4.6 and Table 4.7. Loop-chains such as `iflux` and `vflux`, which reduce the number of messages with a grouped halo, perform latency hiding with core execution but with no reduction of communication, are unlikely to give performance gains on CPU clusters. Loop-chains such as `gradl` which cause an increase in communication and computation, tend to degrade the performance even with sufficient latency hiding core computations and grouped halos. On the other hand, loop-chains such as `jacob`, which reduce communication with latency hiding and no computation increase, always tend to give performance benefits.

In general, there is a message size increase and a message count decrease in the CA version. However, it does not change the load balance of the tasks divided among the processes compared to the OP2 version. Message exchange between processes is always associated with message packing and unpacking costs. CA/OP2 message packing cost ratio is equivalent to the CA/OP2 total halo message sizes ratio. However, the OP2 version does not suffer from a message unpacking cost since the messages related to a particular dataset are directly copied to the relevant dataset array when receiving the message. However, there is an additional message unpacking cost for the CA version

Table 4.7: Hydra loop-chains (LCs) on ARCHER2 - 24M Mesh - Model Components

LC(#Loops)	#Nodes	24M Mesh								
		OP2			CA			Loop-chain Gain%	Comm Reduction %	Comp Increase %
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$			
weight(5)	4	123187680	175002	93379	87674208	174962	463440	-34.63	28.83	79.85
	16	52273008	45374	48580	26528640	45374	220806	-75.07	49.25	78.00
	64	21682080	12001	19674	8653680	12001	70853	-63.24	60.09	72.23
	128	9895104	6198	11408	3988608	6198	40843	-12.12	59.69	72.07
period(6)	4	198469040	328836	186758	73061840	328776	926734	31.78	63.19	79.85
	16	84217624	80496	97160	22107200	80496	441079	25.64	73.75	77.97
	64	34932240	19562	39348	7211400	19562	144921	27.78	79.36	72.85
	128	15942112	9606	22816	664768	9606	81427	34.78	95.83	71.98
iflux(2)	4	20531280	220854	63381	20531280	220854	63381	-1.27	0.00	0.00
	16	8712168	54387	28734	8712168	54387	28734	-5.74	0.00	0.00
	64	3613680	13317	7972	3613680	13317	7972	-4.49	0.00	0.00
	128	1649184	6572	4530	1649184	6572	4530	-5.91	0.00	0.00
vflux(2)	4	232687840	220854	63381	232687840	220854	63381	-12.72	0.00	0.00
	16	98737904	54387	28734	98737904	54387	28734	-18.29	0.00	0.00
	64	40955040	13317	7972	40955040	13317	7972	-11.14	0.00	0.00
	128	18690752	6572	4530	18690752	6572	4530	-11.91	0.00	0.00
jacob(3)	4	349031760	10642	14999	177937760	10642	14999	28.79	49.02	0.00
	16	148106856	5126	9923	75505456	5126	9923	28.36	49.02	0.00
	64	61432560	2220	5851	31318560	2220	5851	18.19	49.02	0.00
	128	28036128	1395	3439	14292928	1395	3439	40.21	49.02	0.00
gradl(2)	4	205312800	164360	78380	438371040	164340	445413	-122.10	-113.51	82.40
	16	87121680	40248	38657	132643200	40248	207445	-157.35	-52.25	81.37
	64	36136800	9781	13823	43268400	9781	61785	-248.07	-19.74	77.63
	128	16491840	4803	7969	19943040	4803	34610	-262.79	-20.92	76.97

when copying the data elements of multiple datasets received in the same message to relevant dataset arrays. However, this unpacking cost becomes negligible due to the chunk memcopy operations performed on the received messages in the CA version compared to the multiple message exchange cost of the OP2 version.

Chapter 5

Integrating Shared- and Distributed-Memory Communication-Avoiding Optimizations for CPUs

So far, we have studied the performance of unstructured-mesh-based applications that utilize communication-avoidance techniques in shared- and distributed-memory systems. It is important to analyze the behavior of each optimization strategy separately and then combine them to enhance the performance of these applications on high-performance systems. By doing so, we can identify new challenges and opportunities for optimizing unstructured-mesh-based applications with communication-avoidance. In this chapter, we evaluate the performance of unstructured-mesh applications for shared- and distributed-memory parallelization (SDMP) optimizations on CPU clusters, using the best-performing OP2 version with the communication-avoidance (CA) version of the same application.

5.1 Shared- and Distributed-Memory Parallelism

In order to address complex computational problems and overcome limitations posed by modern supercomputer architecture, this approach combines two parallelisms: shared-memory and distributed-memory. However, when merging these two parallelisms, applications may face challenges in improving performance with communication-avoidance techniques. In this section, we will examine the advantages and strengths of this hybrid approach regardless of the optimization strategy.

- Scalability and optimal resource utilization

  On-node parallelism exploits shared-memory's high bandwidth and low latency within a single node, while distributed-memory parallelism spans multiple nodes for scalability. Combining both approaches ensures that the total computational

power of each node is harnessed efficiently while also enabling scaling across the entire system for broader problem sizes.

- Balanced communication and computation

  Optimizing the data movement mechanism in a node's memory system can be achieved through on-node memory parallelism while improving inter-node communication strategies can be done through distributed-memory parallelism. Many applications have components that can benefit from both approaches, but it is important to avoid interference between the enhancements to prevent any negative impact on performance.

- Enhanced parallelism

  We can increase the parallelism of an application by increasing the running thread and node counts allocated for the application. This introduces some flexibility to the applications in selecting the optimizations going to be applied. We must develop hybrid algorithms to support shared- and distributed-memory parallelism. Enhancing the algorithms to cater to these requirements is very challenging but rewarding.

Developing a framework to combine shared- and distributed-memory systems presents new challenges, despite the benefits of optimal resource utilization, efficient memory management, scalability, and versatility.

First, we will explore SDMP on CPU clusters. We will elaborate the OP2 implementation that supports SDMP for CPU clusters and the extension of our CA back-end further to add shared-memory parallelization techniques with the SLOPE library along with the distributed-memory parallelization enhancements. We will compare the best-performing CPU version of OP2 with the CA-based SDMP version.

5.2 OP2 based SDMP for CPU Clusters

When dealing with diverse computer systems such as distributed-memory computing clusters, it is essential to utilize multiple layers of parallelism to achieve parallel execution of an application [9]. This involves combining thread-level and process-level parallelism, which is the combination of shared-memory and distributed-memory parallelism. The OP2 library currently has a CPU cluster back-end that uses OpenMP multi-threading for shared-memory parallelization and MPI message communication for distributed-memory parallelism. When integrating shared- and distributed-memory parallelism, OP2 was designed with the following considerations in mind:

1. Combining the distributed-memory handling strategy, the *owner-compute* model [9], with the coloring algorithm used in parallelizing on-node computations.

2. Overlapping mechanism of computation (which includes the coloring algorithm execution and core iteration calculation phase) and communication (which involves inter-node/process message/halo exchange).

Algorithm 5.1: Loop-chain inspection/execution with CA+SLOPE

```
Input: Loop-chain, 𝕃 = {L_0, ..., L_{n-1}}, op_dats, 𝔻 used in 𝕃, Tile size(t)
Result: Execute loop-chain with CA + sparse tiling

   // Find op_dats requiring halo syncs:  𝔻^h ⊆ 𝔻
 1 𝔻^h ← halo_exch_dats(𝔻, < M, mode >, 𝕃);

   // Compute #halo layers required for each loop in 𝕃
 2 ℍ𝕃_l ← calc_halo_layers(𝕃, 𝔻^h);

   // Find core, exec, & non-exec halo #iters for loops in 𝕃
 3 𝕊_l^c, 𝕊_l^h, 𝕊_l^n ← calc_iters(𝕊_l, ℍ𝕃_l);

   // Rearrange and renumber multi-layered core, eeh & enh for each
      op_dat d ∈ 𝔻^h
 4 𝕊^eeh,𝕊^enh ← calc_halos(𝔻^h, ℍ𝕃_l, 𝕊_l^c, 𝕊_l^h, 𝕊_l^n);

   // SLOPE inspection algorithm will assign all iterations including ieh
      to tiles
 5 𝕋 (𝕋^c, 𝕋^h) ← slope_inspection(𝕊_l^c, 𝕊_l^h, 𝕊_l^n, t);

 6 m^{eeh+enh} ← create_grouped_msg(𝔻^h, 𝕊^eeh,𝕊^enh);

 7 MPI_Isend(m^{eeh+enh});
 8 MPI_Irecv(m^{ieh+inh});

 9 foreach color do
10 |  foreach tile T ∈ 𝕋^c && T.color == color do
11 |  |  foreach iteration I ∈ T do
12 |  |  |  execute_iteration(I);
13 |  |  end foreach
14 |  end foreach
15 end foreach

16 MPI_Wait(m^{eeh+enh}, m^{ieh+inh});

17 foreach color do
18 |  foreach tile T ∈ 𝕋^h && T.color == color do
19 |  |  foreach iteration I ∈ T do
20 |  |  |  execute_iteration(I);
21 |  |  end foreach
22 |  end foreach
23 end foreach
```

5.3 Integrating CA Back-End and SLOPE Library

The CA back-end developed and explained in Chapter 4, supports the distributed-memory parallelism with communication-avoidance. The SLOPE library [8, 121] explained in Chapter 3, supports shared-memory parallelism with communication-avoidance through OpenMP, enhanced with full sparse tiling. We incorporated these two versions to achieve shared- and distributed-memory parallelism with inter- and intra-node communication-avoidance.

5.3.1 CA+SLOPE Inspection for Distributed-Memory Parallelism

To enable communication-avoidance in a distributed-memory platform using the SLOPE library, the Algorithm 4.3 is improved as detailed in Algorithm 5.1.

First, it is necessary to divide and distribute the unstructured-mesh among the application's processes as in Figure 5.1. Then, we identify the `op_dats` that require a halo exchange during the execution of a specific loop-chain. Using the algorithm presented in Algorithm 4.4, we determine the additional halo layers needed for these `op_dats`. After that, we calculate the required sizes for the halos (*execute* and *non-execute*), as well as for the *core*. Finally, we rearrange and renumber the multi-layered halo data structure based on the local arrangement of the mesh elements. These steps resemble those of the new CA back-end in Chapter 4 and are further elaborated in Section 4.2.6.

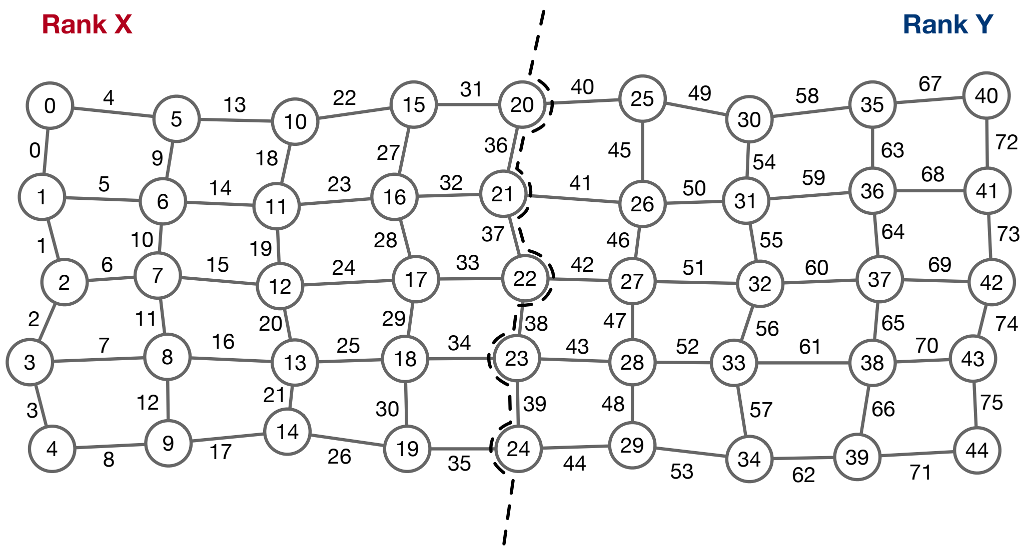


Figure 5.1: Mesh partitioning for processes

The information generated so far in the above-mentioned steps in the CA back-end will be fed to the SLOPE library to carry out its inspection phase for shared-memory parallelism (see line 5 in Algorithm 5.1).

5.3.2 SLOPE Inspection with Distributed-Memory Parallelism

The SLOPE library can now receive information on *core*, *boundary* (*export-exec* and *import-exec*), and *non-exec* iterations since the CA back-end has generated the required halos. The first step involves selecting the *seed loop* for the loop-chain. This loop is the one in which the iteration space is initially partitioned and assigned to tiles. The iteration space should fully represent the unstructured-mesh so that *projections* can be made when assigning iterations of the other loops to tiles. The terminology used in the SLOPE library is well-explained in Chapter 3, but a brief explanation is provided here for clarity. It is important to note that for distributed-memory parallelism, the *seed loop* must be the first loop of the loop-chain [8]. This is a limitation of the SLOPE library.

Firstly, the *core* region of the *seed loop* (L_0) is segmented into partitions and allocated distinct colors. To ensure optimal execution, neighboring partitions must not have the same color. The color assignment follows an ascending order, with lower numbers signifying higher priority. This helps to improve spatial locality when moving execution from one tile to another, particularly when partitions with adjacent color numbers are also physically adjacent.

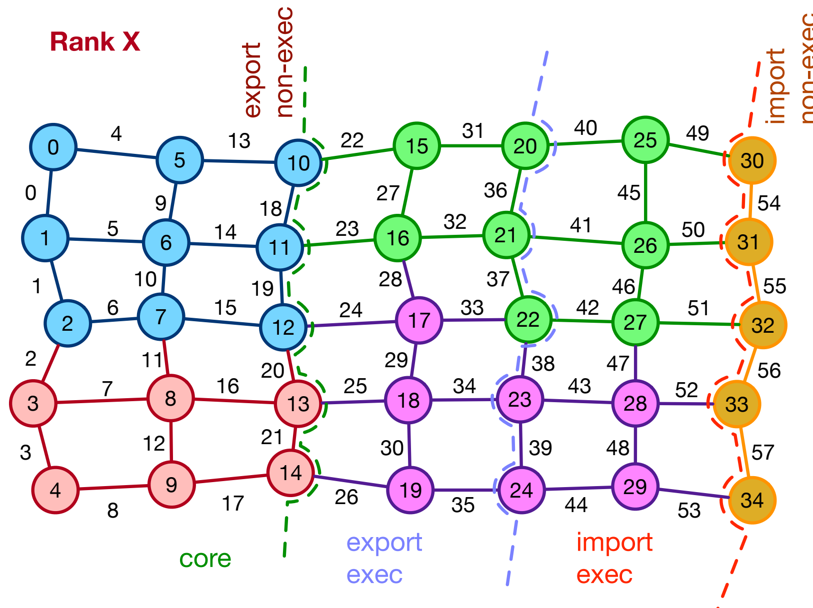


Figure 5.2: Coloring Rank X

The process of partitioning and coloring is applied to the boundary iterations (*export-exec* and *import-exec* iterations) of the *seed loop*, L_0. During the assignment of colors to tiles, color numbers with lesser priority than those assigned to the core region tiles are used. This ensures that core tiles are executed before boundary tiles during the execution phase, while the halo exchange is in-flight, to hide latency.

The SLOPE library maps all iterations of the *non-exec* region to a single tile and assigns the lowest priority color number. These iterations are not executed but are read when executing other iterations. Figure 5.2 and Figure 5.3 show the partitioning and

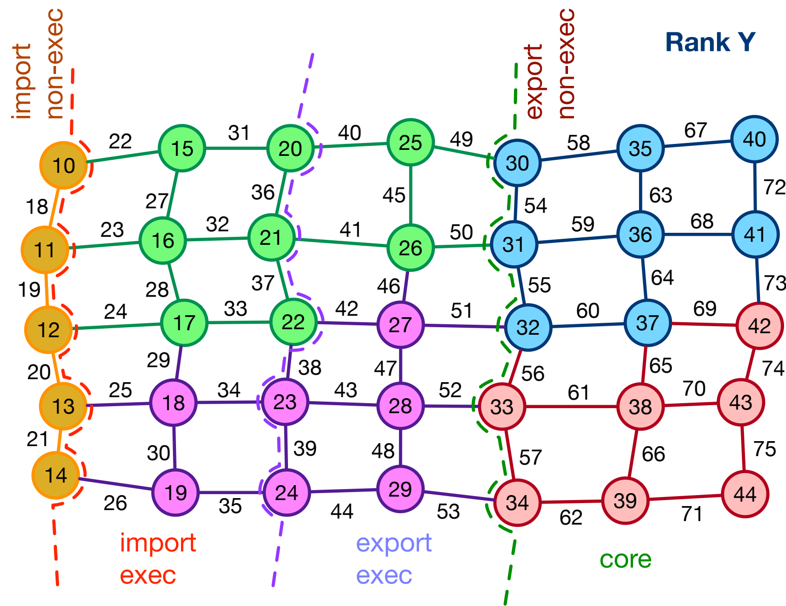


Figure 5.3: Coloring Rank Y

coloring of the mesh in Figure 5.1, distributed between rank X and rank Y of the system. After completing the *seed loop* iteration space partitioning and coloring, the SLOPE inspection phase moves to other loops of the loop-chain. The iterations of the other loops are assigned to tiles using *projections* calculated with the `MAX` function, as explained in Section 3.3.1. This is because only *forward-tiling* is available with distributed-memory parallelism.

5.3.3 SLOPE Execution with Distributed-Memory Parallelism

Once the tile schedule has been established, it is time to move on to the execution phase. This process is clearly outlined in lines 6-23 of Algorithm 5.1. The first step involves creating a grouped halo message using the steps described in Section 4.2.7. Once the message has been formed, it is sent and the loop execution begins. During the message transit, the assigned iterations for the core tiles are executed in parallel. Tiles of the same color will execute their assigned iterations together, with the colors being executed in order of priority, one after the other.

The library waits until the halo exchange is complete before proceeding with the *boundary/halo* tile execution. The execution of *core* tiles during the exchange of *halo* messages allows for overlapping computation and communication. After receiving the halo messages, the boundary tile iterations are executed in the same manner as the *core* tile execution. The *halo* tile computations include redundant computations from imported *halo* regions.

The expected outcome of the SDMP for CPU clusters with the SLOPE library is that the efficiency gained through spatial locality will outweigh the redundant com-

putation overhead. This will be accomplished through communication-avoidance with grouped message exchange and by avoiding multiple synchronization points in loop-chain execution, which will help reduce existing communication overhead. To achieve this goal, it is important that the redundant computations are fewer in number than the core computations. This will also result in a reduction of redundant information exchange. These aspects will be further analyzed and explained in the following sections, along with real-world application testing.

5.3.4 OP2-CA-SLOPE Integration

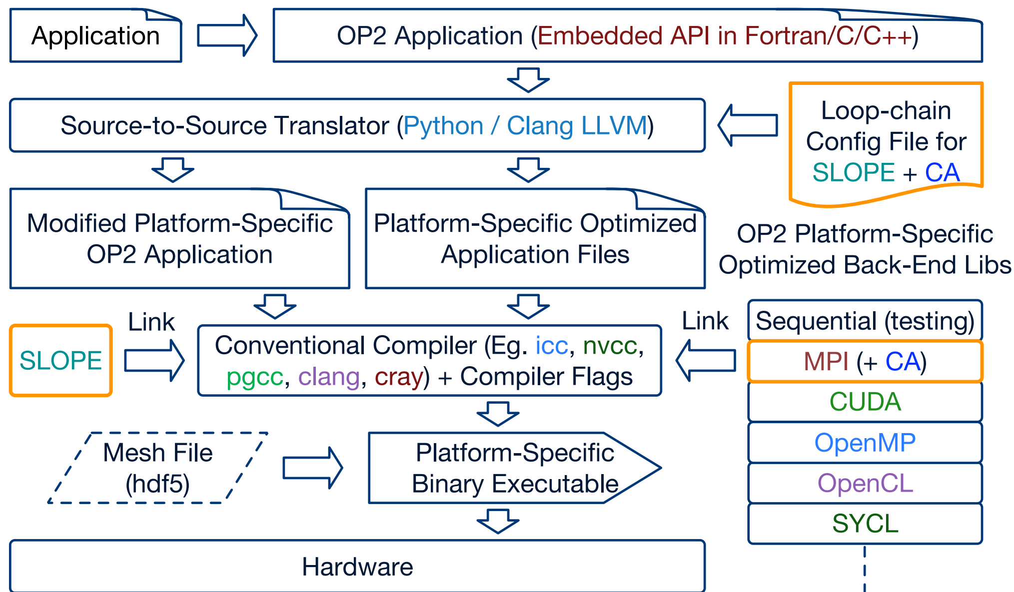


Figure 5.4: OP2 code generation with CA+SLOPE

To enhance both shared- and distributed-memory parallelizations with the OP2 DSL, we integrated the newly developed CA back-end and SLOPE library into the OP2 library. Figure 5.4 shows the modifications added to the original OP2 architecture which is explained in Section 2.6.6. We use a configuration file, similar to that used for the CA framework, to specify the loop count, loops, and their halo extensions. The OP2 code generator was then modified to generate code with both CA and SLOPE optimizations added to the loop-chain. This generated code is compiled using a traditional compiler, with the CA back-end and the SLOPE library linked in, to produce an executable for the designated computing platform.

5.4 Performance Model Extension for CA with On-Node Sparse Tiling

We have created an analytical performance model in Section 4.3 for the new communication-avoidance back-end that we developed in Chapter 4. We are expanding the same model to include support for CA with the SLOPE library.

As we explained earlier, sparse tiling has been integrated to avoid on-node communication. This was done by incorporating the SLOPE library [8, 121] and getting OP2 to generate code that uses SLOPE's inspector/execution functions. The CA with SLOPE execution algorithm, Algorithm 5.1, retains the multi-layered halo creation routines from Algorithm 4.3, but changes the loop execution to a colored tile execution over the *core*, *eeh*, and *ieh* as detailed in previous sections and by Luporini et al. [37]. Tiles per color can now be executed in parallel using OpenMP threads. Tiled execution will result in a modified grind time, γ_l replacing g_l in Equation (4.3). Additionally, each MPI process will handle a larger partition, resulting in larger message sizes sent to each neighbor.

Equation (4.3) needs to be adjusted to reflect these changes. The relevant equations for OP2 and CA versions are shown below.

$$T_{op2,l} = MAX\left[\ g_l S_l^c,\ 2 d_l p_l (L + m_l^1/B)\ \right] + g_l S_l^1 \tag{5.1}$$

$$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} \gamma_l S_l^c,\ p(L + m^r/B + c)\right] + \sum_{l=0}^{n-1} \gamma_l S_l^h \tag{5.2}$$

5.5 Performance

During our analysis in Chapter 3, we looked into the efficiency of the SLOPE library when it comes to shared-memory parallelism. Additionally, in Section 4.4, we looked into the performance of the CA back-end. Our current objective is to evaluate the combined performance of both frameworks for shared- and distributed-memory parallelism. To accomplish this, we will be utilizing MG-CFD and OP2 Hydra as our primary applications, as we did during the CA back-end testing for CPUs. To conduct our performance assessment, we will be utilizing the ARCHER2 supercomputer, of which the specifications are given in Table 4.1.

We conducted a series of tests on ARCHER2 for MG-CFD to find the best combination of MPI processes and thread count per process that can deliver optimal performance with SDMP. Our tests varied the number of MPI processes and thread count per process for a single node, considering both the OP2 MPI+OpenMP and the CA+SLOPE versions of MG-CFD, as shown in Figure 5.5.

Our runtime results, as shown in Figure 5.5 revealed that the MPI-only version (with 128 MPI processes and a single OpenMP thread per MPI process) provided the best

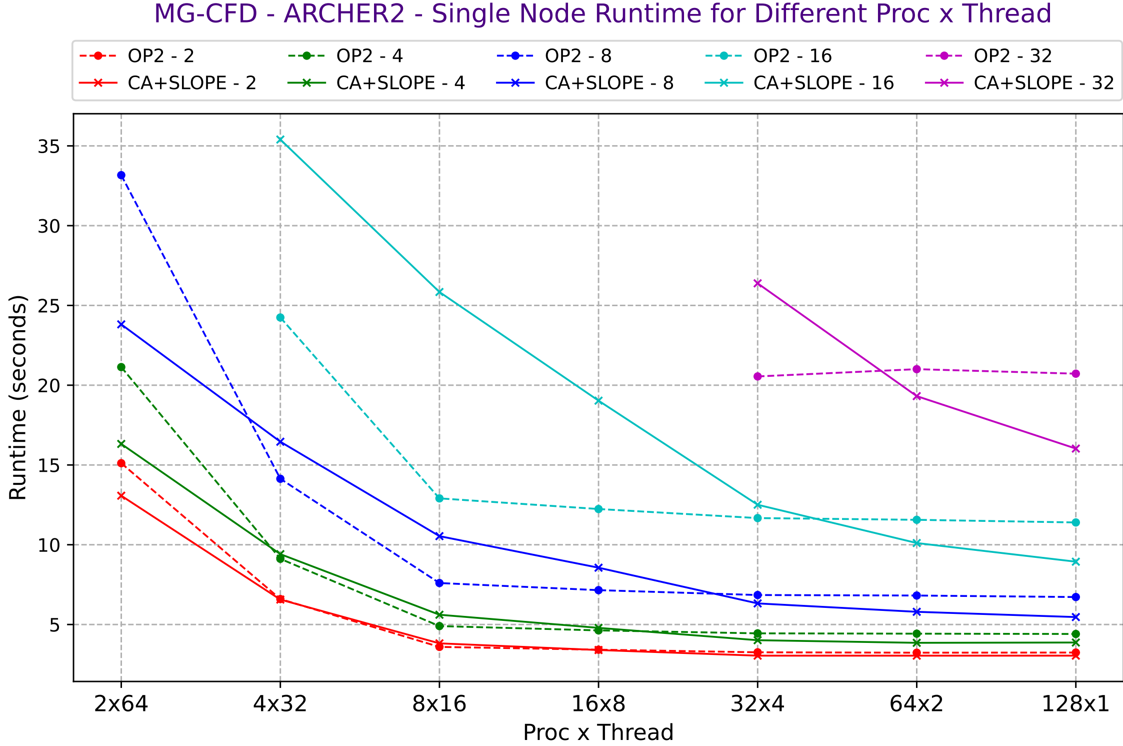


Figure 5.5: MG-CFD single node runtime variation for different Process × Thread combinations on ARCHER2 for different loop counts (⟨version⟩ - ⟨loop count⟩)[1]

performance for the OP2 and CA based MG-CFD version. However, there were instances where combinations of (i) 64 MPI processes per node and 2 OpenMP threads per MPI process and (ii) 32 MPI processes per node and 4 OpenMP threads per MPI process, surpassed the performance of the MPI-only version.

Based on these findings, we decided to continue testing the above process and thread combinations for our SDMP performance testing with MG-CFD and Hydra. Runtimes of all the tests performed in this chapter can be found in Section B.3 of Appendix B.

5.5.1 MG-CFD

We are using the same synthetic loop-chain setup as introduced in Section 4.4.1. The loop-chain is now expressed according to the SLOPE API, or *slopified*, as shown in Listing 5.1. The CA back-end integration and its communication-avoidance code changes made to the loop-chain remain unchanged.

We tested MG-CFD with the *slopified* synthetic loop-chain on ARCHER2, increasing the node count from 1 to 64. We tried two thread combinations, keeping

(i) 64 MPI processes per node and 2 OpenMP threads per MPI process and

(ii) 32 MPI processes per node and 4 OpenMP threads per MPI process.

Out of the two, we received better performance for the 64 MPI processes × 2 OpenMP threads combination. We present results for the said combination in the coming sections.

[1]Larger runtimes for loop counts 16 and 32 are excluded from the graph for clarity.

```
for (int nc = 0; nc < nchains; nc++) {
  for (int color = 0; color < ncolors; color++) {

    // for all tiles of this color
    const int n_tiles_per_color = exec_tiles_per_color(exec[l], color);

    #pragma omp parallel for
    for (int j = 0; j < n_tiles_per_color; j++) {

      tile_t* tile = exec_tile_at(exec[l], color, j);
      if (tile == NULL)
        continue;

      int loop_size;
      int tile_id = 0;

      // loop test_write_kernel
      tile_id = 2 * i + 0;
      iterations_list& le2n_0 = tile_get_local_map(tile, tile_id, "e2n");
      loop_size = tile_loop_size(tile, tile_id);

      for (int k = 0; k < loop_size; k++) {
        test_write_kernel(
          &(((double*)(p_var[l][index]->data))[le2n_0[k*2 + 0] * 5]),
          &(((double*)(p_var[l][index]->data))[le2n_0[k*2 + 1] * 5]));
      }

      // loop test_read_kernel
      tile_id =  2 * i + 1;
      iterations_list& le2n_1 = tile_get_local_map(tile, tile_id, "e2n");
      iterations_list& iterations_1 = tile_get_iterations(tile, tile_id);
      loop_size = tile_loop_size(tile, tile_id);

      for (int k = 0; k < loop_size; k++) {
        test_read_kernel(
          &(((double*)(p_var[l][index]->data))[le2n_1[k*2 + 0] * 5]),
          &(((double*)(p_var[l][index]->data))[le2n_1[k*2 + 1] * 5]),
          &(((double*)(p_edge_weights[l]->data))[iterations_1[k] * 3]),
          &(((double*)(p_fluxes[l]->data))[le2n_1[k*2 + 0] * 5]),
          &(((double*)(p_fluxes[l]->data))[le2n_1[k*2 + 1] * 5]));
      }
    }
  }
}
```

Listing 5.1: Expandable synthetic *2-loop-chain* written using SLOPE API

ARCHER2 Results

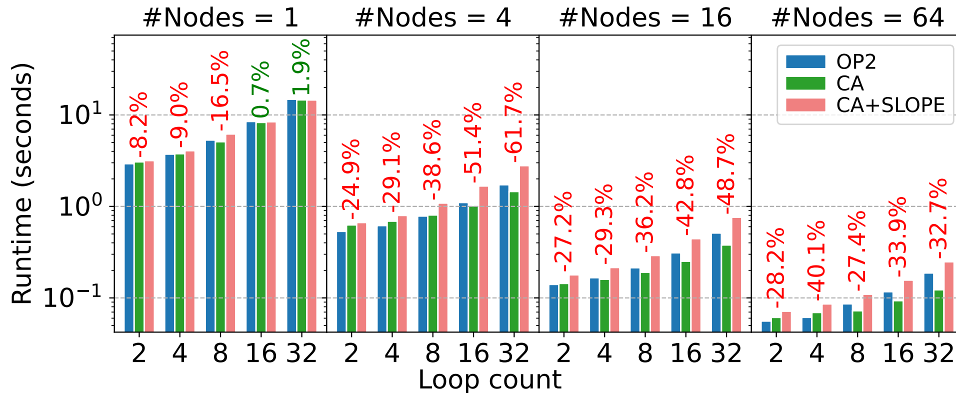


Figure 5.6: MG-CFD CA+SLOPE performance with 8M mesh on ARCHER2

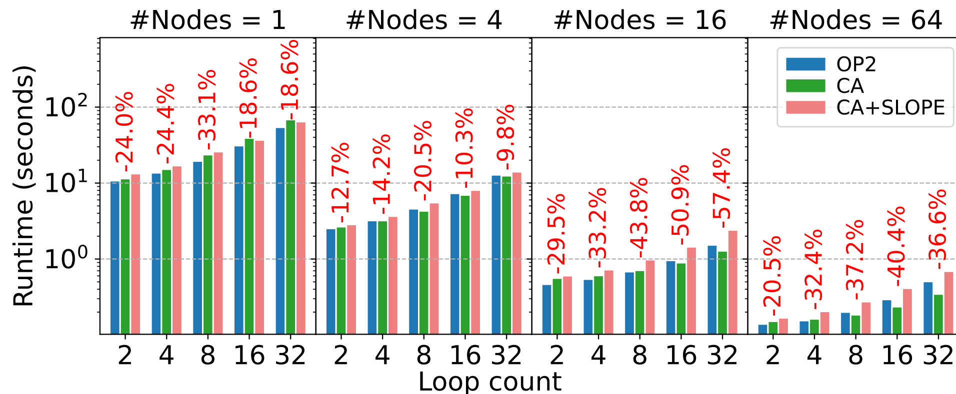
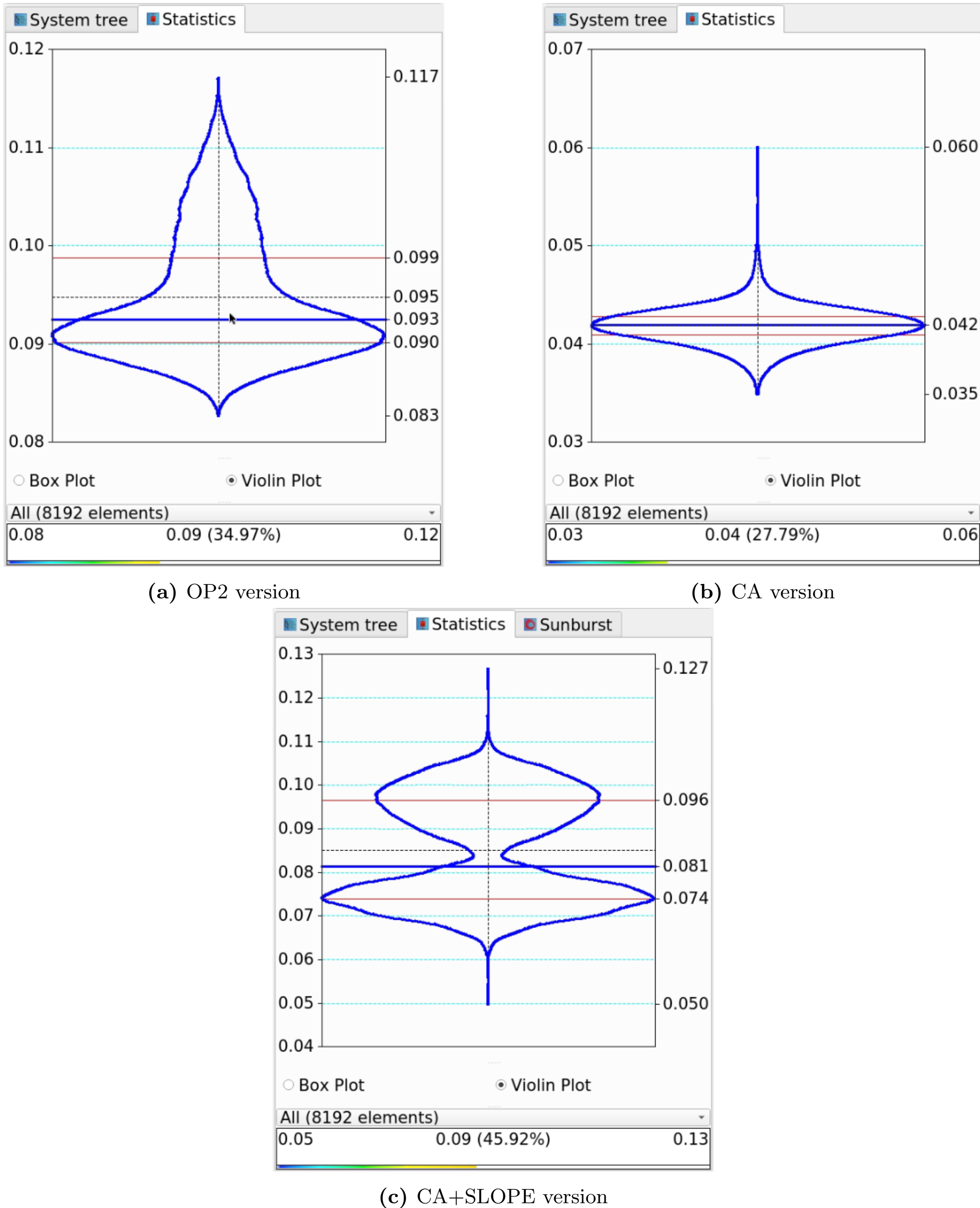


Figure 5.7: MG-CFD CA+SLOPE performance with 24M mesh on ARCHER2

Figure 5.6 and Figure 5.7 showcase the runtimes of three different versions of MG-CFD on ARCHER2: standard OP2, CA-only, and CA+SLOPE. These versions were tested on NASA Rotor 37, 8M and 24M mesh datasets. The results indicate that while the CA+SLOPE version showed only slight improvements in two test cases, it failed to outperform the best-performing versions of OP2 and CA-only in all other scenarios.

We utilized Scalasca [129] and Score-P [130] to identify the performance bottleneck in the CA+SLOPE version. Our analysis showed that the CA+SLOPE version spends a significant amount of time on the implicit barrier that follows the `#pragma omp parallel for` section in the code. To illustrate this, we present Figure 5.8a, Figure 5.8b, and Figure 5.8c, which demonstrate the time spent on the expandable loop-chain by the OP2, CA, and CA+SLOPE versions respectively for the 64 node run with a loop count of 16 on ARCHER2. The CA and the OP2 MPI versions have 8192 MPI processes and the CA+SLOPE version has 4096 MPI processes with 2 threads attached to each MPI process. The mean time values for the 8192 processing elements of each version of the three runs are 9.47×10^{-2}, 4.18×10^{-2}, and 8.51×10^{-2} in seconds. Additionally, the coefficient of variance (CoV) values for the three runs stand at 0.066, 0.042, and 0.138, respectively.



(a) OP2 version

(b) CA version

(c) CA+SLOPE version

Figure 5.8: MG-CFD execution time in seconds for synthetic loop-chain (Configurations: #Nodes - 64, #Loops - 16)

These results suggest that the CA+SLOPE version has a higher CoV for the runtime, making it more susceptible to poor performance due to delays in the execution of some threads. Our testing has shown that the CA and OP2 MPI-only versions of MG-CFD tend to perform better than the CA+SLOPE version since they do not experience thread synchronization issues at the end of the synthetic loop-chain execution.

5.5.2 OP2 Hydra

We used OP2 Hydra with the identified loop-chains illustrated in Table 3.8 to test for the CA+SLOPE combination. We were able to *slopify* only three loop-chains due to the limitation mentioned in Section 5.3.2.

The performance was benchmarked on ARCHER2, increasing the node count from 1 to 64, similar to MG-CFD. We tried two thread combinations, keeping

(i) 64 MPI processes per node and 2 OpenMP threads per MPI process and

(ii) 32 MPI processes per node and 4 OpenMP threads per MPI process.

Out of the two, we received better performance for the 64 MPI processes × 2 OpenMP threads combination. We present results for the said combination in the coming sections. We have presented the individual loop-chain runtimes in the graphs for analysis.

ARCHER2 Results

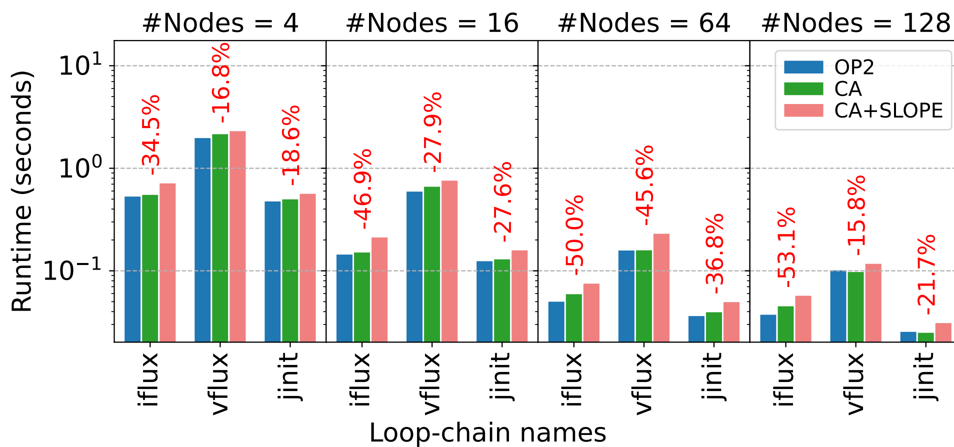


Figure 5.9: Hydra CA+SLOPE performance with 8M mesh on ARCHER2

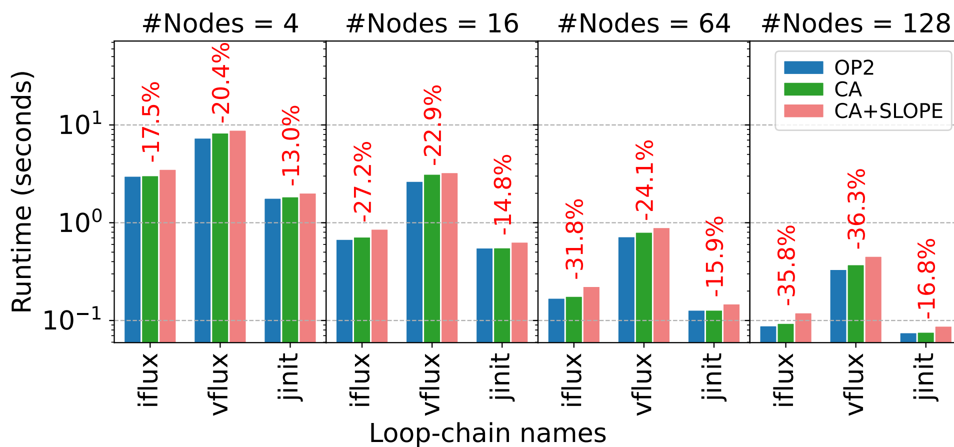


Figure 5.10: Hydra CA+SLOPE performance with 24M mesh on ARCHER2

Figure 5.9 and Figure 5.10 depict the runtimes of Hydra loop-chains on ARCHER2 for the NASA Rotor 37 dataset with 8M and 24M meshes. The three versions of Hydra tested were the standard OP2 version, the CA-only version and the CA+SLOPE version. However, the results were similar to our previous experience with MG-CFD, as the CA+SLOPE version of Hydra was not able to outperform the best-performing version out of the other two, OP2 and CA-only.

5.6 Conclusion

After analyzing the results, it seems that achieving performance improvements with SLOPE for distributed-memory parallelism with CA in the tested applications is extremely challenging. To investigate this issue, we utilized analysis tools that are available on the ARCHER2 supercomputer. Scalsca [129] profiling analysis on ARCHER2 revealed that a significant amount of time is spent on the implicit barrier after the `#pragma omp parallel for` section of the *slopified* loop-chains. This synchronization delay occurs at the end of the loop-chain, and the runtime is determined by the slowest performing thread. This lag in performance is a primary reason why an MPI+OpenMP version of an application cannot outperform the MPI-only version, despite utilizing the benefits of both shared- and distributed-memory parallelism.

Chapter 6

Integrating Shared- and Distributed-Memory Communication-Avoiding Optimizations for GPUs

Now, we will explore how well the shared- and distributed-memory parallelization (SDMP) optimizations work on GPU clusters. We will describe the OP2 implementation that supports the SDMP for GPU clusters, as well as the extension of the communication-avoidance (CA) back-end explained in Chapter 4, for GPU clusters. After that, we will compare the performance of the OP2 distributed-memory parallelized version with the new CA version that was developed for GPUs.

6.1 OP2 based SDMP for GPU Clusters

This explanation of SDMP in OP2 for GPUs is based on the OP2 open-source code base [131] and the 'OP2 Developers Guide - Distributed-Memory (MPI) Parallelisation' by Mudalige et al. [9]. The distributed-memory back end developed for a cluster of GPUs assumes that one MPI process will communicate with a single GPU. Communication between nodes is made possible through MPI messages, while CUDA is utilized for GPU computations. In cases where a single node comprises multiple GPUs, the MPI process can choose an available GPU as its computing device, making the implementation of the OP2 library back-end simple. Applications written with the OP2 API can be executed on heterogeneous platforms without modifications to the existing scientific source code. The OP2 library generates code for hybrid parallelism, and computation and communication overlap is achieved by separating mesh elements into *core* and *boundary* elements, which includes *export exec* and *import exec* elements. Only the *core* computations are performed, while the MPI processes complete the message exchange required for further

computations. The OP2 library partitions both the *core* and *boundary* regions, assigning two sets of colors to these partitions so that the execution priority of the *core* set elements is preserved. Within a region (i.e., *core* or *boundary*), partitions are assigned colors so that no two adjacent partitions have the same color, and partitions with the same color are executed in parallel. The execution mechanism of MPI and CUDA hybrids is described in detail in Algorithm 6.1.

Algorithm 6.1: MPI+CUDA halo exchange [9]

```
1  for each op_dat requiring a halo exchange do
2     Execute CUDA kernel to gather export halo data
3     Copy export halo data from GPU to host
4     Start non-blocking MPI communication
5  end for

6  for each color (i) do
7     if color ≠ core colors then
8        Wait for all MPI communications to complete
9        for each op_dat requiring a halo exchange do
10          Copy import halo data from host to GPU
11       end for
12    end if
13    Execute CUDA kernel for color (i) mini-partitions
14 end for
```

6.2 CA Back-End for GPUs

We extended our communication-avoidance back-end developed in Chapter 4 to support the GPUs. We incorporated our communication-avoidance (CA) algorithm for loop-chain execution with the GPU kernel execution through the OP2 library. In this implementation, the multi-layered halo setup and the steps in the inspection phase (from line 1 to 7) explained in Algorithm 4.3 will remain the same.

The communication-avoiding algorithm in Algorithm 4.3 has targeted only distributed-memory parallel execution. Extending the CA distributed-memory execution to a cluster of GPUs can also be carried out, given that the OP2 library generates code for CUDA with MPI. In the GPU CA version, the halos are transferred via MPI by first copying it to the host over the PCIe bus. This implementation does not utilize NVIDIA's GPUDirect [132] technology for transferring data between the GPUs. Instead, a communication pipeline is set up, allowing for maximum overlap of kernels, memcopy, communication, and core computations. We found that this performs better than GPUDirect, which often did not run simultaneously with the computing kernels [9]. Again, the multi-layered halo setup will remain the same, but an extra data copy from host to device and vice versa will occur during the halo exchange.

6.3 Performance Model Extension for CA with GPUs

We detailed an analytical performance model in Section 4.3 for the new communication-avoidance back-end that we developed in Chapter 4. Here, we expand the same model to include support for GPUs in the CA back-end.

Given that the concepts and behavior of the CPU-based communication-avoidance remain the same for the GPU-based communication-avoidance version, we only have to accommodate the extra data copy from host to device and vice versa that will occur during halo exchange, into our performance model. This cost can be approximated by a larger communication latency Λ replacing L in Equations (4.2) and (4.3). Additionally, g_l will need to be estimated for a GPU.

Then the Equations (4.1) and (4.3) will be modified as below to reflect the changes for the GPU-based CA version.

$$T_{op2,l} = MAX\left[\ g_l S_l^c,\ 2d_l p_l(\Lambda + m_l^1/B)\ \right] + g_l S_l^1 \tag{6.1}$$

$$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} g_l S_l^c,\ p(\Lambda + m^r/B + c)\right] + \sum_{l=0}^{n-1} g_l S_l^h \tag{6.2}$$

6.4 Performance

We conducted a performance test on the CA back-end developed for a GPU cluster, using the same applications, MG-CFD and Hydra, that were used to test the CA back-end on a CPU cluster. A detailed explanation of these applications can be found in Section 3.4.2 and Section 3.4.4. Our testing was carried out on the NASA Rotor 37 datasets of sizes 8M and 24M.

Performance is benchmarked on the Cirrus GPU cluster, an SGI/HPE 8600 GPU cluster at EPCC. Table 6.1 briefly details the system's key hardware and system setup. The Cirrus GPU cluster consists of 4×V100 GPUs per node configuration, each node also consisting of 2×Intel Xeon Gold 6248 (Cascade Lake) processors, each with 20 cores (40 total cores). A node has 384GB of main memory and a single V100 GPU has 16GB of global memory. For the MPI communication, MPT 2.25 was used. Runtimes of all the tests performed in this chapter can be found in Section B.4 of Appendix B.

6.4.1 MG-CFD

The details about the MG-CFD application can be found in Section 3.4.2. The synthetic loop-chains introduced in Section 4.4.1 to MG-CFD and the same loop count combinations, 2, 4, 8, 16, and 32 were used in the tests on the GPU cluster. We used the NASA Rotor 37 datasets with 8M and 24M problem sizes for testing on Cirrus. CUDA codes that are relevant to running MG-CFD with the GPU back-end are automatically gener-

Table 6.1: System specification

System	**Cirrus** [87] SGI/HPE 8600 GPU Cluster
Processor	Intel Xeon Gold 6248 (Cascade Lake) @ 2.5 GHz + NVIDIA Tesla V100-SXM2-16GB GPU
(procs×cores)/node	2×20 + 4×GPUs
Mem/node	384 GB + 40GB/GPU
Interconnect	Infiniband FDR, 54.5 Gb/s
OS	Linux CentOS 7
Compilers	nvfortran (nvhpc 21.2)
Flags	CUDA 11.6 and sm_70 -O2 -Kieee
MPI	MPT 2.25

ated without altering the scientific source of the application. We executed the problems on nodes ranging from 1 to 16, with each node having 4×NVIDIA V100 GPUs. Each GPU was allocated 1 MPI process.

Cirrus Results

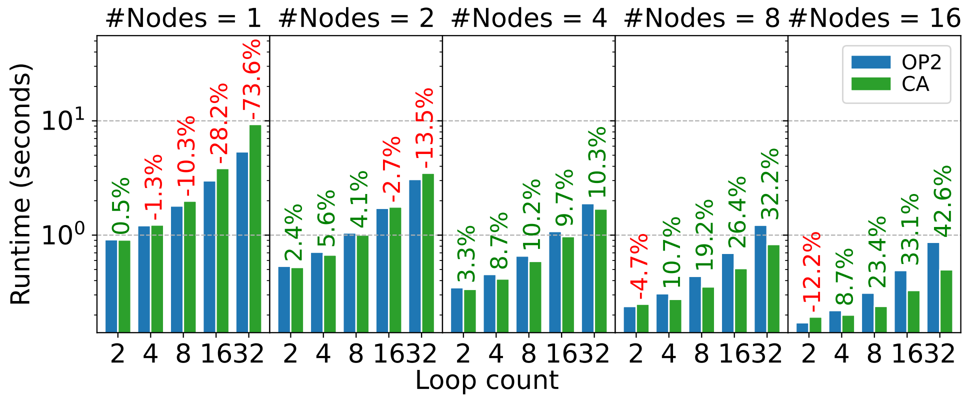


Figure 6.1: MG-CFD CA performance with 8M mesh on Cirrus

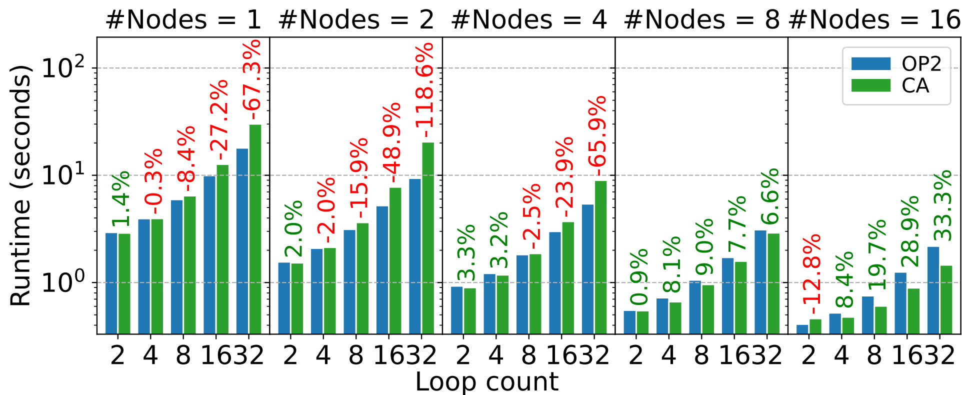


Figure 6.2: MG-CFD CA performance with 24M mesh on Cirrus

Table 6.2: MG-CFD on Cirrus - 8M Mesh - Model Components: OP2 comms ($\sum(2dpm^1)$) - CA comms (pm^r) in bytes, OP2 core iterations ($\sum(S^c)$) - CA core iterations ($\sum(S^c)$), OP2 halo iterations ($\sum(S^1)$) - CA halo iterations ($\sum(S^h)$), and performance gain% of CA over OP2

#Nodes	#Loops	8M Mesh						
		OP2			CA			Gain%
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$	
1	2	1511520	12669842	115550	2968160	12565576	339811	0.55
	4	3023040	25339684	231100	2968160	24840643	679622	-1.29
	8	6046080	50679368	462200	2968160	48784059	1359244	-10.34
	16	12092160	101358736	924400	2968160	94285662	2718488	-28.24
	32	24184320	202717472	1848800	2968160	175975342	5436976	-73.63
2	2	2891600	6476154	102140	5593800	6414004	286807	2.39
	4	5783200	12952308	204280	5593800	12592540	573614	5.62
	8	11566400	25904616	408560	5593800	24287910	1147228	4.06
	16	23132800	51809232	817120	5593800	45183143	2294456	-2.68
	32	46265600	103618464	1634240	5593800	78016632	4588912	-13.46
4	2	2425360	3201376	101282	4770360	3156436	272576	3.27
	4	4850720	6402752	202564	4770360	6138997	545152	8.69
	8	9701440	12805504	405128	4770360	11610254	1090304	10.21
	16	19402880	25611008	810256	4770360	20715755	2180608	9.73
	32	38805760	51222016	1620512	4770360	32605403	4361216	10.34
8	2	2519600	1610000	65194	4995200	1565961	201363	-4.72
	4	5039200	3220000	130388	4995200	2985327	402726	10.66
	8	10078400	6440000	260776	4995200	5447831	805452	19.19
	16	20156800	12880000	521552	4995200	9107757	1610904	26.44
	32	40313600	25760000	1043104	4995200	12839618	3221808	32.21
16	2	2625280	796106	46004	5228160	771171	141630	-12.24
	4	5250560	1592212	92008	5228160	1452591	283260	8.67
	8	10501120	3184424	184016	5228160	2583733	566520	23.44
	16	21002240	6368848	368032	5228160	4094136	1133040	33.11
	32	42004480	12737696	736064	5228160	5242007	2266080	42.57

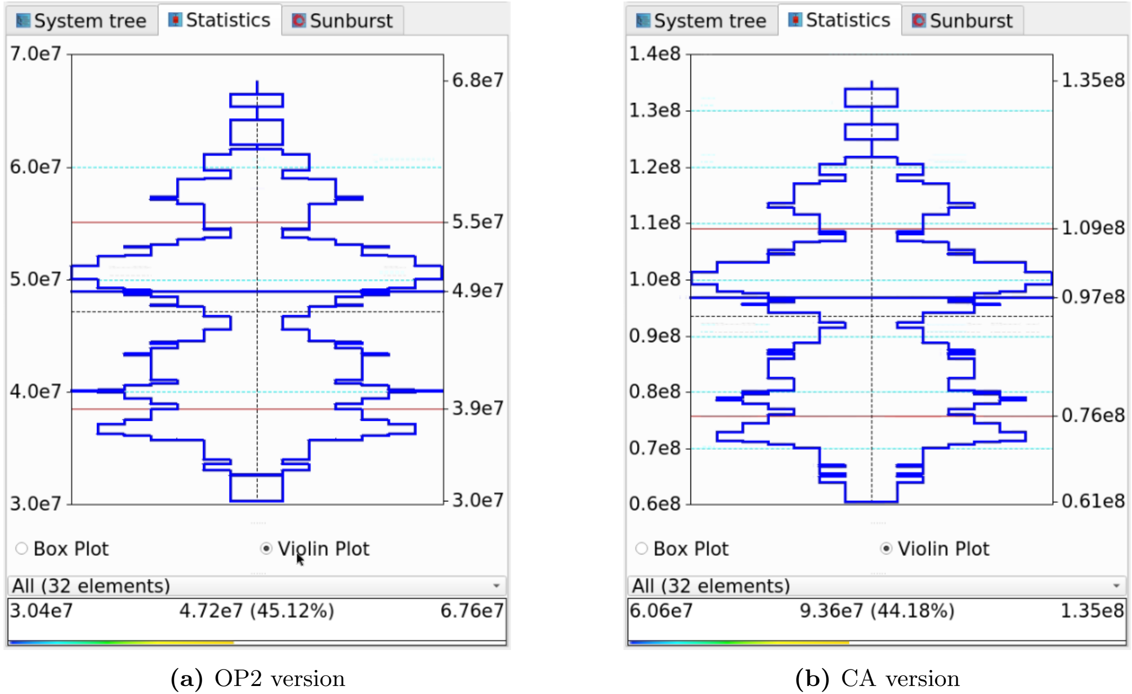
Figure 6.1 and Figure 6.2 detail the execution times (min. of at least five runs each, (CoV < 0.005)) on the GPU cluster. Based on our analysis, it is clear that both the 8M and 24M meshes show better runtimes with higher node counts when using CA on a GPU cluster. Additionally, the performance gains are even more significant for higher loop counts. These findings align with the insights from the performance model explained in Section 4.3.3. The model components in Table 6.2 and Table 6.3 indicate that the CA version can reduce the number of messages sent while keeping the message size the same for the expandable loop-chain for a given node count, resulting in better performance. We have observed up to a 42% improvement in runtimes with CA compared to the original OP2. These results show similar trends to ARCHER2 results in Section 4.4.1.

Let us consider the performance gains of the 8M mesh in node count 8 from the data in Table 6.2. The core computations for the loop count 2 in the OP2 version and the CA version are 1610000 and 1565961, respectively. The total message sizes exchanged between the processes are 2519600 and 4995200 in bytes, whereas the extra halo computations are 65194 and 201363 respectively. Based on these numbers from the performance model, it appears that the OP2 version will perform better with lesser

Table 6.3: MG-CFD on Cirrus - 24M Mesh - Model Components

#Nodes	#Loops	24M Mesh						
		OP2			CA			Gain%
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$	
1	2	3270960	44909058	231958	6433920	44673831	701926	1.36
	4	6541920	89818116	463916	6433920	88420673	1403852	-0.26
	8	13083840	179636232	927832	6433920	173254913	2807704	-8.35
	16	26167680	359272464	1855664	6433920	336929494	5615408	-27.23
	32	52335360	718544928	3711328	6433920	643729070	11230816	-67.34
2	2	4833760	22371632	238952	9599520	22235355	696083	1.95
	4	9667520	44743264	477904	9599520	43940744	1392166	-2.01
	8	19335040	89486528	955808	9599520	85814587	2784332	-15.92
	16	38670080	178973056	1911616	9599520	163554342	5568664	-48.91
	32	77340160	357946112	3823232	9599520	295830306	11137328	-118.57
4	2	5984440	11199072	219758	12012280	11077260	588295	3.29
	4	11968880	22398144	439516	12012280	21685890	1176590	3.18
	8	23937760	44796288	879032	12012280	41534880	2353180	-2.47
	16	47875520	89592576	1758064	12012280	75926241	4706360	-23.89
	32	95751040	179185152	3516128	12012280	127462695	9412720	-65.87
8	2	5237200	5588800	164622	10413200	5493891	445604	0.92
	4	10474400	11177600	329244	10413200	10645647	891208	8.09
	8	20948800	22355200	658488	10413200	20024758	1782416	9.05
	16	41897600	44710400	1316976	10413200	35499926	3564832	7.69
	32	83795200	89420800	2633952	10413200	56573598	7129664	6.58
16	2	7312640	2782282	113964	14569600	2717141	361358	-12.82
	4	14625280	5564564	227928	14569600	5200806	722716	8.44
	8	29250560	11129128	455856	14569600	9573441	1445432	19.75
	16	58501120	22258256	911712	14569600	16277318	2890864	28.93
	32	117002240	44516512	1823424	14569600	23394858	5781728	33.27

communication and computations than the CA version while hiding the latency with a higher number of core computations compared to the CA version. The empirical results in Figure 6.1 also show that the OP2 version is better with 2 loop counts in node count 8. However, as the loop count increases in node count 8, the CA version starts to perform better. At loop count 16, the core computations in the OP2 version and the CA version are 12880000 and 9107757, respectively. The OP2 version requires 521552 extra halo computations and exchanges 20156800 bytes between processes, while the CA version requires 1610904 extra halo computations and exchanges 4995200 bytes. Although the OP2 version has higher core iterations to hide latency and lower extra halos to compute, the amount of communication it has to make is higher compared to the CA version. With this performance model information, we can predict that this higher communication reduction in the CA version will lead to a higher performance gain than the OP2 version at loop count 16 on 8 nodes. This explained behavior can be seen in other higher node and loop counts as well, which can be justified by our analytical model. Similar behavior can be observed with the 24M mesh on Cirrus as shown in Table 6.3.



(a) OP2 version **(b)** CA version

Figure 6.3: MG-CFD MPI data exchange in bytes for the synthetic loop-chain (Configurations: #Nodes - 8, #Loops - 2)

```
Time (%)  Total Time (ns)  Avg (ns)  Min (ns)  Max (ns)    Operation
--------  ---------------  --------  --------  ---------  ------------------
    75.3       40,334,928  77,716.6     1,344  3,803,436  [CUDA memcpy HtoD]
    24.7       13,206,910  43,159.8     1,504     95,871  [CUDA memcpy DtoH]


Total (MB)  Avg (MB)  Min (MB)  Max (MB)      Operation
----------  --------  --------  --------  ------------------
   281.828     0.543     0.000    18.238  [CUDA memcpy HtoD]
   149.925     0.490     0.001     0.664  [CUDA memcpy DtoH]
```

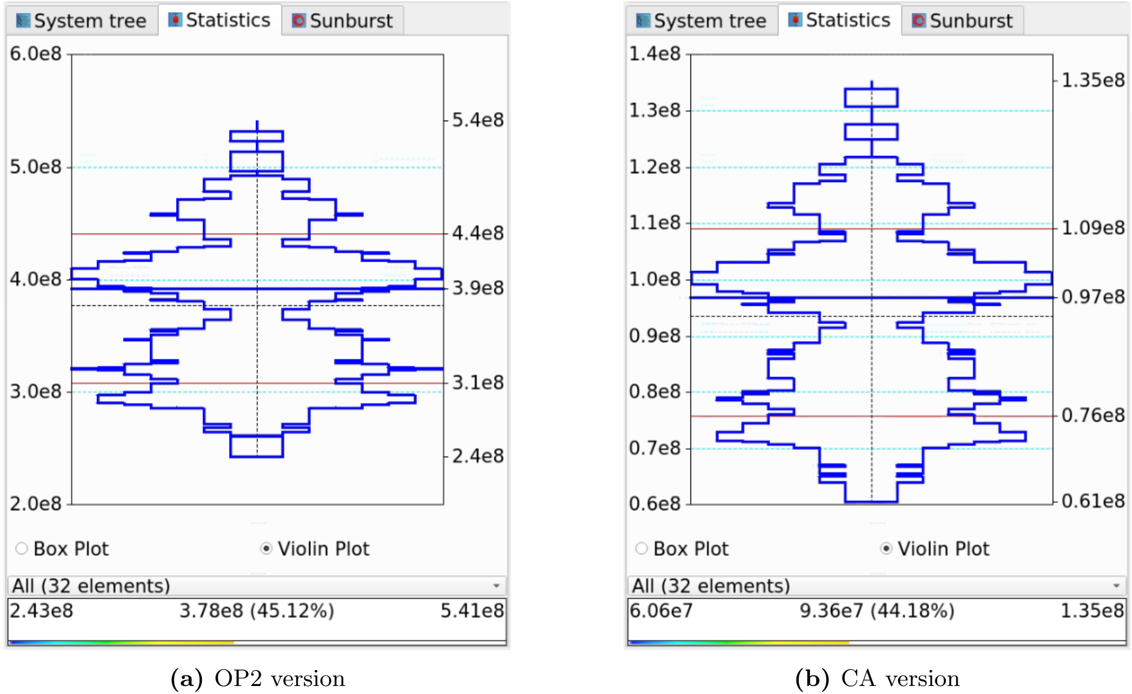
(a) OP2 version

```
Time (%)  Total Time (ns)  Avg (ns)  Min (ns)  Max (ns)    Operation
--------  ---------------  --------  --------  ---------  ------------------
    78.5       48,014,718  91,631.1     1,248  4,183,777  [CUDA memcpy HtoD]
    21.5       13,150,272  42,974.7     1,408    215,423  [CUDA memcpy DtoH]


Total (MB)  Avg (MB)  Min (MB)  Max (MB)      Operation
----------  --------  --------  --------  ------------------
   311.089     0.594     0.000    19.845  [CUDA memcpy HtoD]
   150.642     0.492     0.001     0.994  [CUDA memcpy DtoH]
```

(b) CA version

Figure 6.4: MG-CFD HtoD and DtoH data exchange in bytes for the synthetic loop-chain in a single process. (Configurations: #Nodes - 8, #Loops - 2)



(a) OP2 version

(b) CA version

Figure 6.5: MG-CFD MPI data exchange in bytes for the synthetic loop-chain (Configurations: #Nodes - 8, #Loops - 16)

```
Time (%)  Total Time (ns)  Avg (ns)  Min (ns)  Max (ns)     Operation
--------  ---------------  --------  --------  ---------  ------------------
    67.3       50,611,628  48,478.6     1,280  3,421,377  [CUDA memcpy HtoD]
    32.7       24,620,521  29,627.6     1,440     56,575  [CUDA memcpy DtoH]


Total (MB)  Avg (MB)  Min (MB)  Max (MB)     Operation
----------  --------  --------  --------  ------------------
   426.011     0.408     0.000    17.126  [CUDA memcpy HtoD]
   304.116     0.366     0.001     0.405  [CUDA memcpy DtoH]
```

(a) OP2 version

```
Time (%)  Total Time (ns)  Avg (ns)  Min (ns)  Max (ns)     Operation
--------  ---------------  --------  --------  ---------  ------------------
    75.0       51,199,748  97,709.4     1,216  4,047,743  [CUDA memcpy HtoD]
    25.0       17,047,154  55,709.7     1,408    210,846  [CUDA memcpy DtoH]


Total (MB)  Avg (MB)  Min (MB)  Max (MB)     Operation
----------  --------  --------  --------  ------------------
   327.713     0.625     0.000    19.525  [CUDA memcpy HtoD]
   169.407     0.554     0.001     1.124  [CUDA memcpy DtoH]
```

(b) CA version

Figure 6.6: MG-CFD HtoD and DtoH data exchange in bytes for the synthetic loop-chain in a single process. (Configurations: #Nodes - 8, #Loops - 16)

To verify the results and analytical model explanation further, we utilized different frameworks and tools such as Scalasca [129], Score-P [130], and NVIDIA Nsight [133]. The profiling data obtained from Scalasca and Score-P in Figure 6.3 indicates that the OP2 version involves fewer MPI communications compared to the CA version (mean values: OP2 – 4.72×10^7 bytes, CA – 9.36×10^7 bytes). Additionally, the NVIDIA Nsight statistics output in Figure 6.4 shows that the OP2 version has fewer HtoD (OP2 – 281.828 MB, CA – 311.089 MB) and DtoH (OP2 – 149.925 MB, CA – 150.642 MB) data exchanges than the CA version for the loop count 2 on 8 nodes. Our analytical model supports the notion that the OP2 version will outperform the CA version in this scenario, which is confirmed by these empirical results and profiling data.

On the other hand, the Scalasca and Score-P profiling information in Figure 6.5 shows that the CA version involves fewer MPI communications compared to the OP2 version (mean values: OP2 – 3.78×10^8 bytes, CA – 9.36×10^7 bytes). Similarly, the NVIDIA Nsight statistics output in Figure 6.6 indicates that the CA version has fewer HtoD (OP2 – 426.011 MB, CA – 327.713 MB) and DtoH (OP2 – 304.116 MB, CA – 169.407 MB) data exchanges than the OP2 version for the loop count 16 in node count 8. These findings indicate that the CA version is expected to perform better than the OP2 version. The analytical model insights explained before align with the profiling data, and the empirical results confirm them.

To summarize, we have observed a notable enhancement in runtimes – up to 42% with CA, compared to the original OP2. These results exhibit similar trends as the ARCHER2 results mentioned in Section 4.4.1.

6.4.2 OP2 Hydra

We used the OP2 version of Hydra that is explained in Section 3.4.4 and its identified loop-chains as illustrated in Table 4.4 and Table 4.5 to test the CA back-end developed for GPUs. NASA Rotor 37 datasets of size 8M and 24M were used, similar to CA back-end testing on ARCHER2. Test problems of OP2 Hydra range from a single node to 16 nodes, each having a single MPI process attached to a GPU, totaling 4 MPI processes per node to cater to the 4×V100 GPUs in a Cirrus computing node.

Cirrus Results

Figure 6.7 and Figure 6.8, show the execution times on the Cirrus GPU cluster (min. of at least five runs each, (CoV < 0.07)). These times represent the cumulative time taken by each loop-chain for 20 iterations of the main time-stepping loop. We used Hydra's default partitioner, which is based on the recursive inertial bisection of the mesh for all these experiments. During our analysis, we collected information on message sizes, core and halo sizes, the number of neighboring processes, and other parameters through the CA back-end inspection phase. We will use the data in Table 6.4 and Table 6.5 for our analysis with the analytical model.

The use of CA leads to significantly greater performance gains, particularly with loop-chains such as `iflux`, `vflux`, and `jacob`, which perform up to 31%, 42%, and 68% faster on the node counts benchmarked. For loop-chains `iflux` and `vflux`, the CA version does not show any communication or computation reduction compared to the OP2 version, as shown in Table 6.4 and Table 6.5. However, in the CA version, we send an aggregated message to neighboring processes and a single data exchange from HtoD and DtoH for a loop-chain, which reduces communication overheads arising from message exchanges and makes the CA version faster than the OP2 version. In the `jacob` loop-chain, we reduce a redundant message in the CA version, with no increase in computation, which results in lower MPI and host-device communication, thereby making the loop-chain perform better. In the `gradl` loop-chain, there is an increase in communication and computation in the CA version, as shown in Table 6.4 and Table 6.5. Nonetheless, we observe performance gains in the CA version. This is mainly due to the reduction in the number of messages exchanged and the number of host-device communication initiations. The overheads of MPI message initiation and host-device communication initiation are reduced in the CA version, thereby providing performance benefits.

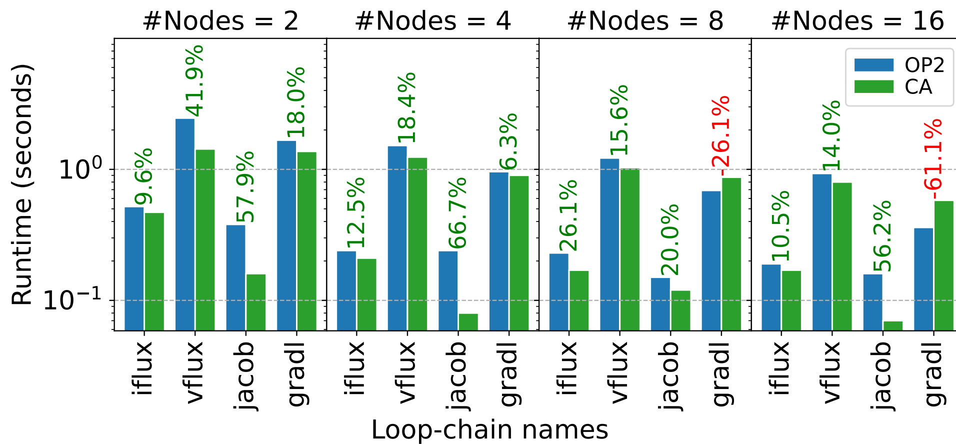


Figure 6.7: Hydra CA performance with 8M mesh on Cirrus

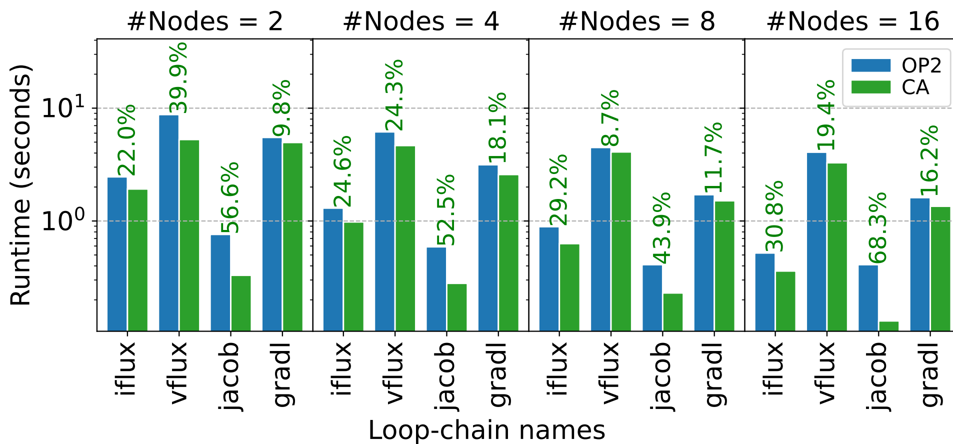


Figure 6.8: Hydra CA performance with 24M mesh on Cirrus

Table 6.4: Hydra loop-chains (LCs) on Cirrus - 8M Mesh - Model Components

LC(#Loops)	#Nodes	8M Mesh OP2			CA			Loop-chain Gain%	Comm Reduction %	Comp Increase %
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$			
iflux(2)	2	2110080	4329931	51070	2110080	4329931	51070	9.61	0	0
	4	2101440	2142362	50641	2101440	2142362	50641	12.48	0	0
	8	2951424	1078221	32597	2951424	1078221	32597	26.09	0	0
	16	2148120	533985	23002	2148120	533985	23002	10.50	0	0
vflux(2)	2	23914240	4329931	51070	23914240	4329931	51070	41.87	0	0
	4	23816320	2142362	50641	23816320	2142362	50641	18.41	0	0
	8	33449472	1078221	32597	33449472	1078221	32597	15.57	0	0
	16	24345360	533985	23002	24345360	533985	23002	13.98	0	0
jacob(3)	2	35871360	43919	38173	18287360	43919	38173	57.89	49.02	0
	4	35724480	23702	21135	18212480	23702	21135	66.66	49.02	0
	8	50174208	15286	12344	25579008	15286	12344	20.01	49.02	0
	16	36518040	9570	8819	18617040	9570	8819	56.24	49.02	0
gradl(2)	2	21100800	3247872	89243	40948800	3247650	350370	17.96	-94.06	74.53
	4	21014400	1605737	71776	41804160	1605594	276585	6.27	-98.93	74.05
	8	29514240	807639	44941	58803840	807477	192585	-26.08	-99.24	76.66
	16	21481200	399196	31821	42680880	399090	134260	-61.10	-98.69	76.30

Table 6.5: Hydra loop-chains (LCs) on Cirrus - 24M Mesh - Model Components

LC(#Loops)	#Nodes	24M Mesh OP2			CA			Loop-chain Gain%	Comm Reduction %	Comp Increase %
		$\sum(2dpm^1)$	$\sum(S^c)$	$\sum(S^1)$	pm^r	$\sum(S^c)$	$\sum(S^h)$			
iflux(2)	2	5864832	14943786	119476	5864832	14943786	119476	21.95	0	0
	4	6329856	7484548	109879	6329856	7484548	109879	24.62	0	0
	8	3926736	3737781	82311	3926736	3737781	82311	29.23	0	0
	16	3926208	3737781	82311	3926208	3737781	82311	30.77	0	0
vflux(2)	2	66468096	14943786	119476	66468096	14943786	119476	39.91	0	0
	4	71738368	7484548	109879	71738368	7484548	109879	24.35	0	0
	8	44503008	3737781	82311	44503008	3737781	82311	8.71	0	0
	16	44497024	3737781	82311	44497024	3737781	82311	19.41	0	0
jacob(3)	2	99702144	116254	69602	50828544	116254	69602	56.59	49.02	0
	4	107607552	61854	37518	54858752	61854	37518	52.54	49.02	0
	8	66754512	34375	21196	34031712	34375	21196	43.90	49.02	0
	16	66745536	34375	21196	34027136	34375	21196	68.29	49.02	0
gradl(2)	2	58648320	11197574	189078	145693200	11197258	717218	9.82	-148.42	73.64
	4	63298560	5603810	147397	125349120	5603626	601317	18.10	-98.03	75.49
	8	39267360	2794400	103507	78320880	2794400	422284	11.70	-99.46	75.49
	16	39262080	2794400	103507	78315600	2794400	422284	16.15	-99.47	75.49

In summary, the majority of loop-chains experience a speedup when utilizing CA on the Cirrus GPU cluster, as compared to the ARCHER2 CPU cluster, as shown in Section 4.4.1.

6.5 Conclusion

When analyzing the results of MG-CFD and Hydra for GPU-based communication-avoidance, it becomes apparent that higher node counts yield greater performance gains. This indicates that reducing inter-process communication can significantly increase the performance of a loop-chain. The Hydra loop-chain performance on the GPU cluster shows significant gains when compared to CPU-based communication-avoidance, as illustrated in Section 4.4.2. The loop-chains `iflux`, `vflux`, and `gradl` exhibit greater performance gains on the GPU cluster, even with fewer nodes than the ARCHER2 CPU cluster. This suggests that cutting down on host-device communications improves performance on GPU clusters. Using data from the inspection phase of the CA back-end and profiling information gathered with NVIDIA Nsight, Scalasca, and Score-P, we have demonstrated that our analytical performance model can effectively reason about loop-chain performance gains. In general, loop-chains on GPUs are more likely to benefit from communication-avoidance compared to CPU-based communication-avoidance, as there is a reduction in both host communication and host-device communication.

Chapter 7

Conclusions and Future Work

The field of high-performance computing (HPC) has experienced significant changes in recent years. The traditional method of increasing processor speed through frequency scaling has reached its limits due to high energy consumption. Instead, processors have evolved into massively parallel designs, resulting in multi-core, many-core, and heterogeneous architectures. These new designs offer hope for improved performance through parallelism.

However, these new architectures have brought new challenges, such as slow memory and network interconnections, which can limit the intended speedups. When an application runs, data moves between various memory levels of the same processor and among the various processes of the application over a network through message passing. Minimizing communication is essential to achieve performance benefits from these computing architectures.

One potential solution is communication-avoidance (CA), but it is a challenge to apply this to existing applications without altering the scientific source. The continued development of these applications is at risk due to the need to exploit the full performance of future computing systems. For decades, these applications have performed well, and to ensure uninterrupted scientific research, it is crucial to maintain their performance portability across a wider range of computing platforms.

Most real-world scientific applications are based on unstructured-meshes [8, 54, 118]. The development of frameworks to support reduced communication for these applications creates different challenges due to the irregular nature of memory accesses. Even if we developed frameworks to support these requirements, unthinkingly applying them to applications can cause performance degradation in some scenarios. Therefore, identifying the profitability of applying these techniques is also a pressing requirement in the HPC community.

7.1 Contributions and Conclusions

In this thesis, we thoroughly reviewed relevant literature in critical areas, including communication-avoidance frameworks for structured-mesh and unstructured-mesh applications. Our ultimate goal is to seamlessly apply communication-avoidance and data movement-reducing techniques to real-world, large-scale, unstructured-mesh applications utilizing the OP2 DSL.

- **Integration of shared-memory CA with OP2 (Chapter 3):** The first contribution of this thesis was to conduct a profitability analysis of utilizing the SLOPE library for shared-memory parallelism by integrating it with the OP2 DSL. Based on previous research conducted by Luporini et al. [37], the performance gains exhibited by benchmarking applications against their OpenMP implementation on Intel-based platforms were promising. In our study, we integrated the SLOPE library with the OP2 DSL and evaluated its performance using both real-world and benchmarking applications, including Airfoil, MG-CFD, Volna, and Hydra, on two Intel-based platforms (Scyrus and Telos) and one AMD-based supercomputer (ARCHER2). To enable the SLOPE library to function with Hydra, we developed a Fortran-based framework. Our findings demonstrate that we achieved better performance gains, ranging from 10% to 60%, in general for Airfoil, MG-CFD, and Hydra, across all three testing platforms. However, we only observed marginal gains for Volna on ARCHER2. These shared-memory optimizations were not previously tested on real-world, large-scale industrial applications. We tested these optimizations with Hydra, which is the largest known real-world application tested with these shared-memory optimizations which gave us more than 50 – 60% performance gains for individual loop-chains for select configurations on our testing platforms.

- **Design of a novel distributed-memory CA back-end (Chapter 4):** As the second contribution, we have developed a novel communication-avoidance (CA) back-end tailored for the OP2 DSL. Our primary emphasis is on enabling efficient distributed-memory parallel operation, and we have carefully considered methods to minimize the volume of MPI send-receive messages that occur during the execution of a series of sequential loops, known as a loop-chain. A significant advancement introduced in this back-end is its capability to seamlessly execute standard loops, alongside selectively integrated loop-chains over unstructured-meshes. This innovative approach is aimed at optimizing the overall performance of applications, providing flexibility and versatility in execution. This developed new open-source CA back-end for the OP2 library is available in [11]. We received 30 – 40% performance improvements for MG-CFD and Hydra with this CA back-end. Again, applying these distributed-memory communication-avoidance enhancements to a large-scale, real-world application like Hydra has not been previously done, according to our knowledge.

- **Analytical modeling of loop-chain performance with CA (Chapter 4, 5, and 6):** As the third contribution, the performance evaluation of loop-chains integrated with communication-avoidance (CA) is achieved through an analytical modeling process. This model is designed by analyzing the critical trade-off: it involves enhancing computations within the shared MPI halos to fulfill loop dependencies while minimizing data movement via message passing. In simple terms, it seeks to strike a balance between executing tasks within the same computational domain (shared MPI halos) to reduce the need for communication between different processing units while ensuring that loop dependencies are satisfied correctly. The analysis performed for Hydra and MG-CFD testing along with the model suggests that the model offers valuable insights into whether a specific loop-chain would benefit from the communication-avoidance (CA) capabilities provided by the new framework. In essence, it assists in making informed decisions about whether to implement CA techniques for a particular loop-chain, based on its inherent characteristics. By introducing this performance model, we tried to fulfill the need for a performance model to evaluate the CA enhancements in unstructured-mesh applications. We believe that this model can further be developed to accurately predict the runtime enhancements quantitatively.

- **Integrating shared- & distributed-memory CA (Chapter 5 and 6):** The fourth contribution is demonstrating communication-avoidance enhancements using both on-node/shared-memory and distributed-memory parallelism for unstructured-mesh-based applications. The implementation of the distributed-memory back-end is enriched by the incorporation of strategies aimed at enhancing shared-memory communication efficiency and reducing data movement. Two distinct techniques play a key role in this optimization process. Firstly, we employ *sparse tiling*, which involves the utilization of cache-blocking tiling optimization methods. This is achieved by integrating the SLOPE library [8] and leveraging OpenMP threads on multi-core CPUs. We tested this approach with MG-CFD and Hydra. We identified that due to the OpenMP thread synchronization overhead, this version is not able to surpass the gains we achieved through our new CA back-end. Secondly, when our system operates within a cluster of GPUs, we strategically employ reduced MPI message passing through our CA back-end. This approach is designed to minimize the inherent overheads associated with GPU-to-GPU communication, efficiently harnessing the capabilities of CUDA. Again we tested this approach with MG-CFD and Hydra and yielded significant loop-chain performance gains leading to 30 – 65% of improvements for some loop-chains in Hydra.

- **Real-world, large-scale application benchmarking with CA (Chapter 3, 4, 5, and 6):** Finally, it is important to note that we are able to automatically generate code for all the mentioned communication-avoidance optimizations for any application written using the OP2 API, without altering the original source

code. These communication-avoiding optimizations were applied to two non-trivial applications, including the OP2 version of Rolls Royce's production CFD application, Hydra on problem sizes representative of real-world workloads. The results indicate that the new communication-avoiding back-end provides between $30 - 65\%$ runtime reductions for the loop-chains in these applications on both an HPE Cray EX system and an NVIDIA V100 GPU cluster.

7.2 Thesis Limitations and Future Work

Upon careful review of the contributions of the thesis, we have identified a number of extensions to the communication-avoiding optimizations used for unstructured-mesh applications with the OP2 DSL. Our proposed future work is closely linked to the identification and resolution of the limitations of the thesis. Proactively addressing these limitations will not only expand our research scope but also establish a strong foundation for future research, contributing to a more comprehensive understanding of the thesis.

7.2.1 Analytical Model Enhancements and Further Validations

Our analytical model effectively determines whether implementing communication-avoiding (CA) techniques will benefit loop-chains, regardless of the platform used. However, we have not quantitatively predicted the performance gains that applying CA techniques will provide on a specific system. As a direction for future research and development, leveraging this analytical model to its full potential offers several promising opportunities. Firstly, focusing on the quantitative prediction of performance gains remains a significant challenge. Enhancing the model to provide precise estimates of speedup or runtime reduction on a particular system is critical. This can involve refining the model's algorithms and incorporating more granular platform-specific data, such as cache latencies, memory bandwidth, and processor architecture details. By doing so, researchers can offer a more concrete and actionable understanding of the benefits that CA techniques can bring to specific computing environments.

Furthermore, extending the model to encompass a broader spectrum of CA techniques and optimizing strategies is valuable. Different CA methods, such as loop transformation and loop tiling with the SLOPE library, may yield varying performance improvements depending on the nature of the application and the underlying hardware. Evaluating a wider array of CA techniques within the model can help developers choose the most effective strategies for their specific use cases.

Additionally, conducting empirical validation by applying the model to more real-world applications and hardware platforms can offer valuable insights and benchmarks. This empirical validation can help refine the model's accuracy and utility in practical scenarios.

7.2.2 CA Back-End Memory Enhancements

When using communication-avoidance enhancements with distributed-memory parallelism, additional halo layers must be generated to support the computations of the loop-chain during the inspection phase of the CA algorithm. The maximum halo extension possible for a loop-chain with n loops is n. However, based on our experiments and real-world code analysis, it is rare to require such extreme halo extensions for a loop-chain to apply CA optimizations, considering the data access patterns of the datasets. Typically, two levels of halo extension are sufficient for a loop-chain.

To support latency hiding of a loop-chain with n loops, we must calculate core sizes. This involves determining the number of mesh elements that can perform computations without halo information coming from other processes. The core sizes relevant to individual loops of the loop-chain gradually decrease from loop 0 to loop $n-1$. To perform this calculation, n levels of halo layers must be generated, which requires a significant amount of memory. Further, the core size calculation has to be performed considering the information of all the halo layers together.

However, we can optimize this situation by developing a mechanism to calculate effective core sizes for the loops of the loop-chain without needing the information of the halos to be exported. We can also analytically determine an effective length for a loop-chain by enhancing the model. Alternatively, we can develop a memory-efficient mechanism to store halo information in the system.

7.2.3 Automate Code Generation with Lazy Evaluation

The practice of generating code with lazy evaluation or delayed evaluation [36, 134], involves creating code or performing computations only when required. This technique is commonly utilized in functional programming languages, where expressions are assessed only when necessary. The purpose of lazy evaluation is to optimize performance and resource utilization by avoiding upfront code generation or computation, particularly in situations where it would be ineffective or repetitive. In our present workflow, loop-chain data is provided to the OP2 code generator via a configuration file, allowing communication-avoidance optimizations to be integrated as elaborated in Section 4.2.8.

Lazy evaluation can significantly enhance performance by avoiding unnecessary code generation or computation until it is genuinely required. This approach conserves system resources by generating code or computing values only when necessary, thereby reducing memory and CPU usage. By utilizing lazy evaluation, software systems can dynamically adjust to changing conditions or user requests without the burden of precomputing everything. Overall, generating code with lazy evaluation is a powerful technique for optimizing software systems and improving their efficiency by deferring code generation and computation until the point of need and applying this to CA code generation will greatly benefit applications.

7.2.4 SLOPE Evaluation for Distributed-Memory Parallelism

In our evaluation of the SLOPE library, we tested its capabilities for both shared- and distributed-memory parallelism. We found that the library performed better in terms of shared-memory parallelism improvements. However, when compared to the MPI communication-avoidance version of the same application, the performance gains with distributed-memory parallelism were not as promising. To determine the reasons behind this, we used profiling tools and performance counters of the respective systems, employing frameworks such as Scalasca [129], Score-P [130], and NVIDIA Nsight [133]. We believe that further performance benefits can be achieved by utilizing cache-blocking tiling enhancements with the SLOPE library in a distributed system, by tuning various parameters such as partitioning mechanism, tile size, and loop fusion scheme. However, this requires analysis with more applications, as the performance analysis of SLOPE on a distributed system is not fully understood and explained in this thesis.

7.2.5 Evaluation Scope

In order to further our research, we intend to broaden the testing range of the CA framework to encompass additional large-scale production applications that possess larger datasets. Furthermore, we will conduct a series of benchmarking exercises on the application utilizing CA enhancements across a variety of hardware, including cloud computing architectures. This comprehensive evaluation will provide us with the ability to assess the framework's versatility and adaptability, which may lead to the discovery of new insights and optimizations that can be advantageous for a vast array of large-scale production applications.

* *

The computing capabilities of modern computer systems are rapidly advancing through the use of massively parallel designs. To fully realize the performance benefits of these systems, there is a continuous effort to improve programming models, languages, and compiler technologies. However, the development of memory and system interconnects is not keeping pace, creating challenges and bottlenecks to achieving optimal performance. As researchers, we are dedicated to finding solutions to these challenges and achieving the desired high performance. We hope that our communication-avoiding framework and analysis will help the HPC community in achieving high performance across a variety of computing systems with unstructured-mesh-based applications.

Bibliography

[1] Karl Rupp. 42 Years of Microprocessor Trend Data, 2018. URL https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[2] NVIDIA. NVIDIA Tesla V100 GPU architecture, 2017. URL http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed Nov 2022.

[3] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 117–128, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307437. doi: 10.1145/1989493.1989508.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.*, 43(6): 101–113, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375595.

[5] Lester Kalms, Tim Hebbeler, and Diana Göhringer. Automatic OpenCL Code Generation from LLVM-IR Using Polyhedral Optimization. PARMA-DITAM '18, page 45–50, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364447. doi: 10.1145/3183767.3183779.

[6] Gihan Mudalige, Mike Giles, I.Z. Reguly, C. Bertolli, and Paul Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. pages 1–12, 05 2012. ISBN 978-1-4673-2632-2. doi: 10.1109/InPar.2012.6339594.

[7] Michelle Mills Strout, Fabio Luporini, Christopher D. Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J. Ramanujam, and Paul H.J. Kelly. Generalizing Run-Time Tiling with the Loop Chain Abstraction. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1136–1145, 2014. doi: 10.1109/IPDPS.2014.118.

[8] Fabio Luporini. *Automated Optimization of Numerical Methods for Partial Differential Equations*. PhD dissertation, Imperial College London, 2016.

[9] Gihan R. Mudalige, Mike Giles, and Istvan Reguly. OP2 Developers Guide - Distributed Memory (MPI) Parallelisation, 2013. URL https://op-dsl.github.io/docs/OP2/mpi-dev.pdf.

[10] Suneth Dasantha Ekanayake, István Zoltán Reguly, Fabio Luporini, and Gihan Ravideva Mudalige. Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 380–391, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708435. doi: 10.1145/3605573.3605604. URL https://doi.org/10.1145/3605573.3605604.

[11] OP2 GitHub repository (CA Branch), 2023. URL https://github.com/OP-DSL/OP2-Common/tree/feature/mpi_comm_avoid.

[12] SLOPE GitHub repository (CA Branch), 2022. URL https://github.com/coneoproject/SLOPE/tree/feature/comm_avoid.

[13] Airfoil GitHub repository (SLOPE Branch), 2021. URL https://github.com/OP-DSL/OP2-Common/tree/perf/airfoil_noRMS.

[14] MG-CFD-app-OP2 GitHub repository (SLOPE Branch), 2021. URL https://github.com/warwick-hpsc/MG-CFD-app-OP2/tree/feature/slope.

[15] Volna GitHub repository (SLOPE Branch), 2020. URL https://github.com/reguly/volna/tree/feature/slope.

[16] MG-CFD-app-OP2 GitHub repository (CA Branch), 2023. URL https://github.com/warwick-hpsc/MG-CFD-app-OP2/tree/feature/mpi_comm_avoid.

[17] Intel. Intel Processor Product Specifications, 2022. URL https://ark.intel.com/content/www/us/en/ark.html#@Processors.

[18] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture UNPRECEDENTED ACCELERATION AT EVERY SCALE, 2022.

[19] ORNL. Frontier, 2022. URL https://www.olcf.ornl.gov/frontier/.

[20] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[21] J. J. Dongarra. The LINPACK Benchmark: An Explanation. In *Proceedings of the 1st International Conference on Supercomputing*, page 456–474, Berlin, Heidelberg, 1988. Springer-Verlag. ISBN 0387189912.

[22] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Ap-*

plications, 30(1):3–10, 2016. doi: 10.1177/1094342015593158. URL https://doi.org/10.1177/1094342015593158.

[23] Top 500, Accessed Aug 2023. URL https://www.top500.org/.

[24] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. J. Kelly, G. Mudalige, B. Van Straalen, and S. Williams. Loop Chaining: A Programming Abstraction for Balancing Locality and Parallelism. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 375–384, 2013. doi: 10.1109/IPDPSW.2013.68.

[25] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008. doi: 10.1109/IPDPS.2008.4536305.

[26] Craig Douglas, Jonathan Hu, Markus Kowarschik, and Ulrich Rüde. Cache Optimization For Structured And Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis (ETNA)*, 10, 07 1999.

[27] Christopher Krieger, Michelle Strout, Jonathan Roelofs, and Amanreet Bajwa. Executing Optimized Irregular Applications Using Task Graphs Within Existing Parallel Models. pages 261–268, 11 2012. ISBN 978-1-4673-6218-4. doi: 10.1109/SC.Companion.2012.43.

[28] Aditya M. Deshpande and Jeffrey T. Draper. Modeling Data Movement in the Memory Hierarchy in HPC Systems. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, page 158–161, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336048. doi: 10.1145/2818950.2818972.

[29] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.*, 40(5):603–612, May 1991. ISSN 0018-9340. doi: 10.1109/12.88484.

[30] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. Ca-svm: Communication-avoiding support vector machines on distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 847–859, 2015. doi: 10.1109/IPDPS.2015.117.

[31] Abhinav Sarje, Sukhyun Song, Douglas Jacobsen, Kevin Huck, Jeffrey Hollingsworth, Allen Malony, Samuel Williams, and Leonid Oliker. Parallel Performance Optimizations on Unstructured Mesh-based Simulations. *Procedia Computer Science*, 51:2016–2025, 12 2015. doi: 10.1016/j.procs.2015.05.466.

[32] Penporn Koanantakool, Alnur Ali, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Leonid Oliker, Katherine A. Yelick, and Sang-Yun Oh. Communication-Avoiding Optimization Methods for Distributed Massive-Scale Sparse Inverse Covariance Estimation. In Amos J. Storkey and Fernando Pérez-Cruz, editors, *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, volume 84 of *Proceedings of Machine Learning Research*, pages 1376–1386. PMLR, 2018. URL http://proceedings.mlr.press/v84/koanantakool18a.html.

[33] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing Communication in Sparse Matrix Solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587448. doi: 10.1145/1654059.1654096. URL https://doi.org/10.1145/1654059.1654096.

[34] Matthew G. Knepley, Michael Lange, and Gerard J. Gorman. Unstructured Overlapping Mesh Distribution in Parallel, 2015. URL https://arxiv.org/abs/1506.06194.

[35] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical Overlapped Tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 207–218, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312066. doi: 10.1145/2259016.2259044. URL https://doi.org/10.1145/2259016.2259044.

[36] István Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, 2018. doi: 10.1109/TPDS.2017.2778161.

[37] Fabio Luporini, Michael Lange, Christian T. Jacobs, Gerard J. Gorman, J. Ramanujam, and Paul H. J. Kelly. Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling. *ACM Trans. Math. Softw.*, 45(2), May 2019. ISSN 0098-3500. doi: 10.1145/3302256.

[38] Michelle Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining Performance Aspects of Irregular Gauss-Seidel Via Sparse Tiling. volume 2481, 08 2002. ISBN 978-3-540-30781-5. doi: 10.1007/11596110_7.

[39] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse tiling for stationary iterative methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, 2004. doi: 10.1177/1094342004041294.

[40] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 111–120, 1991. doi: 10.1145/125826.125893.

[41] M. Wolfe. More iteration space tiling. In *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, 1989. doi: 10.1145/76263.76337.

[42] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[43] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, USA, 2000. ISBN 0792379330.

[44] Roger Kowalewski. *Partial aggregation for collective communication in distributed memory machines*. PhD dissertation, Ludwig-Maximilians-Universität München, August 2021. URL http://nbn-resolving.de/urn:nbn:de:bvb:19-286102.

[45] Penporn Koanantakool. *Communication Avoidance for Algorithms with Sparse All-to-all Interactions*. PhD dissertation, University of California, Berkeley, 2017.

[46] Sven Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software (ICMS'10)*, LNCS 6327, pages 299–302. Springer-Verlag, 2010.

[47] Vincent Loechner. PolyLib: A library for manipulating parameterized polyhedra, 1999. URL https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz.

[48] Doran K. Wilde. A Library for Doing Polyhedral Operations. Technical Report 785, IRISA, December 1993.

[49] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM J. Scientific Computing*, 34, 01 2012. doi: 10.1137/080731992.

[50] Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. Communication-Avoiding Symmetric-Indefinite Factorization. *SIAM Journal on Matrix Analysis and Applications*, 35(4):1364–1406, 2014. doi: 10.1137/130929060.

[51] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal*

of Physics: Conference Series, 180(1):012037, Jul 2009. doi: 10.1088/1742-6596/180/1/012037.

[52] Laura Grigori, James W. Demmel, and Hua Xiang. CALU: A Communication Optimal LU Factorization Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011. doi: 10.1137/100788926.

[53] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. Communication-avoiding QR decomposition for GPUs. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 48–58. IEEE, 2011.

[54] James W. Demmel, Laura Grigori, Ming Gu, and Hua Xiang. Communication Avoiding Rank Revealing QR Factorization with Column Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015. doi: 10.1137/13092157X.

[55] James Demmel, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Michael Mahoney. Prospectus for the Next LAPACK and ScaLAPACK Libraries: Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC). Technical Report 297, ICL-UT-20-07, 2020/07 2020.

[56] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Graph expansion analysis for communication costs of fast rectangular matrix multiplication. In Guy Even and Dror Rawitz, editors, *Design and Analysis of Algorithms*, pages 13–36, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[57] Edgar Solomonik, Abhinav Bhatele, and James Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, November 2011. LLNL-CONF-491442.

[58] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures*, pages 245–252, 2009.

[59] Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. Efficient Mesh Management in Firedrake Using PETSc DMPlex. *SIAM Journal on Scientific Computing*, 38(5):S143–S155, 2016. doi: 10.1137/15M1026092.

[60] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. 43(3), Dec 2016. ISSN 0098-3500. doi: 10.1145/2998441.

[61] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J.

Gorman. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.*, 46(1), Apr 2020. ISSN 0098-3500. doi: 10.1145/3374916.

[62] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3):1165–1187, 2019. doi: 10.5194/gmd-12-1165-2019. URL https://www.geosci-model-dev.net/12/1165/2019/.

[63] Leigh Lapworth. HYDRA-CFD: A Framework for Collaborative CFD Development. 07 2004.

[64] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. ISSN 0360-0300. doi: 10.1145/197405.197406.

[65] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[66] M J Wolfe. *Optimizing supercompilers for supercomputers.* PhD dissertation, Champaign, IL, USA, 1982.

[67] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[68] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGPLAN Not.*, 50(4):429–443, Mar 2015. ISSN 0362-1340. doi: 10.1145/2775054.2694364.

[69] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM*, 61(1):106–115, Dec 2017. ISSN 0001-0782. doi: 10.1145/3150211.

[70] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[71] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with

distributed-memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 375–386, 2013. doi: 10.1109/PACT.2013.6618833.

[72] Uday Bondhugula. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. SC '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323789. doi: 10.1145/2503210.2503289.

[73] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed Halide. *SIGPLAN Not.*, 51(8), Feb 2016. ISSN 0362-1340. doi: 10.1145/3016078.2851157.

[74] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-Stream: A Parametric High Level Compiler. In *Proceedings of HPEC*, 01 2006.

[75] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, pages 7 pp.–, 2006. doi: 10.1109/IPDPS.2006.1639500.

[76] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations. 40(2), 2014. ISSN 0098-3500. doi: 10.1145/2566630.

[77] Michael Lange, Christian Jacobs, Fabio Luporini, Lawrence Mitchell, David Ham, and Gerard Gorman. Seigen: Seismic modelling through code generation. 10 2016. URL https://figshare.com/articles/poster/Seigen_Seismic_modelling_through_code_generation/3798984.

[78] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing, 1966. URL https://dominoweb.draco.res.ibm.com/reports/Morton1966.pdf. Accessed Nov 2022.

[79] David Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück [In German]. *Mathematische Annalen*, 38(3):459–460, 1891. doi: 10.1007/BF01199431.

[80] Gihan R. Mudalige, Istvan Z. Reguly, Arun Prabhakar, Dario Amirante, Leigh Lapworth, and Stephen A. Jarvis. Towards Virtual Certification of Gas Turbine Engines With Performance-Portable Simulations. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 206–217, 2022. doi: 10.1109/CLUSTER51413.2022.00034.

[81] Istvan Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H.J. Kelly, and David Radford. Acceleration of a Full-Scale Industrial CFD Application with OP2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1265–1278, May 2016. ISSN 2161-9883. doi: 10.1109/tpds.2015.2453972.

[82] M. Giles, G. Mudalige, and I. Reguly. OP2 Airfoil Example. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1270b05f5e4f7c580bd30fb4ced7ded938056625.

[83] Andrew Owenson, Steven A. Wright, Richard Bunt, Yoon Ho, Matthew Street, and Stephen Jarvis. An Unstructured CFD Mini-Application for the Performance Prediction of a Production CFD Code. *Concurrency and Computation: Practice and Experience*, 32(10), May 2020. ISSN 1532-0626. doi: 10.1002/cpe.5443. © 2019 The Authors.

[84] Denys Dutykh, Raphaël Poncet, and Frédéric Dias. The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation. *European Journal of Mechanics - B/Fluids*, 30(6):598–615, 2011. ISSN 0997-7546. doi: https://doi.org/10.1016/j.euromechflu.2011.05.005. URL https://www.sciencedirect.com/science/article/pii/S0997754611000574. Special Issue: Nearshore Hydrodynamics.

[85] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12): 1901–1909, 1966. doi: 10.1109/PROC.1966.5273.

[86] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sep 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. URL https://doi.org/10.1109/TC.1972.5009071.

[87] Cirrus. Cirrus, Accessed Aug 2022. URL https://www.cirrus.ac.uk/.

[88] NVIDIA. NVIDIA ADA LOVELACE PROFESSIONAL GPU ARCHITECTURE, 2023. URL https://images.nvidia.com/aem-dam/en-zz/Solutions/technologies/NVIDIA-ADA-GPU-PROVIZ-Architecture-Whitepaper_1.1.pdf. Accessed Sept 2023.

[89] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4): 65–76, Apr 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

[90] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[91] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL http://www.cs.virginia.edu/stream/. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[92] P. Jarzebski, K. Wisniewski, and R. Taylor. On parallelization of the loop over elements in FEAP. *Computational Mechanics*, 56, 05 2015. doi: 10.1007/s00466-015-1156-z.

[93] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. ISSN 0167-8191. doi: https://doi.org/10.1016/0167-8191(96)00024-5. URL https://www.sciencedirect.com/science/article/pii/0167819196000245.

[94] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30218-6.

[95] Intel MPI library, Accessed Aug 2022. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html.

[96] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. IPDPS'06, page 84, USA, 2006. IEEE Computer Society. ISBN 1424400546.

[97] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. C: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. doi: https://doi.org/10.1002/cpe.1631. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631.

[98] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, and Alex Aiken. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[99] Michael Wolfe and Chau-Wen Tseng. The Power Test for Data Dependence. *IEEE Trans. Parallel Distributed Syst.*, 3:591–601, 1992.

[100] Leslie Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2):

83–93, Feb 1974. ISSN 0001-0782. doi: 10.1145/360827.360844. URL https://doi.org/10.1145/360827.360844.

[101] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-Affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 185–194, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326704. doi: 10.1145/2581122.2544141. URL https://doi.org/10.1145/2581122.2544141.

[102] Wayne Anthony Kelly. *Optimization within a Unified Transformation Framework*. PhD dissertation, University of Maryland, College Park, MD 20742, 1996.

[103] Chun Chen. Polyhedra Scanning Revisited. *SIGPLAN Not.*, 47(6):499–508, Jun 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254123. URL https://doi.org/10.1145/2345156.2254123.

[104] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral Code Generation in the Real World. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, pages 185–201, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33051-6.

[105] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Principles and Pratice of Parallel Programming, PPoPP'91*, volume Volume 26, pages Pages 39 – 50, Williamsburg, Virginia,, United States, April 1991. doi: 10.1145/109626.109631. URL https://hal-mines-paristech.archives-ouvertes.fr/hal-00752774. 12 pages.

[106] Aravind Sukumaran Rajam, Luis Esteban Campostrini, Juan Manuel Martinez Caamano, and Philippe Clauss. Speculative Runtime Parallelization of Loop Nests: Towards Greater Scope and Efficiency. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 245–254, 2015. doi: 10.1109/IPDPSW.2015.10.

[107] Chandan G. Introduction to Polyhedral Model, 2013. URL https://events.csa.iisc.ac.in/summerschool2013/slides/automatic-parallelization-introduction-polyhedral-models.pdf. Accessed: 2023–05-05.

[108] Didem Unat, Cy P. Chan, Weiqun Zhang, John Bell, and John Shalf. Tiling as a Durable Abstraction for Parallelism and Data Locality. United States, 2013. URL https://www.osti.gov/biblio/1407206.

[109] Tobias Gysi, Oliver Fuhrer, Carlos Osuna, Benjamin Cumming, and Thomas Schulthess. STELLA: A domain-specific embedded language for stencil codes on

structured grids. In *EGU General Assembly Conference Abstracts*, EGU General Assembly Conference Abstracts, page 8464, May 2014.

[110] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. 48(6):519–530, Jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL https://doi.org/10.1145/2499370.2462176.

[111] Matthias Christen, Olaf Schenk, and Yifeng Cui. Patus for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press. ISBN 9781467308045.

[112] Uday Bondhugula. Automatic Distributed Memory Code Generation using the Polyhedral Framework. Technical Report IISc-CSA-TR-2011-3, Indian Institute of Science, Bangalore, 2011.

[113] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.*, 22, 2012.

[114] Chris Lattner and Vikram Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[115] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[116] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simburger, Armin Grosslinger, and Louis-Noël Pouchet. Polly – Polyhedral optimization in LLVM. 2012.

[117] P. I. Crumpton and M. B. Giles. Multigrid Aircraft Computations Using the OPlus Parallel Library. In N. Satofuka A. Ecer, J. Periaux and S. Taylor, editors, *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pages 339–346. North-Holland, 1996.

[118] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, Katherine Yelick, P Balaji, et al. ASCR programming challenges for exascale computing. In *Report of the 2011 Workshop on Exascale Programming Challenges*, 2011.

[119] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 39–50, 2014.

[120] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. ICS '12, page 311–320, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313162. doi: 10.1145/2304576.2304619. URL https://doi.org/10.1145/2304576.2304619.

[121] SLOPE GitHub repository, 2022. URL https://github.com/coneoproject/SLOPE.

[122] François Pellegrini. *Scotch and PT-Scotch Graph Partitioning Software: An Overview*. 01 2012. ISBN 978-1-4398-2735-2. doi: 10.1201/b11644-15.

[123] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, September 1998.

[124] George Karypis, Kirk Schloegel, and Vipin Kumar. PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. 01 1997. URL https://conservancy.umn.edu/handle/11299/215345.

[125] ARCHER2. ARCHER2, 2022. URL https://www.archer2.ac.uk. Accessed Nov 2022.

[126] Istvan Z. Reguly, Andrew M. B. Owenson, Archie Powell, Stephen A. Jarvis, and Gihan R. Mudalige. Under the Hood of SYCL – An Initial Performance Analysis with An Unstructured-Mesh CFD Application. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*, page 391–410, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-78712-7. doi: 10.1007/978-3-030-78713-4_21.

[127] I. Z. Reguly, D. Giles, D. Gopinathan, L. Quivy, J. H. Beck, M. B. Giles, S. Guillas, and F. Dias. The VOLNA-OP2 tsunami code (version 1.5). *Geoscientific Model Development*, 11(11):4621–4635, 2018. doi: 10.5194/gmd-11-4621-2018. URL https://gmd.copernicus.org/articles/11/4621/2018/.

[128] MG-CFD-app-OP2 GitHub repository (Main Branch), 2023. URL https://github.com/warwick-hpsc/MG-CFD-app-OP2.

[129] Scalasca. Scalasca, Scalable Performance Analysis of Large-Scale Applications, 2023. URL https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/docs/manual/intro.html. Accessed on September 2023.

[130] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011, editor = Brunst, Holger and Müller, Matthias S. and Nagel, Wolfgang E. and Resch, Michael M.*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31476-6.

[131] OP2 GitHub repository (Main Branch), 2023. URL https://github.com/OP-DSL/OP2-Common.

[132] NVIDIA. NVIDIA GPUDirect, 2023. URL https://developer.nvidia.com/gpudirect.

[133] NVIDIA. NVIDIA Nsight Systems, 2023. URL https://developer.nvidia.com/nsight-systems.

[134] Peter Henderson and James H. Morris. A Lazy Evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, page 95–103, New York, NY, USA, 1976. Association for Computing Machinery. ISBN 9781450374774. doi: 10.1145/800168.811543. URL https://doi.org/10.1145/800168.811543.

Appendix A

Analytical Performance Model and Extensions

A.1 Model Parameters

Table A.1: Model Parameters

Symbol	Parameter
δ	Size of a data element of an `op_dat`, d (in bytes)
γ_l	Compute time for one iteration of the loop, l with OpenMP threads
Λ	Latency in a GPU cluster
B	Bandwidth
c	Message packing and unpacking cost
d_l	Number of `op_dats` in loop, l
g_l	Compute time for one iteration of the loop, l
h_l	Halo extension for loop, l
L	Latency
m^r	Maximum grouped message size (in bytes) sent to each neighbor with r halo layers
m_l^1	Maximum message size (in bytes) sent to a neighbor by loop, l with a single halo extension
r	Maximum number of halo layers
S_l^1	Execute halo number of iterations in loop, l
S_l^c	Number of *core* iterations, in loop, l
S_d^{eeh,h_l}	*eeh* size of `op_dat`, d in level, h_l
S_d^{enh,h_l}	*enh* size of `op_dat`, d in level, h_l
$T_{ca,\mathbb{L}}$	Total CA runtime of the full loop-chain, $\mathbb{L}$
$T_{op,\mathbb{L}}$	Total OP2 runtime of the full loop-chain, $\mathbb{L}$
$T_{op2,l}$	Time taken by an OP2 loop, l

A.2 Model Equations

Table A.2: Model Equations for CA in CPUs

Equation	No.
$T_{op2,l} = MAX\left[\ g_l S_l^c,\ 2 d_l p_l (L + m_l^1/B)\ \right] + g_l S_l^1$	(4.1)
$T_{op2,\mathbb{L}} = \sum_{l=0}^{n-1} T_{op2,l}$	(4.2)
$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} g_l S_l^c,\ p(L + m^r/B + c)\right] + \sum_{l=0}^{n-1} g_l S_l^h$	(4.3)
$m^r = \sum_{l=0}^{n-1} \left(\sum_{d=0}^{d_l-1} (S_d^{eeh,h_l} + S_d^{enh,h_l}) \times \delta \right)$	(4.4)

Table A.3: Model Equations for CA+SLOPE in CPUs

Equation	No.
$T_{op2,l} = MAX\left[\ g_l S_l^c,\ 2 d_l p_l (L + m_l^1/B)\ \right] + g_l S_l^1$	(5.1)
$T_{op2,l} = T_{op2,\mathbb{L}} = \sum_{l=0}^{n-1} T_{op2,l}$	(4.2)
$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} \gamma_l S_l^c,\ p(L + m^r/B + c)\right] + \sum_{l=0}^{n-1} \gamma_l S_l^h$	(5.2)
$m^r = \sum_{l=0}^{n-1} \left(\sum_{d=0}^{d_l-1} (S_d^{eeh,h_l} + S_d^{enh,h_l}) \times \delta \right)$	(4.4)

Table A.4: Model Equations for CA in GPUs

Equation	No.
$T_{op2,l} = MAX\left[\ g_l S_l^c,\ 2 d_l p_l (\Lambda + m_l^1/B)\ \right] + g_l S_l^1$	(6.1)
$T_{op2,l} = T_{op2,\mathbb{L}} = \sum_{l=0}^{n-1} T_{op2,l}$	(4.2)
$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} g_l S_l^c,\ p(\Lambda + m^r/B + c)\right] + \sum_{l=0}^{n-1} g_l S_l^h$	(6.2)
$m^r = \sum_{l=0}^{n-1} \left(\sum_{d=0}^{d_l-1} (S_d^{eeh,h_l} + S_d^{enh,h_l}) \times \delta \right)$	(4.4)

Appendix B

Runtimes of Benchmarked Applications

B.1 Chapter 3 Runtimes

B.1.1 Airfoil Runtimes

Airfoil on Scyrus

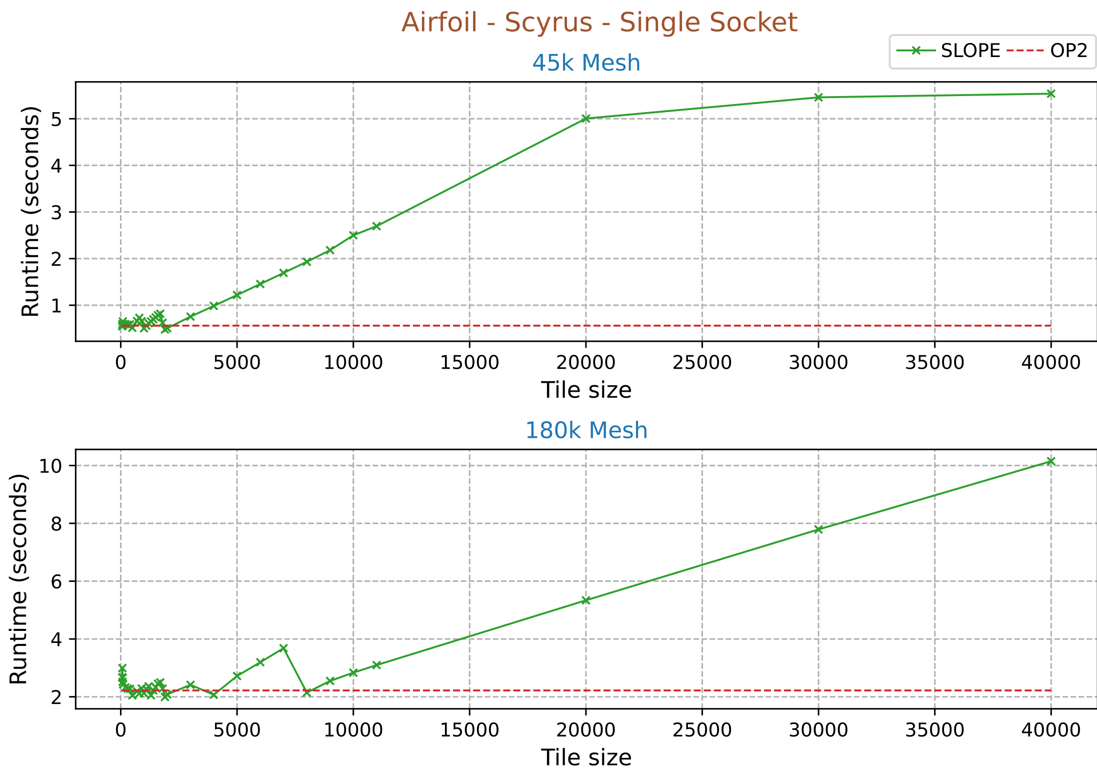


Figure B.1: Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: single socket, 12 threads)

Table B.1: OP2 Airfoil single socket runtimes on Scyrus (Figure 3.5 and Figure B.1 runtimes in seconds)

Dataset	45k	180k	720k	2880k
Runtime	0.561775	2.220399	12.935843	56.355947

Table B.2: SLOPE Airfoil single socket runtimes on Scyrus (Figure 3.5 and Figure B.1 runtimes in seconds)

Tile Size	Dataset			
	45k	180k	720k	2880k
10	0.714785	4.190064	19.750397	80.882672
20	0.610721	3.455958	17.228653	69.129627
30	0.581380	3.376573	17.921530	76.310450
40	0.577666	2.811221	14.770937	60.431816
50	0.560892	2.886324	15.269426	61.041949
60	0.547484	2.674847	14.049804	56.384127
70	0.595857	2.993242	15.938952	56.686419
80	0.561616	2.518239	12.821020	52.298359
90	0.650726	2.661099	13.362161	54.002008
100	0.583686	2.431420	12.293901	49.826369
200	0.594954	2.324475	10.998508	43.465889
300	0.581575	2.285609	10.481665	42.722016
400	0.589496	2.275106	10.210875	40.691182
500	0.515577	2.049839	10.412590	41.698055
700	0.660148	2.189034	10.564054	42.313412
800	0.730214	2.115431	10.157988	39.183951
900	0.656346	2.289435	10.453009	40.628714
1000	0.505558	2.123638	9.930281	40.065023
1100	0.561643	2.266564	10.577793	40.083843
1200	0.612298	2.363878	9.726265	39.296631
1300	0.654401	2.049866	9.791468	40.969865
1400	0.698142	2.223718	9.648914	40.140013
1500	0.740457	2.329224	9.480500	39.417587
1600	0.786118	2.455126	9.796535	39.384502
1700	0.817963	2.499810	9.511664	39.772540
1800	0.623608	2.287381	9.845416	40.038697
1900	0.479601	1.993180	9.420419	39.693293
2000	0.507002	2.085001	9.697634	39.635236
3000	0.755358	2.414803	9.401034	38.316301
4000	0.985140	2.070845	9.702355	38.565953

5000	1.218848	2.719587	9.314103	39.714908
6000	1.454535	3.195992	9.325539	39.869166
7000	1.692932	3.681877	10.590664	41.485369
8000	1.931627	2.146237	9.938513	39.437245
9000	2.181366	2.549360	11.079361	40.239452
10000	2.499895	2.836063	9.625196	39.186767
20000	5.004690	5.338074	12.318517	40.241840
30000	5.458979	7.789340	10.388432	42.310622
40000	5.537685	10.148882	12.562068	43.017358
50000	5.493871	12.462985	14.830921	49.866435
60000	5.538027	14.921072	17.236441	43.412955
70000	5.498342	17.359141	19.559304	47.773494
80000	5.529213	19.740128	21.992207	52.577039
90000	5.374573	22.140423	24.450038	57.816414
100000	5.356998	22.121871	26.847816	62.478880
200000	5.510362	22.054166	51.562993	60.979093

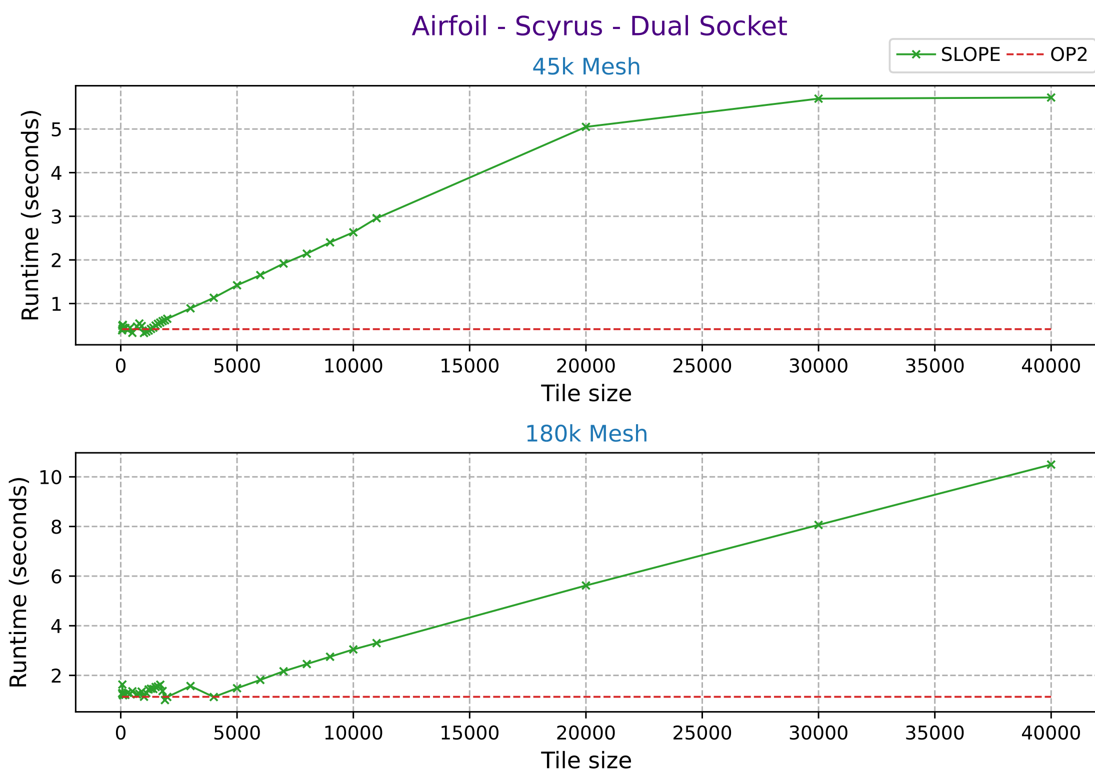


Figure B.2: Runtime variation of SLOPE Airfoil with tile sizes on Scyrus (Configurations: dual socket, 24 threads)

Table B.3: OP2 Airfoil dual socket runtimes on Scyrus (Figure 3.6 and Figure B.2 runtimes in seconds)

Dataset	45k	180k	720k	2880k
Runtime	0.412988	1.136187	10.563269	59.265646

Table B.4: SLOPE Airfoil dual socket runtimes on Scyrus (Figure 3.6 and Figure B.2 runtimes in seconds)

Tile Size	Dataset			
	45k	180k	720k	2880k
10	0.433405	1.770485	17.072581	62.349406
20	0.384686	1.404496	13.251190	55.061190
30	0.393538	1.341518	12.723296	59.151089
40	0.443276	1.230942	11.188272	49.725604
50	0.385354	1.253973	10.938642	49.686617
60	0.386365	1.279974	10.303955	47.001029
70	0.491185	1.631375	11.462427	49.708697
80	0.420674	1.229336	9.593036	43.524737
90	0.512089	1.280238	9.923822	47.607112
100	0.436825	1.197229	8.994714	44.970755
200	0.438565	1.213759	7.710096	38.093569
300	0.397200	1.297851	7.396432	35.654002
400	0.443092	1.257238	7.118014	37.414774
500	0.328596	1.357440	7.600414	40.015298
700	0.471690	1.235115	7.434603	36.807422
800	0.542792	1.257140	7.101304	34.500297
900	0.484172	1.347269	7.154988	39.025431
1000	0.324739	1.139175	7.099369	38.309201
1100	0.355050	1.290239	7.249109	37.751486
1200	0.380499	1.431875	6.551782	33.005407
1300	0.417712	1.463059	6.563885	37.618948
1400	0.447063	1.450801	6.485960	37.445905
1500	0.496285	1.531334	6.502588	34.017422
1600	0.535406	1.553923	6.532751	36.861093
1700	0.564411	1.624192	6.536165	37.799055
1800	0.596810	1.368917	6.654997	36.388806
1900	0.617261	1.005147	6.394931	36.864405
2000	0.651502	1.129851	6.447443	37.572793
3000	0.890323	1.570894	6.513751	35.392721
4000	1.132446	1.127704	6.016922	35.369937

5000	1.419000	1.483060	5.849367	32.723671
6000	1.651840	1.817098	6.484060	36.017985
7000	1.916368	2.164393	7.271735	34.969183
8000	2.143349	2.457315	6.101321	33.992531
9000	2.401822	2.750451	6.614361	34.689807
10000	2.632416	3.045109	7.218386	34.050142
20000	5.049091	5.621079	7.543451	37.688152
30000	5.696119	8.062770	10.835841	40.896651
40000	5.721782	10.494105	13.054772	40.579907
50000	5.636676	12.901638	15.275511	44.560704
60000	5.726252	15.445660	17.899137	41.031675
70000	5.639073	18.021478	20.395213	39.255162
80000	5.552789	20.503967	22.991800	38.377582
90000	5.747018	23.160678	25.385062	39.214770
100000	5.666040	23.334943	28.056769	40.495644
200000	5.668282	23.117419	53.028009	63.774301

Airfoil on Telos

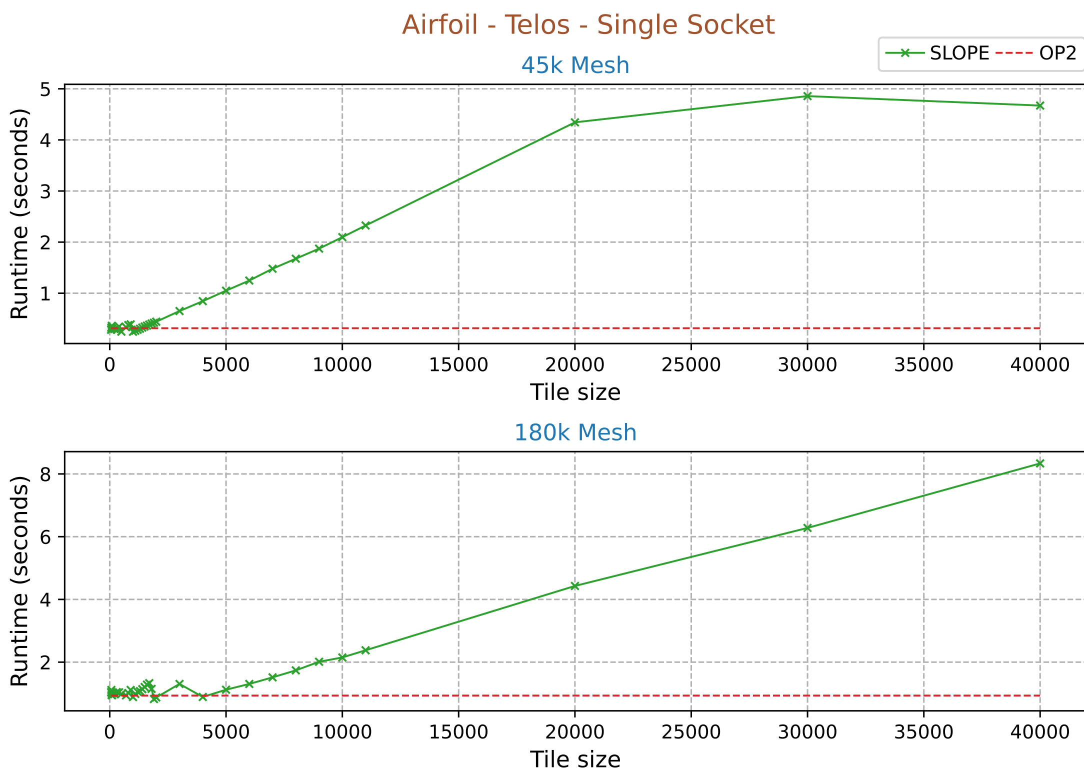


Figure B.3: Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: single socket, 24 threads)

Table B.5: OP2 Airfoil single socket runtimes on Telos (Figure 3.7 and Figure B.3 runtimes in seconds)

Dataset	45k	180k	720k	2880k
Runtime	0.314889	0.934244	6.010700	33.049485

Table B.6: SLOPE Airfoil single socket runtimes on Telos (Figure 3.7 and Figure B.3 runtimes in seconds)

Tile Size	Dataset			
	45k	180k	720k	2880k
10	0.326647	1.383191	11.281713	47.866050
20	0.284708	1.109996	8.323846	39.909846
30	0.278887	1.083859	8.584480	40.297558
40	0.293748	1.017714	6.758000	34.359125
50	0.288099	1.002145	6.762467	34.070897
60	0.284200	1.042978	6.581178	32.195019
70	0.328253	1.113048	7.167197	31.443945
80	0.306969	0.953830	5.779171	29.543800

90	0.362502	1.004215	5.914965	30.269083
100	0.306432	0.944363	5.442379	28.318925
200	0.300338	0.982150	4.863403	24.745245
300	0.288333	1.044464	4.611417	23.346048
400	0.347167	1.039411	4.593589	22.823226
500	0.247640	0.982713	4.841101	22.648141
700	0.335099	0.939781	4.662177	23.266360
800	0.376633	1.016644	4.690303	21.172355
900	0.390853	1.114022	4.635190	22.065230
1000	0.245961	0.887450	4.323784	21.663082
1100	0.265805	0.986084	5.001544	21.567178
1200	0.285607	1.053439	4.319924	21.238750
1300	0.304897	1.076292	4.173937	22.160948
1400	0.327884	1.138184	4.304943	21.803871
1500	0.348824	1.207496	4.095102	21.750731
1600	0.366333	1.282107	4.386773	21.450900
1700	0.386875	1.330398	4.346725	21.391056
1800	0.406566	1.147570	4.502188	21.765455
1900	0.422813	0.823689	4.093225	21.487715
2000	0.442973	0.874888	4.145816	21.098102
3000	0.651155	1.305060	4.330634	20.379830
4000	0.844520	0.889514	4.045517	21.014105
5000	1.050331	1.121405	3.810552	21.076505
6000	1.247852	1.304915	4.294515	20.952834
7000	1.479323	1.515155	5.184481	21.905345
8000	1.675557	1.738439	4.031838	20.757691
9000	1.872525	2.012349	4.519653	20.349526
10000	2.096996	2.149783	5.029571	20.135149
20000	4.343870	4.429180	5.342553	21.035578
30000	4.857210	6.279182	7.427241	22.014496
40000	4.672218	8.335975	9.642531	26.389965
50000	4.830525	10.332107	11.503359	30.678299
60000	4.696699	12.351136	13.481256	23.263006
70000	4.739407	14.313449	15.666895	23.975629
80000	4.648723	16.362397	17.616654	25.662047
90000	4.649929	18.472699	19.801606	27.408524
100000	4.770324	18.290331	21.875405	29.054171
200000	4.729834	18.352099	43.007626	49.790345

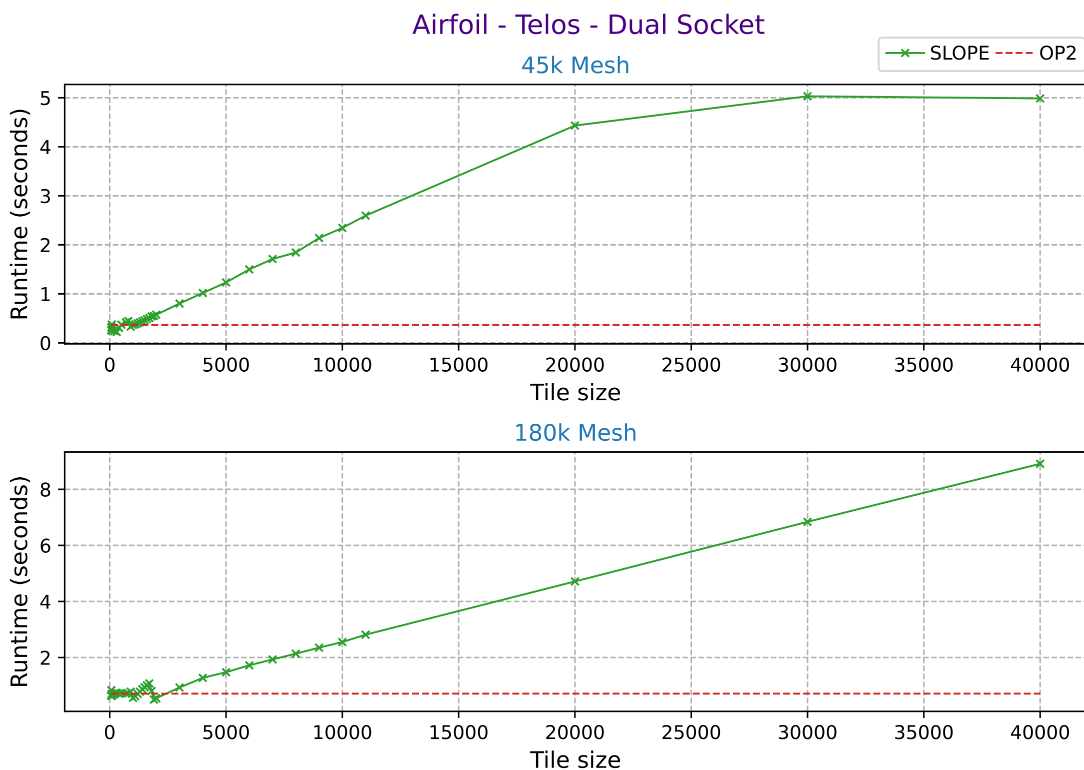


Figure B.4: Runtime variation of SLOPE Airfoil with tile sizes on Telos (Configurations: dual socket, 48 threads)

Table B.7: OP2 Airfoil dual socket runtimes on Telos (Figure 3.8 and Figure B.4 runtimes in seconds)

Dataset	45k	180k	720k	2880k
Runtime	0.364595	0.711550	2.433992	21.973881

Table B.8: SLOPE Airfoil dual socket runtimes on Telos (Figure 3.8 and Figure B.4 runtimes in seconds)

Tile Size	Dataset			
	45k	180k	720k	2880k
10	0.246403	0.719211	7.928876	39.890182
20	0.246545	0.651104	5.286274	30.738565
30	0.237736	0.647529	4.197228	27.296473
40	0.257689	0.765753	3.553281	24.898964
50	0.240236	0.639011	3.262831	23.632566
60	0.258204	0.634364	3.017306	22.645859
70	0.310798	0.835055	3.908188	19.819081
80	0.322543	0.702049	2.718799	20.610236
90	0.375375	0.629522	3.136567	20.156827
100	0.252522	0.648719	2.579825	19.294939

200	0.239801	0.685253	2.288864	17.019304
300	0.220872	0.742494	2.277544	15.914276
400	0.306701	0.713848	2.287923	15.342409
500	0.372067	0.733795	2.590767	14.725309
700	0.420619	0.707113	2.614190	15.197392
800	0.447005	0.723073	2.360668	15.139343
900	0.330242	0.780929	2.238191	14.194450
1000	0.365616	0.560747	2.503488	13.787795
1100	0.378008	0.618328	2.494869	13.300055
1200	0.398647	0.701810	2.275069	14.841986
1300	0.417747	0.782268	2.039064	14.182133
1400	0.439447	0.870220	2.230224	14.313324
1500	0.461164	0.938773	2.399918	14.686659
1600	0.486538	1.007140	2.308257	13.700386
1700	0.507062	1.076715	2.362023	13.038931
1800	0.540664	0.805510	2.445114	13.782824
1900	0.548432	0.498888	1.929516	14.346134
2000	0.574254	0.546686	2.098114	12.929859
3000	0.802666	0.930260	2.668906	12.031945
4000	1.017630	1.273634	2.063766	12.536388
5000	1.233446	1.475829	2.769259	13.072716
6000	1.498142	1.721539	2.848898	13.677677
7000	1.710935	1.931000	3.254027	15.170143
8000	1.843843	2.137402	2.097465	14.089327
9000	2.138721	2.350707	2.435492	14.982730
10000	2.344507	2.551807	2.797979	13.696222
20000	4.432708	4.713426	6.087756	16.408144
30000	5.030773	6.840138	8.127291	16.320459
40000	4.986417	8.910655	10.248862	17.183143
50000	4.879248	10.683413	12.485792	17.734345
60000	4.933151	12.915222	14.660914	21.130399
70000	4.916940	14.993983	16.807490	23.333301
80000	4.840861	16.933324	18.960504	25.219948
90000	4.944569	19.102339	21.194465	27.449257
100000	4.938915	19.174475	23.233967	28.381594
200000	4.879722	19.105937	44.591647	50.632948

Airfoil on ARCHER2

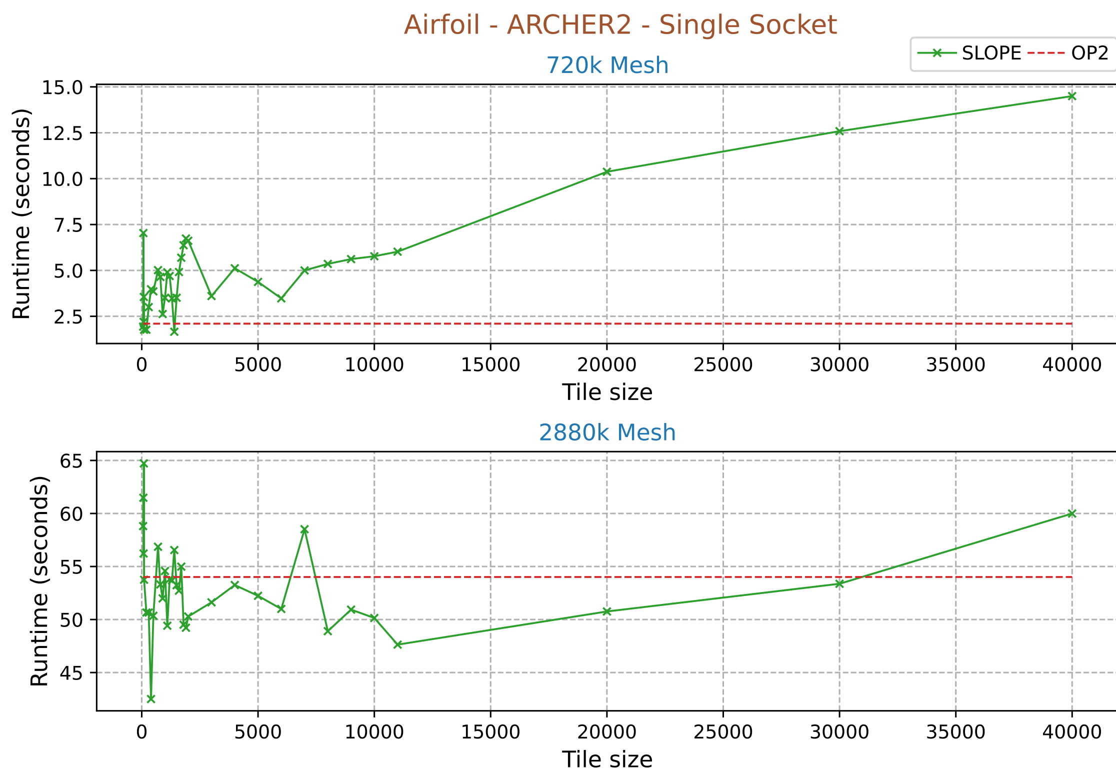


Figure B.5: Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: single socket, 64 threads)

Table B.9: OP2 Airfoil single socket runtimes on ARCHER2 (Figure 3.9 and Figure B.5 runtimes in seconds)

Dataset	720k	2880k	6480k	11520k
Runtime	2.098819	54.010133	243.103564	442.813455

Table B.10: SLOPE Airfoil single socket runtimes on ARCHER2 (Figure 3.9 and Figure B.5 runtimes in seconds)

Tile Size	Dataset			
	720k	2880k	6480k	11520k
10	2.776299	96.797528	285.042764	404.782525
20	2.097814	70.249535	241.574711	359.479268
30	1.936240	61.284793	213.178551	340.420254
40	1.860452	61.186344	209.865580	336.509560
50	1.828446	57.274016	200.169877	334.851348
60	1.939031	58.809502	204.781928	333.232781
70	7.036345	61.483404	201.729857	387.893776
80	2.196397	56.225664	195.912861	323.298345

90	3.553884	64.721229	195.508436	376.175293
100	1.764851	53.750807	190.878717	314.289960
200	1.770972	50.660311	169.203114	281.014433
300	3.001669	50.657479	162.391133	266.467724
400	3.972443	42.501758	160.132528	261.389471
500	3.858908	50.353884	155.896357	306.268432
700	5.015537	56.862163	151.696960	294.008871
800	4.645305	53.305503	148.566345	249.543516
900	2.623576	51.992299	153.673903	290.775615
1000	3.511091	54.578062	146.150419	289.808401
1100	4.900559	49.415965	149.338218	285.284003
1200	4.694448	53.813354	149.419284	242.754410
1300	3.475146	53.686826	146.643772	285.835179
1400	1.658303	56.550766	149.092366	284.871961
1500	3.511534	53.233150	146.133837	285.562455
1600	4.912663	52.719789	151.699894	240.330565
1700	5.691431	54.986408	151.683009	290.780104
1800	6.371815	49.514963	151.821948	286.371570
1900	6.738621	49.227214	149.478900	287.210583
2000	6.617326	50.286845	145.150467	282.030008
3000	3.604012	51.620310	146.602608	282.204639
4000	5.118087	53.253875	146.014791	278.761972
5000	4.374946	52.233323	142.328291	274.522582
6000	3.477047	50.998690	141.014924	264.045715
7000	4.998790	58.515646	137.038918	260.253534
8000	5.354281	48.902816	135.367101	254.606259
9000	5.617710	50.934431	130.169186	255.408677
10000	5.771688	50.145820	128.633806	247.331762
20000	10.374723	50.759350	126.663513	224.347470
30000	12.586128	53.371906	127.186634	229.229612
40000	14.497252	59.993755	145.552905	252.631524
50000	16.870494	68.355873	155.604306	282.021534
60000	18.040263	71.867263	171.112379	303.707871
70000	19.298107	72.423159	170.774450	323.682965
80000	21.172859	71.784852	178.031119	316.008810
90000	23.044332	74.289881	170.249120	321.385849
100000	23.979683	73.068051	170.710569	316.932817
200000	38.357019	83.787013	162.438671	295.685308

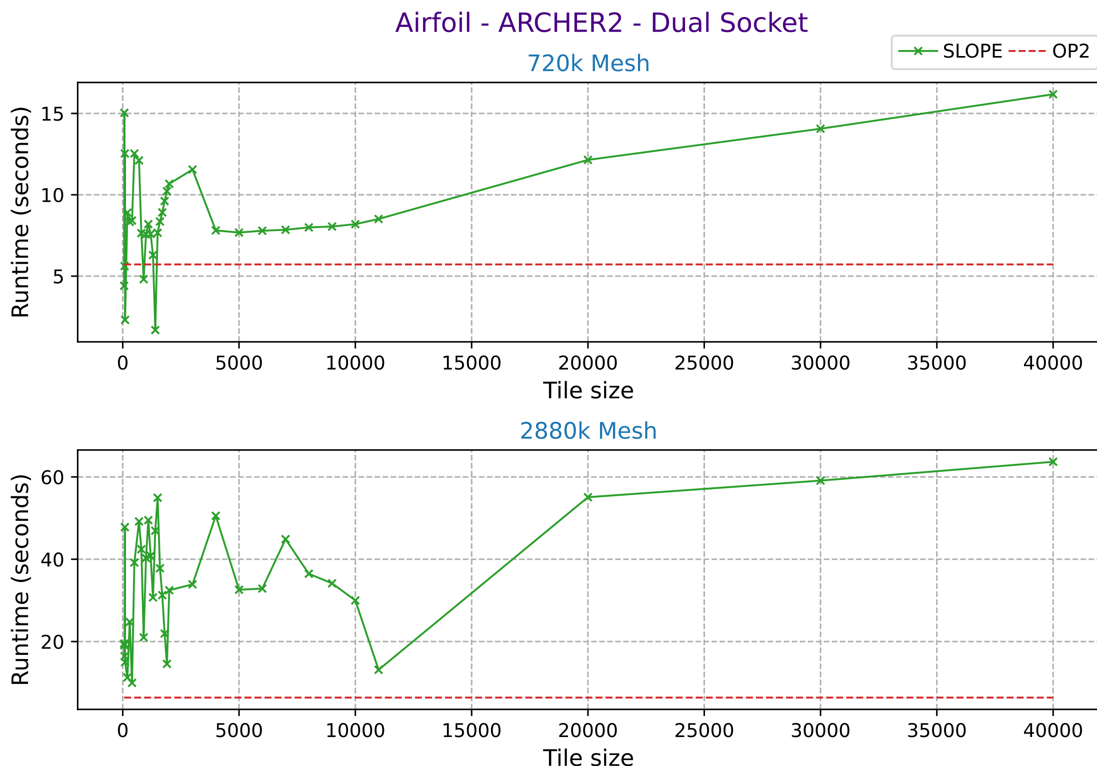


Figure B.6: Runtime variation of SLOPE Airfoil with tile sizes on ARCHER2 (Configurations: dual socket, 128 threads)

Table B.11: OP2 Airfoil dual socket runtimes on ARCHER2 (Figure 3.10 and Figure B.6 runtimes in seconds)

Dataset	720k	2880k	6480k	11520k
Runtime	5.719125	6.381400	159.464374	451.544639

Table B.12: SLOPE Airfoil dual socket runtimes on ARCHER2 (Figure 3.10 and Figure B.6 runtimes in seconds)

Tile Size	Dataset			
	720k	2880k	6480k	11520k
10	2.307857	67.336804	261.186822	423.116941
20	2.429602	41.551264	194.027518	372.866546
30	3.112655	28.983436	176.000177	337.999412
40	3.707923	23.368659	164.565760	322.758051
50	2.135930	21.154470	168.506715	326.162950
60	4.404856	19.505137	157.318391	332.268872
70	15.039407	19.106999	177.955689	452.913442
80	5.616679	16.445856	151.364931	316.506769
90	12.539012	47.795653	145.707741	385.411420
100	2.308343	14.980698	153.610612	305.208350
200	8.901996	11.275063	145.001654	277.408654

300	8.336638	24.766135	136.266640	264.191270
400	8.416379	9.956405	139.742306	258.219157
500	12.539065	39.187939	158.084911	310.529849
700	12.115432	49.182059	157.685667	290.677843
800	7.641549	42.428218	135.039467	252.425217
900	4.810070	21.014798	127.377968	295.788716
1000	7.574638	40.279520	148.432124	287.201598
1100	8.197793	49.508780	153.221279	290.615159
1200	7.594798	40.847601	138.985930	252.135329
1300	6.294267	30.717664	132.436368	292.713819
1400	1.684884	46.933277	144.980276	296.069677
1500	7.664590	54.950595	150.334532	289.685101
1600	8.358552	37.813420	158.499218	253.522992
1700	8.914331	31.274914	154.482784	286.163039
1800	9.614951	21.933444	158.096642	275.668477
1900	10.221579	14.570019	145.220865	274.618608
2000	10.674624	32.472173	148.435990	288.168530
3000	11.552667	33.889885	142.167435	282.153732
4000	7.808235	50.557685	135.192855	283.472914
5000	7.680660	32.590458	154.957468	281.690941
6000	7.790711	32.877176	126.442662	277.660433
7000	7.848474	44.883912	145.144240	264.937470
8000	7.995536	36.473220	159.290301	272.527429
9000	8.048782	34.141873	142.923058	265.009825
10000	8.193214	29.998711	136.474513	258.583432
20000	12.145644	55.078333	127.179831	241.242701
30000	14.060348	59.110585	138.834320	238.820168
40000	16.178879	63.679446	156.191807	267.728153
50000	18.741501	69.323181	160.922162	299.179294
60000	19.708379	75.088615	175.005477	344.994032
70000	21.104475	74.424369	174.753076	351.467380
80000	25.055466	74.232365	178.435701	322.097234
90000	24.457430	76.299700	175.462612	325.881928
100000	25.618732	76.201162	176.792638	323.453651
200000	40.407442	85.126102	167.208434	304.170476

B.1.2 MG-CFD Runtimes

MG-CFD on Scyrus

Table B.13: OP2 MG-CFD runtimes on Scyrus (Figure 3.14 and Figure 3.15 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Dataset	1M	8M	1M	8M
Runtime	1.760776	13.820041	1.468171	13.420122

Table B.14: SLOPE MG-CFD runtimes on Scyrus (Figure 3.14 and Figure 3.15 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Tile Size	Dataset			
	1M	8M	1M	8M
10	7.939703	62.955604	6.232795	50.578720
20	5.389059	43.396475	4.194236	33.467970
30	4.448334	35.278409	3.354948	27.602593
40	4.066493	31.385879	2.974452	25.968002
50	3.636544	28.278403	2.759377	22.914829
60	3.284215	25.754759	2.514562	20.978687
70	3.128684	24.621912	2.402059	19.563343
80	2.929191	23.017472	2.273236	19.673442
90	2.770091	21.501499	2.123871	17.755781
100	2.486191	20.428958	2.049454	16.696700
200	1.952867	15.172964	1.588995	13.184359
300	1.698048	13.069231	1.399728	11.789455
400	1.613303	12.190750	1.284641	11.074802
500	1.564664	11.589213	1.218530	10.234793
700	1.518623	11.133854	1.135140	9.190694
800	1.475565	11.022027	1.101928	9.009418
900	1.456017	10.507981	1.066310	8.821902
1000	1.428164	10.629663	1.054433	8.474080
1100	1.352805	10.700190	1.041419	8.423364
1200	1.387800	10.199983	1.020354	8.398445
1300	1.402804	10.385612	1.009819	8.316531
1400	1.397682	10.426708	0.991365	8.203343
1500	1.356989	10.443432	0.989825	7.968122
1600	1.348749	10.525191	0.971450	7.977849
1700	1.359209	10.003141	0.972574	7.769816

1800	1.323318	9.832828	0.968915	7.855542
1900	1.326283	9.934556	0.977543	7.685347
2000	1.347742	9.924918	0.964078	7.533777
3000	1.373582	9.521758	0.978464	7.170999
4000	1.373923	9.647112	0.912657	6.958864
5000	1.341486	9.492282	0.979938	6.665794
6000	1.459825	9.334325	1.060357	6.995231
7000	1.413459	9.216858	0.995943	6.949896
8000	1.474247	9.591742	1.075343	7.016371
9000	1.633316	9.760198	1.142772	7.031111
10000	1.530314	9.914950	1.111997	7.305823
20000	1.725586	11.223732	1.180524	8.594357
30000	1.917623	11.928001	1.210997	9.199701
40000	1.518152	11.279708	1.365917	8.969287
50000	1.449752	11.885534	1.436894	9.058953
60000	1.979901	11.191337	2.002135	9.024264
70000	2.027487	11.648975	2.041609	9.054110
80000	2.287114	11.196433	2.420382	9.292282
90000	2.589478	11.069466	2.715925	8.645147
100000	2.428816	11.946976	2.478976	9.685748
200000	5.903811	12.514334	6.107621	9.442719

MG-CFD on Telos

Table B.15: OP2 MG-CFD runtimes on Telos (Figure 3.16 and Figure 3.17 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Dataset	1M	8M	1M	8M
Runtime	0.721983	6.401104	0.585744	1.764256

Table B.16: SLOPE MG-CFD runtimes on Telos (Figure 3.16 and Figure 3.17 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Tile Size	Dataset			
	1M	8M	1M	8M
10	3.599551	28.780068	2.332777	21.954373
20	2.276020	19.493188	1.616972	14.081973
30	1.872717	16.093425	1.247262	12.970889
40	1.674570	13.916012	1.132111	12.032626
50	1.523361	12.650854	1.011694	10.659607
60	1.400645	11.805784	0.944210	10.079135
70	1.273348	11.042734	0.884342	7.833606
80	1.182164	10.429928	0.847787	9.362309
90	1.128697	9.740297	0.795115	9.179010
100	1.046743	9.239099	0.759705	8.707754
200	0.789475	7.109023	0.544022	7.187283
300	0.668309	6.028640	0.504018	5.879311
400	0.625215	5.382601	0.456768	5.381697
500	0.596699	5.002959	0.418088	5.291686
700	0.556176	4.730640	0.385848	4.370531
800	0.549698	4.527607	0.377431	4.229830
900	0.544051	4.570188	0.359665	4.102553
1000	0.535177	4.430808	0.349957	3.954079
1100	0.511405	4.360550	0.353950	3.847662
1200	0.509214	4.280895	0.325498	3.522612
1300	0.510633	4.249656	0.324577	3.326501
1400	0.508712	4.184378	0.311773	3.601852
1500	0.521788	4.030419	0.334201	2.579643
1600	0.499869	4.087464	0.323956	4.015107
1700	0.512031	3.941751	0.317326	3.922187
1800	0.493992	3.942140	0.316471	3.894580

1900	0.508460	3.908546	0.321262	3.872578
2000	0.500956	3.861167	0.330089	3.319634
3000	0.535601	3.602337	0.335962	3.054438
4000	0.517436	3.597788	0.308013	3.483324
5000	0.559656	3.570290	0.329872	3.392768
6000	0.631131	3.604194	0.368228	3.202515
7000	0.599854	3.662455	0.348269	3.449719
8000	0.672450	3.734479	0.337476	3.547593
9000	0.737808	3.796476	0.393885	3.533058
10000	0.701054	3.944737	0.422591	3.579572
20000	0.777937	4.913187	0.629235	4.230025
30000	0.856493	5.363671	0.832020	4.814375
40000	1.007668	5.241790	0.994504	4.762174
50000	1.076766	5.117237	1.070516	4.826534
60000	1.584168	5.292683	1.571269	5.067150
70000	1.605126	5.413499	1.604146	4.029694
80000	1.858387	5.462502	1.842085	4.279601
90000	2.116304	5.009750	2.123021	3.936481
100000	1.994557	6.089356	1.965338	4.318543
200000	5.027318	5.652018	5.054690	5.231214

MG-CFD on ARCHER2

Table B.17: OP2 MG-CFD runtimes on ARCHER2 (Figure 3.18 runtimes in seconds)

Configuration	Single Socket	
Dataset	1M	8M
Runtime	1.279166	24.685086

Table B.18: SLOPE MG-CFD runtimes on ARCHER2 (Figure 3.18 runtimes in seconds)

Configuration	Single Socket	
Tile Size	Dataset	
	1M	8M
10	3.710585	41.218753
20	2.508669	33.057660
30	2.461383	29.273404
40	2.358940	29.242021
50	2.246009	27.976216
60	2.331704	26.933452
70	2.143611	25.439981
80	1.966228	27.247368
90	2.010967	24.139178
100	1.935035	24.783447
200	1.692404	22.718691
300	1.799512	21.530610
400	1.920217	19.978984
500	1.765485	19.299961
700	1.721805	18.341839
800	1.661764	17.306678
900	1.655385	16.979023
1000	1.597075	16.709506
1100	1.562559	17.725141
1200	1.535571	16.815970
1300	1.555862	16.588503
1400	1.536940	15.776978
1500	1.540632	16.067251
1600	1.462462	15.963022
1700	1.477918	15.547380
1800	1.451671	15.564542
1900	1.551131	15.520512
2000	1.463854	15.508275

3000	1.423240	14.742529
4000	1.447099	14.685381
5000	1.371646	14.103297
6000	1.357494	13.920225
7000	1.348410	13.972873
8000	1.340421	13.652662
9000	1.334576	13.989957
10000	1.336149	14.155744
20000	1.319138	13.609494
30000	1.455471	13.744435
40000	1.625466	13.522063
50000	1.765295	13.312235
60000	2.266779	14.063736
70000	2.319472	13.886287
80000	2.611098	13.424312
90000	2.886669	13.658245
100000	2.807055	13.689275
200000	6.004063	14.138655

B.1.3 Volna Runtimes[1]

Volna on Scyrus

Table B.19: OP2 Volna runtimes on Scyrus (Figure 3.21 and Figure 3.22 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Dataset	Catalina	NU3	Catalina	NU3
Runtime	2.496774	124.809682	1.543411	89.449858

Table B.20: SLOPE Volna runtimes on Scyrus (Figure 3.21 and Figure 3.22 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Tile Size	Dataset			
	Catalina	NU3	Catalina	NU3
10	29.921846		13.933021	
20	19.501469		9.250273	7447.795986
30	9.124965	5568.861942	4.594535	3614.275555
40	4.862043	4225.478018	2.534343	2710.710986
50	4.554766	1292.571921	3.691698	843.519793
60	4.417344	881.942552	2.360177	574.917470
70	4.295775	6023.918484	2.256121	3870.777673
80	9.008719	3919.129473	2.140471	2580.356229
90	10.572906	5357.268460	5.326607	3130.967418
100	4.011382	988.939786	2.254798	640.180885
200	3.823003	539.181064	2.248541	336.843385
300	3.372189	537.730997	1.972008	348.064940
400	3.363709	474.924598	1.863129	306.905193
500	3.015037	499.933385	1.780696	317.521603
700	3.187168	204.383188	1.891337	129.164990
800	3.119536	201.850173	1.665280	200.954898
900	3.123099	376.147761	1.868140	243.646002
1000	2.843973	204.567597	1.948652	133.199018
1100	2.993942	202.958643	2.083368	132.152273
1200	3.164456	210.542395	2.109166	134.774208
1300	3.278477	206.861241	2.227893	139.061513
1400	3.471352	309.128048	2.343514	135.573922
1500	3.045969	210.491820	1.895420	141.249795
1600	3.127717	204.674414	1.990102	128.416163

[1]Some of the SLOPE Volna runtimes for smaller tile sizes are not included in the tables due to increased execution time.

1700	3.089828	206.790264	2.074759	136.802580
1800	3.164951	201.283914	1.946865	133.880048
1900	2.885482	203.600191	1.813383	135.177901
2000	2.832507	204.581578	1.854197	136.131357
3000	3.561948	192.773536	2.092827	126.762589
4000	3.034038	183.488172	2.718769	118.424412
5000	3.184274	170.950104	3.129409	114.365665
6000	3.608053	188.345554	3.550377	127.438008
7000	4.075217	157.453148	4.036531	109.013354
8000	4.498560	166.742431	4.496186	116.394681
9000	4.883690	160.352694	5.061389	110.903383
10000	5.358443	158.724387	5.331901	104.989887
20000	9.597188	132.268719	9.601459	87.221041
30000	13.818190	174.867655	13.913055	104.374340
40000	16.884571	124.627123	16.938577	125.090795
50000	20.817461	146.832063	20.971385	147.308641
60000	20.831699	168.428924	20.932167	168.633580
70000	20.861719	192.090424	21.009545	192.453780
80000	20.858138	213.548625	20.971221	213.948832
90000	20.675212	236.770531	20.809683	237.408588
100000	20.408281	260.443272	20.515981	261.807362
200000	20.517654	487.798751	20.543645	490.760746

Volna on Telos

Table B.21: OP2 Volna runtimes on Telos (Figure 3.23 and Figure 3.24 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Dataset	Catalina	NU3	Catalina	NU3
Runtime	1.026248	60.705009	0.713585	49.422140

Table B.22: SLOPE Volna runtimes on Telos (Figure 3.23 and Figure 3.24 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Tile Size	Dataset			
	Catalina	NU3	Catalina	NU3
10	9.257547		4.702890	
20	6.121004	4741.469493	3.827582	4321.453749
30	3.010537	2344.052679	1.885374	1560.160739
40	1.679095	1722.669075	1.104433	1099.755066
50	2.283657	531.136033	1.549770	183.799853
60	1.585417	362.142410	1.933445	248.957546
70	1.552561	2433.063550	1.034077	2030.852856
80	1.461314	1631.509096	2.128603	1063.149433
90	3.531927	2008.556217	2.281476	1265.714917
100	1.525437	414.646671	1.048522	276.728015
200	1.413664	228.306144	1.075826	131.614666
300	1.258100	233.011007	0.982733	170.891401
400	1.287376	204.750068	0.938550	150.766042
500	1.221639	211.705800	0.878664	152.763001
700	1.285403	86.217697	0.971244	63.370069
800	1.141996	133.315049	0.837308	65.155204
900	1.227658	162.040126	0.944965	121.337284
1000	1.285089	88.122266	0.948360	66.340566
1100	1.389142	88.318372	0.930195	155.087454
1200	1.396157	90.416048	1.057659	167.440811
1300	1.469816	93.091612	1.097419	69.965877
1400	1.558598	90.118478	1.149589	101.534756
1500	1.195387	94.282091	0.805069	69.613247
1600	1.271160	85.679405	0.854109	65.564485
1700	1.388402	91.675268	0.863106	68.294300
1800	1.260975	89.258015	0.980857	118.138838
1900	1.125692	90.475349	1.023368	69.434958

2000	1.163983	91.377746	1.037813	71.403193
3000	1.398509	86.144495	1.387845	67.494515
4000	1.728368	79.748089	1.776605	66.352718
5000	2.143319	78.857438	2.173021	67.300309
6000	2.520546	89.751848	2.578760	73.773945
7000	2.893919	75.834212	2.900672	64.430613
8000	3.155727	81.704993	3.243714	68.842992
9000	3.496842	78.629092	3.551363	63.721414
10000	3.811064	73.810100	3.894678	57.904235
20000	6.883825	60.072484	6.995613	59.691949
30000	10.100779	76.354523	10.223722	75.316994
40000	11.969175	92.760372	12.108369	92.283532
50000	15.051260	109.738016	15.136187	108.854130
60000	15.067212	125.458974	15.145740	125.329704
70000	15.026410	144.316248	15.184368	143.279255
80000	14.962971	159.324152	15.125359	158.425760
90000	14.915412	177.387419	15.056660	176.393065
100000	14.714286	194.472391	14.839329	193.872290
200000	14.671915	365.238359	14.826270	363.700666

Volna on ARCHER2

Table B.23: OP2 Volna runtimes on ARCHER2 (Figure 3.25 and Figure 3.26 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Dataset	Catalina	NU3	Catalina	NU3
Runtime	1.545005	66.631699	2.903588	64.237475

Table B.24: SLOPE Volna runtimes on ARCHER2 (Figure 3.25 and Figure 3.26 runtimes in seconds)

Configuration	Single Socket		Dual Socket	
Tile Size	Dataset			
	Catalina	NU3	Catalina	NU3
10	4.017659		7.548402	
20	3.499108		8.512839	
30	2.143481	3014.337009	4.858965	4276.985132
40	1.264754	2105.899395	3.255981	1627.902361
50	1.475060	556.120471	3.469751	942.955235
60	1.622196	536.507315	6.443164	465.193135
70	1.692036	988.279696	3.363306	1392.911143
80	3.435506	308.793921	6.912753	280.038068
90	1.754331	1578.707033	3.350593	1447.335421
100	1.833271	608.407148	3.318451	436.519588
200	1.798045	940.333450	2.776503	1613.022212
300	1.564665	3049.070303	2.277948	708.455172
400	1.384516	125.801805	1.945414	146.520006
500	1.326572	225.070493	1.702930	147.657249
700	1.386064	119.760735	1.749781	140.910106
800	1.116954	124.062751	1.507749	143.348112
900	1.211124	122.721198	1.542785	140.218940
1000	1.282077	129.612550	1.576839	233.349609
1100	1.344120	123.945603	1.635220	143.395861
1200	1.421053	302.761403	1.711425	139.876780
1300	1.494327	128.509121	1.784012	140.555878
1400	1.536533	214.712946	1.841274	238.136794
1500	1.165310	123.129371	1.425677	141.131066
1600	1.209043	120.793912	1.475108	134.933626
1700	1.259070	119.614611	1.523736	139.891403
1800	1.299620	225.782905	1.566481	155.995546

1900	1.355705	124.188684	1.626564	133.294108
2000	1.432454	119.330523	1.685851	133.884447
3000	1.919302	111.949290	2.181636	126.710590
4000	2.465322	108.219987	2.742361	111.477223
5000	3.090459	100.709302	3.430276	109.002377
6000	3.553315	105.214075	3.855603	106.942209
7000	4.093264	98.376264	4.421244	99.240162
8000	4.706117	98.299956	5.242819	103.596324
9000	5.214249	93.909034	5.687052	96.773318
10000	5.635220	96.032544	6.082998	97.718178
20000	10.242768	102.683110	10.733723	105.333116
30000	14.960879	117.797522	15.741874	122.214644
40000	17.031453	137.257175	17.857068	142.483744
50000	21.914704	158.501889	23.001771	164.836012
60000	21.936089	180.035647	23.046504	187.332683
70000	21.926703	204.626917	22.953647	214.383883
80000	21.915375	229.310140	22.877380	237.570998
90000	21.918167	256.478577	22.973167	266.664109
100000	21.961801	280.657113	23.026474	291.013086
200000	21.940809	539.734834	23.039283	554.547419

B.1.4 OP2 Hydra Runtimes

OP2 Hydra on Scyrus

Table B.25: OP2 Hydra single socket runtimes on Scyrus (Figure 3.29 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	25.399872	81.787834	20.236053	272.804718	668.075500	207.988312

Table B.26: SLOPE Hydra single socket runtimes on Scyrus (Figure 3.29 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	73.088005	142.235870	33.935959	860.716858	1309.488953	383.828125
20	58.804207	126.364021	28.952049	680.940369	1126.035828	323.568481
30	50.227982	117.544106	25.743820	558.324585	1027.863892	280.776428
40	48.164032	111.659981	23.595947	554.480164	980.108490	268.660461
50	45.447899	107.063766	22.343994	527.829468	936.964233	250.032379
60	43.212135	103.608284	21.547943	477.220703	878.759583	237.152039
70	39.799957	101.352066	21.059982	460.050537	883.320404	234.283966
80	37.428185	100.880104	20.447937	449.024231	870.642670	231.247986
90	36.711899	100.208046	20.180084	421.580505	864.360291	221.971924
100	35.384010	99.151863	19.884209	413.543854	845.616852	219.599487
200	29.564240	91.568329	18.415939	330.704926	764.280090	203.012207
300	27.539902	87.655746	17.872009	313.372131	731.504089	197.867737
400	26.672173	86.047882	17.611977	294.802551	712.764191	195.152405
500	26.455994	84.808075	17.476067	286.452179	700.836792	192.519897
700	25.203941	83.396042	17.299965	271.432831	673.715729	189.919617
800	24.412048	82.379822	17.116035	270.204529	675.727722	188.372253
900	24.488106	81.980003	17.043991	258.767090	675.604401	189.076019
1000	24.435867	82.720001	17.275978	256.652679	667.860352	187.775482
1100	24.423813	83.132126	17.359924	256.620667	668.323730	186.888031
1200	24.027863	83.016045	17.240051	255.142975	667.268555	186.480011
1300	24.004143	82.768051	17.172020	253.847260	668.892334	186.575775
1400	24.419731	82.964035	17.384079	252.055237	660.199738	185.216003
1500	24.315941	83.164139	17.459984	251.386017	659.724152	185.160126
1600	24.091919	83.412064	17.375977	250.568176	661.987793	184.364197
1700	22.624138	81.156120	16.915970	250.316711	661.787598	186.864105
1800	23.848015	82.624092	17.203995	247.872192	658.784363	185.103882

1900	23.924042	82.400215	17.248009	248.856995	659.203918	184.219940
2000	23.160042	82.300186	17.339928	248.244080	657.299591	185.036499
3000	24.680130	85.335915	18.140068	242.888702	654.011627	184.223724
4000	23.356239	81.952179	17.536026	247.011841	651.764771	184.080292
5000	25.472099	87.339767	18.704002	249.640594	648.448975	184.372528
6000	26.147942	89.464195	19.268013	249.633301	652.964203	185.396149
7000	24.536240	83.419594	18.071823	242.731842	650.151947	186.599548
8000	27.396042	93.347946	20.504036	240.812714	644.208435	184.099945
9000	26.103790	91.128006	19.679962	243.558899	651.291351	185.799805
10000	26.663773	93.568253	20.107918	243.340668	654.227142	188.147797
20000	31.915955	109.691925	23.899948	246.859192	662.757080	190.515930
30000	32.948166	113.068092	24.864014	249.807800	670.891968	191.275757
40000	33.604034	116.819878	25.660141	261.273010	700.223663	200.571503
50000	36.248199	127.252014	27.319931	257.677307	688.579590	201.815979
60000	45.012161	156.544106	34.199928	263.195374	715.748413	204.896362
70000	49.316277	176.616028	37.796005	274.975647	750.476593	215.300262
80000	65.251556	236.568108	50.695961	275.212738	736.944427	212.468506
90000	62.612083	225.399696	47.956085	268.131683	716.095001	207.655670
100000	69.747444	250.283882	53.375931	276.209503	736.456482	212.944244
200000	113.899643	402.143738	88.487938	340.543579	912.243866	269.252167

Table B.27: OP2 Hydra dual socket runtimes on Scyrus (Figure 3.30 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	47.943764	98.404045	27.116249	344.860809	748.428162	252.400452

Table B.28: SLOPE Hydra dual socket runtimes on Scyrus (Figure 3.30 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	120.979919	190.983749	47.340134	1198.798584	1647.084900	549.487732
20	91.328140	160.891983	38.415909	921.516174	1323.620178	440.175598
30	79.287842	149.864258	33.615952	739.504150	1229.329895	381.456360
40	76.555695	138.436127	31.740082	736.390869	1141.378845	355.735535
50	72.500061	133.427872	30.292053	676.569092	1063.501099	322.076660
60	69.867645	127.467789	28.504044	606.394104	1004.705200	302.723938
70	64.515976	123.972153	27.279922	600.415527	1006.523560	297.512085
80	61.391998	121.531937	26.420059	573.848877	972.395996	288.147644

90	59.792114	120.580261	26.191879	533.980774	960.764160	277.064270
100	57.300201	119.664093	25.963882	510.680115	944.487305	266.904175
200	46.715942	102.215805	21.980087	401.072449	822.917419	233.007446
300	42.036331	95.155716	20.207932	361.051697	777.934387	222.764771
400	39.788208	92.608032	19.620026	336.392273	743.380127	212.915344
500	38.659821	91.160004	19.484085	326.301270	728.907654	211.904236
700	37.363792	88.951828	19.255875	299.028748	702.745422	205.316101
800	35.108063	86.427811	18.595863	293.327942	714.200439	203.547913
900	35.839478	87.656181	18.995895	284.683472	698.111755	201.528076
1000	35.383530	86.916054	18.912056	284.407471	694.083740	202.735107
1100	35.023163	89.371841	19.344162	284.100830	692.959412	200.364136
1200	34.511803	87.659996	19.064018	279.955017	684.268494	198.779846
1300	34.391907	88.867966	19.088005	274.589233	686.688171	200.248047
1400	34.811890	89.816010	19.367996	282.496216	686.096680	197.555664
1500	34.523895	90.280212	19.748062	277.695862	684.760254	197.356262
1600	33.859940	89.552307	19.284012	269.848572	683.004578	196.212341
1700	33.016251	87.519928	18.996086	269.076904	675.802734	195.715179
1800	33.807953	89.996078	19.284073	264.264709	672.900879	194.579834
1900	34.512596	90.052193	19.515976	269.167542	676.972717	195.460022
2000	33.984116	90.876022	19.560089	263.940735	671.372314	194.632324
3000	33.687836	95.020210	20.824043	258.804993	664.563843	193.851410
4000	32.027908	90.811722	20.099976	257.331879	664.020782	195.203644
5000	35.419678	102.088127	22.436073	262.800507	669.582703	195.515961
6000	35.139969	100.303947	22.327942	254.508087	669.681000	195.572144
7000	32.639923	97.103973	21.792038	259.884399	675.647522	199.380371
8000	37.615913	110.179947	25.183914	260.817078	667.663818	198.883789
9000	36.359840	109.619873	24.515991	264.383881	678.604065	200.524017
10000	35.756088	111.667862	25.099976	259.755310	674.459839	199.292236
20000	39.448151	121.880196	27.415916	273.087952	705.867615	212.755798
30000	53.519867	176.924347	39.804085	286.139587	723.040344	218.071869
40000	59.428040	202.287788	45.523857	299.367218	742.535736	228.115906
50000	73.271782	256.503693	56.227821	300.220917	743.431396	232.552185
60000	89.952347	314.808098	69.835892	319.659912	815.752594	245.452362
70000	100.111893	354.024376	76.484047	312.995514	781.540405	242.655884
80000	130.864685	472.312012	100.499641	336.542725	843.540039	262.304504
90000	126.000183	454.072006	97.516182	330.080872	833.027527	256.060333
100000	139.644196	503.676346	108.331802	350.371887	868.038788	268.476074
200000	229.403290	808.927658	179.903793	430.392456	1112.140381	334.278931

OP2 Hydra on Telos

Table B.29: OP2 Hydra single socket runtimes on Telos (Figure 3.31 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	28.870171	73.221611	21.049042	297.475311	575.923676	190.775986

Table B.30: SLOPE Hydra single socket runtimes on Telos (Figure 3.31 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	70.295441	125.656189	31.882080	844.550964	1150.273926	350.913635
20	55.489227	111.842224	26.239029	659.184875	976.719727	285.046875
30	47.755142	104.409302	23.058571	538.266785	897.493530	247.379395
40	45.191467	97.767708	21.504135	539.400391	846.502930	237.596191
50	42.092606	92.459229	19.929550	499.319092	794.661926	220.684570
60	40.434410	89.476364	19.272743	457.152405	755.973633	207.822266
70	37.333183	87.399811	18.502144	442.100708	743.301147	201.430939
80	35.457062	87.315514	18.293884	436.008972	738.746460	200.761414
90	34.478271	85.606033	17.945068	406.730225	724.001678	195.261658
100	33.308517	84.198120	17.593254	388.026733	701.280823	186.964508
200	28.266495	77.182480	15.862862	322.421936	649.172516	172.820831
300	25.596222	73.394806	15.256905	290.840088	621.726379	167.070709
400	24.841225	72.568184	15.052849	274.639893	604.055695	163.112885
500	24.323792	72.626617	14.913429	266.152863	599.021362	164.068634
700	23.574478	72.730034	14.909981	254.053955	578.403931	157.474091
800	22.548874	70.299599	14.635078	245.168396	576.933014	156.885742
900	22.608047	71.711815	14.897003	244.372681	575.415833	156.830933
1000	22.787750	72.304359	14.859550	241.222504	576.012787	158.433777
1100	23.261299	73.283554	15.201958	238.339203	569.434937	155.135010
1200	22.289543	73.071739	15.033073	232.908295	560.945221	153.667938
1300	22.587929	73.607628	15.239395	228.998688	567.492645	158.285736
1400	22.452576	74.085678	15.542870	230.917297	555.297821	154.437622
1500	23.549271	76.586189	16.470573	227.706940	550.949127	152.399017
1600	22.810631	74.264908	15.309196	228.493774	556.750641	157.104462
1700	22.015877	72.552338	15.070686	225.027710	556.542389	154.915009
1800	23.153053	75.754967	15.781975	221.976166	560.017639	153.999878
1900	22.876015	74.351669	15.379898	229.255341	561.866486	155.501251
2000	22.536667	74.977570	15.272362	226.023102	560.419891	154.456635

3000	24.282776	79.449059	16.830658	221.952148	561.291931	156.222046
4000	22.721939	75.614464	15.964409	221.996216	558.573792	155.432800
5000	25.076340	85.147087	17.920067	221.920013	559.280396	155.581757
6000	25.524559	83.908485	17.817764	222.501953	560.541992	156.865356
7000	23.627876	81.557068	17.523270	224.340942	568.492157	158.566284
8000	28.050751	91.521759	20.048836	225.652130	567.011047	161.810883
9000	26.834717	91.457970	19.656654	229.009430	566.934174	159.111115
10000	27.654816	94.044426	20.167259	224.377991	563.888245	160.136627
20000	30.088478	103.251259	22.423317	234.056183	595.236481	170.081299
30000	43.614845	150.929520	33.376312	241.281403	612.422028	177.192017
40000	48.294334	170.232094	35.814308	251.699951	624.595398	184.142731
50000	59.758064	212.756790	44.779503	251.084442	619.019440	181.412903
60000	74.582634	262.585289	55.676666	267.232605	689.718109	201.842438
70000	82.014999	297.175392	61.441315	261.950989	655.612244	194.054230
80000	108.304245	399.593613	81.675560	279.857208	697.941742	203.831299
90000	104.996056	382.432892	79.011551	278.138489	704.166473	209.216339
100000	117.379021	427.811020	90.399330	287.464630	715.865112	210.121948
200000	192.112419	680.017830	146.926300	358.897552	929.166626	275.062805

Table B.31: OP2 Hydra dual socket runtimes on Telos (Figure 3.32 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	48.963837	84.995422	27.717880	367.243225	594.236633	202.846405

Table B.32: SLOPE Hydra dual socket runtimes on Telos (Figure 3.32 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	92.289490	161.047394	46.518143	1168.999512	1447.282227	445.107178
20	78.699997	136.960999	29.713409	840.721008	1151.231873	340.991516
30	79.026031	128.211044	28.647568	742.088013	1052.126099	294.095886
40	72.440796	118.184723	27.405380	780.582031	1014.806824	276.015198
50	67.391403	111.542007	26.348999	656.139954	934.309143	302.708740
60	70.965775	109.072464	23.327332	596.317322	854.076477	242.789551
70	64.510361	104.972031	24.079147	566.965637	795.201660	227.800903
80	66.139847	105.043320	27.014130	542.181885	833.019165	228.550171
90	60.397690	102.051102	22.853165	490.756165	790.952332	216.964783
100	58.235275	99.482300	24.061478	487.946838	784.654236	277.753113

200	50.993958	87.765839	19.032074	499.941162	718.351257	214.287476
300	42.676498	81.829880	17.979904	475.048279	661.395935	182.051758
400	39.181366	78.794769	19.305206	395.684143	631.628967	173.982910
500	38.886139	78.322372	19.142151	424.567139	620.856323	177.212585
700	39.330383	78.800720	18.575302	395.909485	588.008667	207.231689
800	35.792725	77.654556	18.034851	371.603516	588.455872	183.036926
900	37.163086	78.058197	18.178177	289.516663	586.703430	165.977966
1000	37.267654	79.255127	18.865189	346.524536	590.620300	170.834351
1100	37.228455	80.996857	18.131454	367.591431	589.110901	170.953613
1200	36.757553	80.957458	17.870361	365.499817	600.345154	167.668762
1300	34.746902	81.487320	18.794662	364.640381	598.905212	168.704651
1400	34.446793	82.537735	18.450226	390.919067	598.060852	169.524109
1500	35.866989	82.509659	18.985992	375.591675	586.466919	168.419250
1600	36.578339	85.759979	19.807602	351.313660	591.378540	168.995605
1700	34.896835	81.621765	18.174194	351.703674	582.528259	189.302795
1800	36.069214	86.715927	19.081436	332.375183	577.910950	183.593750
1900	37.250885	86.780304	20.029083	345.021484	587.742249	169.479126
2000	37.187698	86.071198	18.799179	338.162903	581.279785	172.609131
3000	37.744919	97.024033	21.855774	277.062256	575.036621	171.344788
4000	36.178024	92.465561	20.534805	306.189026	585.501221	176.184265
5000	40.679276	119.100388	25.773804	324.949402	589.270020	173.816711
6000	38.388351	99.639069	22.176895	304.445679	581.915955	181.152222
7000	38.132309	96.876007	22.152161	292.534851	600.097290	175.417725
8000	38.214355	108.072937	25.403427	339.946533	599.547180	176.765198
9000	36.935623	108.909821	24.531097	321.302551	603.796936	189.511780
10000	39.223114	119.866028	26.958588	276.689087	610.690796	181.100952
20000	60.188446	204.746689	45.162552	312.336792	677.568176	203.249756
30000	86.671738	297.759842	65.225632	339.323425	684.049561	202.993713
40000	96.772369	341.098907	73.426025	381.059265	733.146240	217.959045
50000	119.654510	426.907883	89.927399	347.972168	745.047913	224.320374
60000	149.416626	530.725815	114.130386	373.222290	816.821350	252.014221
70000	164.898178	590.698547	124.468140	377.350586	756.419128	234.156555
80000	218.093552	803.150879	164.822144	365.705505	777.217957	241.263611
90000	211.357758	761.367401	158.410721	398.687927	849.470581	249.554443
100000	235.349045	855.437714	178.042007	367.881042	848.567017	258.568542
200000	390.451416	1386.409348	306.288971	518.796448	1265.501770	386.538330

OP2 Hydra on ARCHER2

Table B.33: OP2 Hydra single socket runtimes on ARCHER2 (Figure 3.33 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	123.986202	189.672642	57.442844	2064.460693	2299.738647	949.684753

Table B.34: SLOPE Hydra single socket runtimes on ARCHER2 (Figure 3.33 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	197.336494	313.567123	86.736122	3149.510193	3243.888733	1246.821045
20	133.657852	248.175560	61.419418	2643.987122	2825.856934	1026.038574
30	157.384033	259.829575	67.046471	2383.356262	2655.644165	941.837280
40	142.201263	240.553398	66.011147	2369.808960	2610.481018	965.908264
50	136.206932	238.269043	65.242149	2304.694214	2518.972839	951.851379
60	150.415154	244.718315	69.342049	2203.286194	2454.261902	938.880310
70	152.604813	250.327301	69.620178	2204.829895	2493.563110	940.817871
80	154.649643	246.945503	68.310127	2106.375916	2475.817322	916.775024
90	131.603722	229.924469	62.163040	2109.142517	2440.929871	882.747986
100	134.699677	235.236954	60.897675	2149.651367	2436.195129	876.452026
200	144.591415	215.027336	57.315491	1993.498474	2190.335999	794.551819
300	135.105057	192.744453	51.687546	1915.475342	2109.682190	779.093323
400	137.690933	194.532875	52.207466	1811.203552	2012.395447	750.681824
500	132.364983	191.455994	52.434166	1740.465881	1962.513977	725.481689
700	132.841805	189.472282	51.170082	1631.052124	1831.332092	685.548218
800	126.272568	183.529198	49.407120	1592.353271	1787.842224	661.343811
900	129.214211	187.626266	50.907364	1582.593811	1766.163208	653.730530
1000	115.386520	171.520142	46.311493	1551.014709	1735.447327	641.801697
1100	117.391235	175.342705	47.069397	1533.376648	1749.209656	637.213196
1200	122.603592	176.689346	49.104851	1513.080139	1722.849487	628.673828
1300	121.266258	174.820396	47.579643	1492.497253	1697.943909	620.916809
1400	123.717323	177.436668	47.589859	1457.518433	1635.105652	608.976624
1500	121.921967	175.584526	47.592834	1469.643860	1654.082397	608.366394
1600	123.045799	178.108749	48.101669	1426.782776	1618.257751	601.506226
1700	117.884094	174.942665	46.817345	1422.641663	1621.450256	594.148438
1800	123.690147	175.011345	47.844124	1418.707947	1593.676208	591.129211

1900	122.186050	178.812866	47.592331	1389.405884	1568.904114	589.108215
2000	125.319916	183.419327	48.868225	1383.502991	1564.383240	582.061584
3000	116.379875	180.410034	48.102509	1351.400024	1546.007141	560.196472
4000	121.417763	180.597321	48.599983	1319.438782	1520.905518	545.637512
5000	118.594994	180.383217	51.141136	1310.881165	1511.309631	548.084351
6000	123.963127	176.285309	51.401131	1284.274780	1465.033142	537.757874
7000	122.607780	182.476265	53.207870	1282.660583	1457.668457	525.416016
8000	121.009560	186.268532	53.463692	1299.457642	1472.978271	527.068054
9000	114.731766	177.186325	51.902946	1292.680603	1464.800476	531.876343
10000	111.999565	182.978279	51.580605	1287.059387	1471.544067	528.750244
20000	111.279564	226.487701	59.565479	1316.202026	1615.906189	600.978882
30000	123.996704	262.635719	77.028687	1316.991516	1610.733398	628.341553
40000	136.238998	234.684586	74.271030	1383.690125	1659.044739	665.155090
50000	157.319466	240.523148	74.899483	1274.887207	1526.135010	641.848450
60000	190.696472	258.760117	79.029137	1269.557800	1454.793945	615.124939
70000	200.573273	246.445557	76.450768	1228.247742	1439.569214	603.982605
80000	255.913017	282.394257	88.111069	1246.916260	1444.042725	611.095947
90000	244.987717	250.003578	77.350609	1124.419861	1303.839844	545.593262
100000	265.879982	251.954239	78.490318	1124.895081	1307.900757	537.360474
200000	296.781174	222.206230	68.702583	831.640076	1033.874207	351.883850

Table B.35: OP2 Hydra dual socket runtimes on ARCHER2 (Figure 3.34 runtimes in seconds)

Dataset	1M			8M		
Loop-chain	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
Runtime	330.913330	483.549698	158.775543	4206.099976	4799.358459	2016.226807

Table B.36: SLOPE Hydra dual socket runtimes on ARCHER2 (Figure 3.34 runtimes in seconds)

Dataset	1M			8M		
Tile Size	Loop-chain					
	`iflux`	`vflux`	`jinit`	`iflux`	`vflux`	`jinit`
10	535.870621	598.662598	191.044968	6781.271484	6828.332520	2885.359253
20	332.429199	410.443405	127.878937	5647.601929	6050.831909	2509.407715
30	444.459167	551.112839	167.194702	5202.412231	5716.537598	2322.251099
40	395.769958	507.816589	157.550400	5155.490356	5587.999146	2374.281250
50	413.486954	526.756165	164.728683	5013.248047	5447.409058	2335.802124
60	449.147827	589.133316	178.353195	4854.680054	5344.083740	2257.328125
70	433.487137	588.425415	175.853439	4740.109863	5417.717285	2289.335938

80	476.774292	624.518768	184.794281	4713.338501	5438.072510	2206.750366
90	348.058075	467.069489	147.691727	4704.209839	5443.615723	2186.931519
100	364.658600	482.497681	142.280838	4789.292480	5436.142578	2168.114624
200	441.246033	558.371628	156.611725	4378.978394	4656.067505	1769.830566
300	357.421371	478.693420	131.010468	4330.541992	4521.027222	1723.598877
400	395.046799	552.413162	141.499374	4053.523315	4367.587280	1646.079956
500	357.911224	496.359756	131.077576	3846.475830	4155.664307	1553.095215
700	382.570190	536.558105	138.758209	3455.767578	3879.185425	1434.619507
800	351.097580	500.351059	130.563858	3407.194336	3804.308716	1381.213928
900	366.460800	517.088806	132.100525	3292.340332	3707.071411	1353.455933
1000	322.939285	471.264694	123.372162	3239.146362	3659.856445	1345.947571
1100	322.688248	472.627060	121.878082	3212.364014	3596.536621	1313.375549
1200	321.788361	481.625015	124.895584	3142.365601	3547.388672	1282.970093
1300	331.238724	492.931671	126.561996	3140.703125	3533.443359	1284.847046
1400	345.929184	515.184372	130.996582	3084.749390	3459.455261	1253.193604
1500	344.029343	506.526016	128.084991	2998.802490	3431.746460	1240.782959
1600	337.549591	504.803268	129.032806	3003.247925	3379.809753	1221.167908
1700	337.071091	498.111816	127.996979	2983.241211	3347.098022	1212.414795
1800	342.454407	510.844528	130.037872	2967.259094	3351.786621	1207.256409
1900	347.932846	511.274628	131.159424	2974.486084	3322.058044	1201.058411
2000	329.167236	491.963455	126.457672	2869.693481	3271.166260	1174.378967
3000	316.121109	473.051437	121.102058	2821.223511	3135.558289	1120.922668
4000	343.445374	471.978424	121.289932	2764.006592	3061.336182	1090.361877
5000	340.948883	470.316147	116.258156	2739.197327	3038.557373	1073.992004
6000	352.286636	468.835266	119.744492	2746.900696	3023.345581	1073.439392
7000	347.957809	454.919434	119.760757	2721.781433	3013.649658	1064.176453
8000	348.468933	477.174911	122.538567	2675.129150	2985.896606	1059.108704
9000	312.850937	431.642868	110.446159	2669.615845	3018.427673	1066.748840
10000	297.727432	409.295944	111.302650	2653.486267	2968.921570	1059.705200
20000	302.475143	488.110229	131.763260	2693.872681	2967.045593	1087.492249
30000	333.377228	531.059677	162.409554	2458.898987	2681.209961	1030.502869
40000	348.609314	466.614975	150.853058	2172.135742	2317.156555	907.437256
50000	381.318298	473.871384	151.680717	1895.254456	1983.659668	807.814148
60000	440.049194	515.130844	160.025261	1736.673035	1788.948120	728.991455
70000	468.232666	478.278534	154.379990	1687.629089	1753.675232	722.814026
80000	584.577087	549.682144	176.386032	1623.768921	1674.896912	685.266968
90000	560.474121	482.132126	154.592903	1394.610901	1367.849731	566.834045
100000	599.614746	484.025192	155.163780	1354.224182	1327.393188	552.188110
200000	619.518814	417.862976	135.212654	1036.879639	937.083740	354.233521

B.2 Chapter 4 Runtimes

B.2.1 MG-CFD on ARCHER2

Table B.37: MG-CFD on ARCHER2 with CA (Figure 4.8 and Figure 4.9 Runtimes in seconds)

#Nodes	#Loops	8M Mesh		24M Mesh	
		OP2	CA	OP2	CA
1	2	2.930993	3.070072	10.720874	11.440359
	4	3.729141	3.76727	13.615567	15.198768
	8	5.314055	5.097041	19.446233	23.646258
	16	8.487252	8.286247	31.047089	39.046114
	32	14.84507	14.608413	54.202636	68.485632
4	2	0.533928	0.62771	2.523079	2.661125
	4	0.617511	0.690082	3.209099	3.205468
	8	0.784929	0.805743	4.57852	4.290228
	16	1.103811	1.016651	7.312483	6.951929
	32	1.725986	1.456325	12.803113	12.462673
16	2	0.140511	0.144029	0.464795	0.560076
	4	0.166266	0.159771	0.542713	0.606314
	8	0.213651	0.190464	0.680995	0.709576
	16	0.311171	0.251564	0.957947	0.893476
	32	0.511894	0.378806	1.529867	1.277988
64	2	0.055831	0.061244	0.139191	0.15138
	4	0.061306	0.069327	0.154382	0.162465
	8	0.086132	0.072372	0.199901	0.18425
	16	0.116883	0.093278	0.293077	0.23541
	32	0.187194	0.122697	0.505667	0.345632

B.2.2 OP2 Hydra on ARCHER2

Table B.38: Hydra loop-chains (LCs) on ARCHER2 with CA (Figure 4.15 and Figure 4.16 Runtimes in seconds)

Loop-chain	#Loops	# Nodes	8M Mesh		24M Mesh	
			OP2	CA	OP2	CA
weight	5	4	0.003632	0.003502	0.016106	0.021683
		16	0.001663	0.00177	0.005264	0.009216
		64	0.001663	0.00148	0.002075	0.003387
		128	0.002136	0.001831	0.002014	0.002258
period	6	4	0.005638	0.005463	0.036079	0.024612
		16	0.002777	0.001801	0.011307	0.008408
		64	0.002594	0.001495	0.003845	0.002777
		128	0.002228	0.001465	0.002808	0.001831
iflux	2	4	0.538475	0.559158	3.005577	3.043602
		16	0.146362	0.15332	0.679062	0.718018
		64	0.050812	0.06015	0.169861	0.17749
		128	0.037903	0.045837	0.088684	0.093933
vflux	2	4	2.002327	2.184647	7.38031	8.319183
		16	0.601913	0.671631	2.662613	3.149704
		64	0.16037	0.161072	0.722778	0.803314
		128	0.102722	0.098938	0.334167	0.373962
jacob	3	4	0.245201	0.182465	1.167717	0.831589
		16	0.08873	0.084671	0.635101	0.454987
		64	0.038177	0.022675	0.149994	0.122711
		128	0.026886	0.017883	0.115051	0.068787
gradl	2	4	1.102699	2.66539	5.095749	11.317764
		16	0.259811	1.009399	2.040604	5.251663
		64	0.090149	0.36319	0.406281	1.414154
		128	0.062042	0.187866	0.207031	0.751099

B.3 Chapter 5 Runtimes

B.3.1 MG-CFD on ARCHER2

Table B.39: SLOPE MG-CFD on ARCHER2 with CA+SLOPE (Figure 5.6 and Figure 5.7 Runtimes in seconds)

#Nodes	#Loops	8M Mesh			24M Mesh		
		OP2	CA	CA+SLOPE	OP2	CA	CA+SLOPE
1	2	2.930993	3.070072	3.1699	10.720874	11.440359	13.28962
	4	3.729141	3.76727	4.066019	13.615567	15.198768	16.943315
	8	5.314055	5.097041	6.190342	19.446233	23.646258	25.889221
	16	8.487252	8.286247	8.42447	31.047089	39.046114	36.816682
	32	14.84507	14.608413	14.567548	54.202636	68.485632	64.280083
4	2	0.533928	0.62771	0.666905	2.523079	2.661125	2.842304
	4	0.617511	0.690082	0.797093	3.209099	3.205468	3.664547
	8	0.784929	0.805743	1.087776	4.57852	4.290228	5.51643
	16	1.103811	1.016651	1.671237	7.312483	6.951929	8.066924
	32	1.725986	1.456325	2.791609	12.803113	12.462673	14.063178
16	2	0.140511	0.144029	0.178726	0.464795	0.560076	0.602067
	4	0.166266	0.159771	0.214978	0.542713	0.606314	0.722715
	8	0.213651	0.190464	0.291028	0.680995	0.709576	0.979169
	16	0.311171	0.251564	0.444391	0.957947	0.893476	1.445786
	32	0.511894	0.378806	0.761229	1.529867	1.277988	2.407379
64	2	0.055831	0.061244	0.07155	0.139191	0.15138	0.167686
	4	0.061306	0.069327	0.085915	0.154382	0.162465	0.20439
	8	0.086132	0.072372	0.109746	0.199901	0.18425	0.274202
	16	0.116883	0.093278	0.15648	0.293077	0.23541	0.411426
	32	0.187194	0.122697	0.248331	0.505667	0.345632	0.690589

B.3.2 OP2 Hydra on ARCHER2

Table B.40: Hydra loop-chains (LCs) on ARCHER2 with CA+SLOPE (Figure 5.9 and Figure 5.10 Runtimes in seconds)

LC	#Loops	# Nodes	8M Mesh			24M Mesh		
			OP2	CA	CA+SLOPE	OP2	CA	CA+SLOPE
iflux	2	4	0.538475	0.559158	0.724365	3.005577	3.043602	3.532005
		16	0.146362	0.15332	0.215073	0.679062	0.718018	0.86351
		64	0.050812	0.06015	0.076233	0.169861	0.17749	0.223907
		128	0.037903	0.045837	0.058044	0.088684	0.093933	0.120422
vflux	2	4	2.002327	2.184647	2.338379	7.38031	8.319183	8.884483
		16	0.601913	0.671631	0.769852	2.662613	3.149704	3.272095
		64	0.16037	0.161072	0.233521	0.722778	0.803314	0.89679
		128	0.102722	0.098938	0.118958	0.334167	0.373962	0.455566
jinit	3	4	0.482712	0.505889	0.572700	1.789101	1.849808	2.022552
		16	0.126160	0.131485	0.160965	0.554749	0.556625	0.636688
		64	0.036804	0.040100	0.050354	0.128235	0.128296	0.148590
		128	0.025818	0.025269	0.031433	0.075378	0.076294	0.088013

B.4 Chapter 6 Runtimes

B.4.1 MG-CFD on Cirrus

Table B.41: MG-CFD on Cirrus with CA (Figure 6.1 and Figure 6.2 Runtimes in seconds)

#Nodes	#Loops	8M Mesh		24M Mesh	
		OP2	CA	OP2	CA
1	2	0.911974	0.906979	2.934365	2.894385
	4	1.210799	1.226389	3.944769	3.955132
	8	1.799157	1.985154	5.942899	6.439425
	16	2.987328	3.830864	9.951074	12.661138
	32	5.35759	9.302618	17.976473	30.081926
2	2	0.534429	0.521662	1.557592	1.527173
	4	0.709469	0.669624	2.089259	2.131246
	8	1.045167	1.00278	3.133208	3.631902
	16	1.716853	1.762862	5.216451	7.767716
	32	3.070534	3.483697	9.377847	20.497252
4	2	0.346886	0.335531	0.928538	0.897967
	4	0.452898	0.413558	1.217522	1.17884
	8	0.656619	0.589563	1.821676	1.866581
	16	1.076307	0.971551	2.996623	3.712419
	32	1.888532	1.693314	5.412619	8.9781
8	2	0.237628	0.24885	0.552221	0.547137
	4	0.307161	0.274408	0.72114	0.662816
	8	0.436196	0.35251	1.05518	0.959713
	16	0.693798	0.510327	1.715036	1.583089
	32	1.221451	0.828059	3.110476	2.905805
16	2	0.1712	0.192156	0.40924	0.461693
	4	0.218904	0.199921	0.521129	0.477164
	8	0.311043	0.238145	0.752973	0.604293
	16	0.490245	0.327924	1.254412	0.891559
	32	0.866135	0.497408	2.184823	1.45791

B.4.2 OP2 Hydra on Cirrus

Table B.42: Hydra loop-chains (LCs) on Cirrus with CA (Figure 6.7 and Figure 6.8 Runtimes in seconds)

Loop-chain	#Loops	# Nodes	8M Mesh		24M Mesh	
			OP2	CA	OP2	CA
iflux	2	2	0.520004	0.470032	2.460052	1.920059
		4	0.239975	0.210037	1.300079	0.980042
		8	0.22995	0.169952	0.889938	0.629776
		16	0.189957	0.170013	0.52002	0.360016
vflux	2	2	2.45993	1.429947	8.770309	5.269836
		4	1.519897	1.240097	6.159912	4.660034
		8	1.219994	1.030029	4.480026	4.090027
		16	0.929993	0.799942	4.069977	3.279984
jacob	3	2	0.379929	0.159988	0.760056	0.329956
		4	0.239914	0.079987	0.590042	0.280045
		8	0.150009	0.119987	0.410004	0.229996
		16	0.159973	0.070007	0.410019	0.13002
gradl	2	2	1.670013	1.370056	5.499924	4.95993
		4	0.960251	0.900085	3.150024	2.579941
		8	0.690048	0.870026	1.709946	1.509949
		16	0.360016	0.580002	1.609985	1.349899