View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

Open Access Dissertations | Theses and Dissertations

Fall 2014

Scaling finite difference methods in large eddy simulation of jet engine noise to the petascale: numerical methods and their efficient and automated implementation

Yingchong Situ
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Situ, Yingchong, "Scaling finite difference methods in large eddy simulation of jet engine noise to the petascale: numerical methods and their efficient and automated implementation" (2014). *Open Access Dissertations*. 365.
https://docs.lib.purdue.edu/open_access_dissertations/365

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Yingchong Situ

Entitled
Scaling Finite Difference Methods in Large Eddy Simulation of Jet Engine Noise to the Petascale: Numerical Methods and Their Efficient and Automated Implementation

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Zhiyuan Li

Ananth Y. Grama

Robert D. Skeel

Ahmed H. Sameh

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Zhiyuan Li

Approved by: Sunil Prabhakar/William J. Gorman 10/16/2014

Head of the Department Graduate Program Date

SCALING FINITE DIFFERENCE METHODS IN LARGE EDDY SIMULATION OF JET ENGINE NOISE TO THE PETASCALE: NUMERICAL METHODS AND THEIR EFFICIENT AND AUTOMATED IMPLEMENTATION

A Dissertation
Submitted to the Faculty
of
Purdue University
by
Yingchong Situ

In Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

December 2014
Purdue University
West Lafayette, Indiana

ACKNOWLEDGMENTS

I first would like to thank Prof. Zhiyuan Li, my advisor, without whose academic guidance and research assistantship support during the course of this study completion of this dissertation would not have been possible. I would also like to thank the past and current members of the NSF PetaApps project team led by Profs. Gregory Blaisdell, Anastasios Lyrintzis and Zhiyuan Li at Purdue University including Ye Wang, Nitin Dhamankar, Chandra Sekhar Martha, Kurt Aikens and Lixia Liu, collaboration with whom has also been indispensable to this study.

Several institutions have provided financial and computing resource support to the underlying research project of this study. The research project has received direct financial support from the National Science Foundation (NSF) through awards CCF-0811587 and OCI-0904675. Computational experiments in the project have been conducted on Extreme Science and Engineering Discovery Environment (XSEDE) and formerly TeraGrid resources hosted at the National Institute of Computational Sciences (NICS) at the University of Tennessee and the Texas Advanced Computing Center (TACC) at the University of Texas at Austin provided by allocations TG-ASC040044N and TG-ASC090008. Additional computing support has come from the community clusters hosted by the Rosen Center for Advanced Computing (RCAC) at Purdue University.

Last but not least, I would like to express my gratitudes to my family, especially my wife Biying. They have promised and provided every support that I have needed throughout my time at Purdue University.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF ALGORITHMS ix

LIST OF LISTINGS x

ABBREVIATIONS xi

ABSTRACT xiii

1 COMPUTATIONAL AND PROGRAMMING CHALLENGES OF FINITE DIFFERENCE METHODS IN JET ENGINE NOISE PREDICTION . . 1
- 1.1 Practices of jet engine noise prediction 1
- 1.2 Finite difference methods in three-dimensional large eddy simulation of jet engine noise 4
- 1.3 Tridiagonal linear system solvers used in large eddy simulation . . . 11
  - 1.3.1 Transposition method 12
  - 1.3.2 Multiblock method 14
  - 1.3.3 Schur complement method 17
  - 1.3.4 Need for a more scalable tridiagonal linear system solver . . 21
- 1.4 Programming finite difference-based large eddy simulation of jet engine noise 22
  - 1.4.1 Programming challenges in authoring numerical applications 22
  - 1.4.2 Need for dedicated programming tools for finite difference-based large eddy simulation of jet engine noise 25

2 AN EFFICIENT TRIDIAGONAL LINEAR SYSTEM SOLVER BASED ON THE TRUNCATED SPIKE ALGORITHM 30
- 2.1 General SPIKE algorithm applied to tridiagonal linear systems . . . 30
- 2.2 Efficient solution of tridiagonal linear systems using the truncated SPIKE algorithm 39
  - 2.2.1 Basic truncated SPIKE algorithm 39
  - 2.2.2 Truncated SPIKE algorithm enhanced with block Jacobi iteration 40
- 2.3 Theoretical scalability analysis of the truncated SPIKE algorithm . 47
  - 2.3.1 Analysis of the truncated SPIKE algorithm 48
  - 2.3.2 Analysis of the transposition method 51
  - 2.3.3 Analysis of the Schur complement method 54

Page
2.4 Empirical scalability verification 54
3 EFFICIENT IMPLEMENTATION OF FINITE DIFFERENCE-BASED THREE-DIMENSIONAL LARGE EDDY SIMULATION OF JET ENGINE NOISE 57
3.1 Software engineering considerations 57
3.2 Managing the three-dimensional computational spacing partitioning 58
3.3 Communication optimizations 59
3.3.1 Overlapping computation and communication in stencil computation 59
3.3.2 Reducing communication overhead in block Jacobi iteration 60
3.4 Computation optimizations 62
3.4.1 Manipulating four-dimensional arrays using two-dimensional computational kernels 64
3.4.2 Optimizing the two-dimensional computation kernels 65
3.5 Implementation validation 71
3.6 Performance experiments 72
3.6.1 Experimental setup 72
3.6.2 Experimental results 72
4 A PROGRAMMING MODEL BASED ON GENERALIZED ELEMENTAL SUBROUTINES FOR REGULAR GRID-BASED NUMERICAL APPLICATIONS 78
4.1 Introduction 78
4.2 Semantic model and proof-of-concept implementation of generalized elemental subroutines 78
4.2.1 Semantic model 78
4.2.2 Proof-of-concept implementation 82
4.3 Generating optimized code for generalized elemental subroutines . . 83
4.3.1 Loop nest generation 84
4.3.2 Local variable transformation 87
4.3.3 Subroutine invocation aggregation 90
4.3.4 Example 92
4.4 Empirical evaluation 94
4.4.1 Thomas algorithm 96
4.4.2 Sixth-order compact spatial partial differentiation scheme . . 98
4.4.3 Application in finite difference-based three-dimensional large eddy simulation of jet engine noise 100
5 A FUNCTIONAL ARRAY PROGRAMMING MODEL FOR REGULAR GRID-BASED NUMERICAL APPLICATIONS 107
5.1 Introduction 107
5.2 Basic concepts 108
5.3 Types of array definitions 110

Page
5.3.1 Definition by arithmetic 110
5.3.2 Definition by communication 111
5.3.3 Definition by procedure invocation 112
5.3.4 Definition by procedure iteration 113
5.3.5 Examples 114
5.4 Extending procedure invocation and procedure iteration to enable application of lower-dimensional algorithms in higher-dimensional contexts 114
5.4.1 Extension of procedure invocation 114
5.4.2 Extension of procedure iteration 119
5.4.3 Examples 120
5.5 Code generation and optimization 120
5.5.1 Dimensional procedure rewriting 122
5.5.2 Computation–communication interleaving 123
5.5.3 Computation scheduling 127
5.5.4 Array dimension elimination 128
5.5.5 Array coalescing 132
5.6 Empirical evaluation 136
5.6.1 Prototype implementation 136
5.6.2 Benchmarks for experimental evaluation 138
5.6.3 Effect of dimensional procedure rewriting 139
5.6.4 Effect of computation–communication interleaving 142
5.6.5 Effect of array dimension elimination and array coalescing . 144
5.6.6 Applicability in finite difference-based three-dimensional large eddy simulation of jet engine noise 147

6 CONCLUSIONS AND SUGGESTED DIRECTIONS FOR FUTURE EXPLORATION 150
6.1 Conclusions 150
6.2 Suggested directions for future exploration 152
6.2.1 Efficient numerical methods and implementation of finite difference-based large eddy simulation of jet engine noise using implicit time integration 152
6.2.2 Extension of the semantic model and code optimization strategies of the functional array programming model 154

REFERENCES 156

VITA 162

LIST OF TABLES

Table Page

1.1 Meanings of symbols in the governing equation of three-dimensional large eddy simulation in Equation (1.1) 5

2.1 Computed numbers of block Jacobi iterations needed by the truncated SPIKE algorithm 46

2.2 Empirical running times of the truncated SPIKE algorithm in weak scaling experiments on Kraken 56

3.1 Hardware configurations of Kraken and Ranger 73

4.1 Lines of code needed by the individual components of the sixth-order compact spatial partial differentiation scheme in the three implementations 105

5.1 Major algorithmic components of large eddy simulation-based jet engine noise prediction 148

LIST OF FIGURES

Figure Page

1.1 Wavenumber-based comparison of explicit and compact central difference schemes 8

1.2 Transposition method 13

1.3 Overlapping grid lines in the multiblock method 16

1.4 System to be solved by the Schur complement method 18

1.5 System in Figure 1.4 after permutation 19

1.6 System in Figure 1.4 after permutation and partial LU factorization . . 20

2.1 Coefficient matrix of 36-point spatial partial differentiation 31

2.2 Diagonal blocks of the coefficient matrix $\boldsymbol{A}$ in Figure 2.1 33

2.3 Spike matrix $\boldsymbol{S}$ in the spike factorization of the matrix $\boldsymbol{A}$ in Figure 2.1 34

2.4 Elements of the spike matrix $\boldsymbol{S}$ in Figure 2.3 representing coupling between partitions 35

2.5 Coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system in Equation (2.3) . 36

2.6 Coefficient matrix $\tilde{\boldsymbol{S}}$ of the SPIKE reduced system after truncation . . 41

2.7 Shifted partitioning of the coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system 45

2.8 Transposed partitioning of the coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system 45

2.9 Iteration matrix $\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}$ of the residual vector of the SPIKE reduced system 46

2.10 Matrix $\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}$ representing the truncated spike tips removed from matrix $\tilde{\boldsymbol{S}}$ 50

2.11 Transposition process in the transposition method 53

3.1 Application of two-dimensional computation kernels along the three coordinate directions 66

3.2 Speedup achieved by the columnwise Thomas algorithm using different loop tiling factors 70

Figure Page

3.3 Parallel speedup achieved by the large eddy simulation-based jet engine noise prediction application 74

3.4 Parallel efficiency achieved by the large eddy simulation-based jet engine noise prediction application 74

3.5 Performance comparison of the truncated SPIKE algorithm and the transposition method in the large eddy simulation-based jet engine noise prediction application 76

4.1 Speedup achieved by generalized elemental subroutines for the Thomas algorithm over simple repeated subroutine invocation 99

4.2 Speedup achieved by generalized elemental subroutines for the sixth-order compact spatial partial differentiation scheme over simple repeated subroutine invocation 101

4.3 Running times of the three implementations of the sixth-order compact spatial partial differentiation scheme 103

4.4 Running times of the three-dimensional large eddy simulation-based jet engine noise prediction using the three implementations of the sixth-order compact spatial partial differentiation scheme 105

5.1 Abstract syntax tree of the code sketch in Listing 5.8 131

5.2 Normalized running times of benchmark `conpq` with dimensional procedure rewriting disabled 140

5.3 Normalized running times of benchmark `deriv` with dimensional procedure rewriting disabled 140

5.4 Normalized running times of benchmark `matmul` with dimensional procedure rewriting disabled 141

5.5 Normalized running times of benchmark `thomas` with dimensional procedure rewriting disabled 141

5.6 Normalized running times of benchmark `interlv` with greedy computation–communication interleaving 143

5.7 Normalized running times of benchmark `conpq` under different conditions 145

5.8 Normalized running times of benchmark `deriv` under different conditions 145

5.9 Normalized running times of benchmark `matmul` under different conditions 146

5.10 Normalized running times of benchmark `thomas` under different conditions 146

LIST OF ALGORITHMS

Algorithm Page

2.1 General SPIKE algorithm 38

3.1 Stencil computation of right-hand side vectors in spatial partial differentiation and spatial filtering 61

3.2 Block Jacobi iteration using the ready-mode point-to-point communication of MPI 63

4.1 Loop nest generation for generalized elemental subroutines 85

4.2 Local variable dimension reordering for generalized elemental subroutines 88

4.3 Local variable dimension elimination for generalized elemental subroutines 91

LIST OF LISTINGS

Listing Page

3.1 Loop-tiled implementation of the columnwise version of the Thomas algorithm 68

4.1 Generalized elemental subroutine for the vector dot product 81

4.2 Generalized elemental subroutine for stencil computation 93

4.3 Code for stencil computation in the three-dimensional space generated using generalized elemental subroutines 95

4.4 Generalized elemental subroutine for the Thomas algorithm 97

5.1 Specification of the Thomas algorithm in the functional array programming model 115

5.2 Specification of the periodic first-order central difference scheme in the functional array programming model 116

5.3 Application of the Thomas algorithm specified in the functional array programming model in a three-dimensional computational space 121

5.4 Application of the first-order central difference scheme specified in the functional array programming model in a three-dimensional computational space 121

5.5 Rewritten specification of the Thomas algorithm in the functional array programming model 124

5.6 Rewritten specification of the first-order central difference scheme in the functional array programming model 125

5.7 Fortran code for the procedure `ThomasSolve` in Listing 5.5 129

5.8 A code sketch represented using the Fortran syntax 131

5.9 Fortran code for the procedure `ThomasSolve` in Listing 5.5 after array dimension elimination 133

5.10 Fortran code for the procedure `ThomasSolve` in Listing 5.5 after array dimension elimination and array coalescing 137

ABBREVIATIONS

ADI	Alternating-direction implicit
AMD	Advanced Micro Devices
ANTLR	Another Tool for Language Recognition
API	Application programming interface
ArBB	Array Building Blocks
AST	Abstract syntax tree
BDF	Backward difference formula
BFS	Breadth-first search
BiCGSTAB	Biconjugate gradient stabilized
CAA	Computational aeroacoustics
CBC	COIN-OR Branch-and-Cut
CFD	Computational fluid dynamics
CFL	Courant–Friedrichs–Lewy
CLooG	Chunky Loop Generator
CrayPat	Cray Performance Analysis Tool
CS	Computer science
DNS	Direct numerical simulation
DSL	Domain-specific language
FMG	Full multigrid
GLPK	GNU Linear Programming Kit
GMRES	Generalized minimum residual
HPC	High-performance computing
HPF	High Performance Fortran
IEEE	Institute of Electrical and Electronics Engineers

ISL	Integer Set Library
`ispc`	Intel SPMD Program Compiler
LES	Large eddy simulation
LMM	Linear multistep method
MG	Multigrid
MILP	Mixed integer linear programming
MPI	Message Passing Interface
NaN	Not a number
NICS	National Institute of Computational Sciences
NSF	National Science Foundation
OASPL	Overall sound pressure level
PAPI	Performance Application Programming Interface
PFLOPS	Quadrillion floating-point operations per second
QMR	Quasi-minimum residual
RANS	Reynolds-averaged Navier–Stokes equations
RCAC	Rosen Center for Advanced Computing
SaC	Single Assignment C
SIMD	Single-instruction multiple-data
SOA	Structure-of-arrays
SPMD	Single-program multiple-data
TACC	Texas Advanced Computing Center
XSEDE	Extreme Science and Engineering Discovery Environment

ABSTRACT

Situ, Yingchong PhD, Purdue University, December 2014. Scaling Finite Difference Methods in Large Eddy Simulation of Jet Engine Noise to the Petascale: Numerical Methods and Their Efficient and Automated Implementation. Major Professor: Zhiyuan Li.

Reduction of jet engine noise has recently become a new arena of competition between aircraft manufacturers. As a relatively new field of research in computational fluid dynamics (CFD), computational aeroacoustics (CAA) prediction of jet engine noise based on large eddy simulation (LES) is a robust and accurate tool that complements the existing theoretical and experimental approaches. In order to satisfy the stringent requirements of CAA on numerical accuracy, finite difference methods in LES-based jet engine noise prediction rely on the implicitly formulated compact spatial partial differentiation and spatial filtering schemes, a crucial component of which is an embedded solver for tridiagonal linear systems spatially oriented along the three coordinate directions of the computational space. Traditionally, researchers and engineers in CAA have employed manually crafted implementations of solvers including the transposition method, the multiblock method and the Schur complement method. Algorithmically, these solvers force a trade-off between numerical accuracy and parallel scalability. Programmingwise, implementing them for each of the three coordinate directions is tediously repetitive and error-prone.

In this study, we attempt to tackle both of these two challenges faced by researchers and engineers. We first describe an accurate and scalable tridiagonal linear system solver as a specialization of the truncated SPIKE algorithm and strategies for efficient implementation of the compact spatial partial differentiation and spatial filtering schemes. We then elaborate on two programming models tailored for composing regular grid-based numerical applications including finite difference-based LES of jet engine noise, one based on generalized elemental subroutines and the other based on functional array programming, and the accompanying code optimization and generation methodologies. Through empirical experiments, we demonstrate that truncated SPIKE-based spatial partial differentiation and spatial filtering deliver the theoretically promised optimal scalability in weak scaling conditions and can be implemented using the two programming models with performance on par with handwritten code while significantly reducing the required programming effort.

1 COMPUTATIONAL AND PROGRAMMING CHALLENGES OF FINITE DIFFERENCE METHODS IN JET ENGINE NOISE PREDICTION

1.1 Practices of jet engine noise prediction

In the recent decades, aviation has assumed a critical role in supporting global economic growth thanks to its ability to transport people and goods at speeds unparalleled by other means of transportation. Aircraft noise, a byproduct of aviation, however, is proving to have a tangible negative impact on the overall benefit of aviation, which ranges from physical damages to the human body to financial penalties imposed on its originators and costs of noise mitigation measures. The demand for quieter aircraft has expanded the already fierce competition between major aircraft manufacturers to a new arena. In civil aviation, low-noise in-flight experience is gaining emphasis in advertising campaigns targeting ordinary customers. In military scenarios, injury-incurring sound levels near aircraft also corroborate the necessity of noise reduction in aircraft design.

Traditionally, prediction of sound levels generated by jet engines is conducted using theoretical derivations and empirical experiments. The modern discipline of aeroacoustics originates from [41, 42], which date back to the early 1950s, when researchers started to subject the mechanisms of noise generation by jet engines to scientific scrutiny. The theoretical approaches of aeroacoustics are based on acoustic analogies, where the Navier–Stokes equations are recast as wave equations which describe perturbations in air density and pressure in terms of some acoustic source terms. Based on their formulations, the acoustic source terms are then likened to idealized noise sources, from which theoretical results are derived such as the celebrated Lighthill's noise scaling law that the radiated power scales as the eighth power of the jet velocity. What challenges the rigorousness of theoretical aeroacoustics is the fact the acoustic source terms in the recast Navier–Stokes equations are ultimately unknown variables themselves, and no formal relationship between them and the noise-generating turbulent structures have been established. In the mean time, empirical experiments for determining noise levels generated by jet engines typically involve putting scaled models of jet engines inside anechoic chambers and measuring the sound levels at different locations using microphones. While empirical experiments can lead to realistic measurements, the monetary and time costs of manufacturing models, ensuring the result accuracy and operating the apparatus are prohibitively expensive for rapid design iteration.

Since the 1980s, computational aeroacoustics (CAA) has developed as a robust and accurate tool that complements the traditional theoretical and experimental approaches for jet engine noise prediction. CAA is a relatively new discipline of computational fluid dynamics (CFD) which focuses on prediction of sound levels generated by aircraft airframes and engines. It applies the principles of theoretical acoustic analogies and uses realistic CFD simulations to resolve the precise dynamics of the acoustic source terms. Unlike the general practices in the broader field of CFD, CAA relies heavily on the accurate prediction of small-amplitude acoustic fluctuations and their correct propagation to the far field. To accomplish its mission, CAA imposes tight restrictions on the underlying numerical methods. An appropriate numerical method for CAA is expected to provide high accuracy and good spectral resolution while maintaining a low level of dispersion and diffusion errors. Such stringent requirements pose serious challenges to CAA researchers.

The state of the art of CAA prediction of far-field jet engine noise is based on time-dependent CFD simulation of the noise-generating turbulent flows. Postprocessing integral methods based on acoustic analogies [47] are then used to propagate the near-field noise computed by the CFD simulation to the observer location at the far field. Traditionally, such numerical simulations are carried out by solving the Reynolds-averaged Navier–Stokes equations (RANS). In RANS, the effect of the entire spectrum of turbulent scales is represented by empirical turbulence models. Capturing the effect of all turbulent scales through modeling enables RANS to be computationally inexpensive but sacrifices physical fidelity. Turbulence information of significance to the acoustics is lost during the averaging process that characterizes RANS. At the opposite end to the methodology of RANS, direct numerical simulation (DNS) aims for the highest possible physical fidelity by explicitly resolving all relevant turbulent length scales. However, the computational cost of DNS grows rapidly as the Reynolds number increases due to the fact that higher Reynolds numbers require finer grid spacing to fully capture the dynamics of the relevant turbulent scales. Furthermore, finer grid spacing requires proportionally smaller time steps during time integration as prescribed by the Courant–Friedrichs–Lewy (CFL) condition to preserve numerical stability. As a consequence, DNS is typically limited to turbulent flows of low Reynolds numbers. For CAA problems of practical interest, neither RANS nor DNS provides a satisfactory solution that offers both accuracy and efficiency.

Large eddy simulation (LES) embodies a pragmatic eclecticism between RANS and DNS. In LES, small turbulent scales, which have a more universal behavior, are modeled as in RANS, whereas large turbulent scales, which are more flow-dependent, are completely resolved as in DNS. LES exploits the fact that small-scale turbulence tends to be self-similar and thus is very suitable for modeling; in the meantime, it retains all the physical characteristics of the larger eddies. Such a philosophy enables LES to use coarser grids than DNS to significantly reduce the computational cost but avoid the loss of valuable turbulent information as in RANS simulations. We refer the reader to [26, 63] for general treatments on LES.

Using the CAA methodology coupled with LES described above, many researchers have conducted simulations for jet engine noise prediction. References [76, 77, 78] use a multiblock solver with overlapping grid partitions to perform high-fidelity simulations of subsonic jets with nozzles, both with and without chevrons, on grids with up to 500 million grid points. References [11, 12] use high-order methods on meshes with 252 million points to study the effect of important parameters on the noise in subsonic conditions. Compared to subsonic jet simulations, there are comparatively few high-fidelity supersonic jet simulations [1]. High-accuracy simulations which include the turbulent boundary layer in the nozzle have recently been completed to study of impact of beveling the nozzle exit [3]. While structured solvers remain the dominant tool for LES, there has been a push towards unstructured solvers recently in search for greater flexibility in meshing and simplicity in geometric modeling, though at the expense of accuracy. Existing work includes simulations of round jets from converging–diverging nozzles with chevrons [14, 50] and rectangular jets with and without chevrons [55]. These simulations utilize hundreds of millions of grid points and up to 163,840 processors on the Intrepid cluster hosted at the Argonne National Laboratory. Reference [34] provides the details of their numerical methods.

1.2 Finite difference methods in three-dimensional large eddy simulation of jet engine noise

The essence of three-dimensional LES [26, 63] for jet engine noise prediction is to solve a system of Favre-filtered unsteady, compressible nondimensionalized Navier–Stokes equations formulated in the conservative form. Because the Navier–Stokes equations are mathematically continuous, for practical purposes, the region of interest of the physical domain is mapped to a computational space in which the Navier–Stokes equations are discretized. In [74], the system of Navier–Stokes equations expressed in the coordinate system of the computational space is succinctly written as

$$\frac{1}{J}\frac{\partial \boldsymbol{Q}}{\partial t} + \frac{\partial}{\partial \xi}\left(\frac{\boldsymbol{F} - \boldsymbol{F_v}}{J}\right) + \frac{\partial}{\partial \eta}\left(\frac{\boldsymbol{G} - \boldsymbol{G_v}}{J}\right) + \frac{\partial}{\partial \zeta}\left(\frac{\boldsymbol{H} - \boldsymbol{H_v}}{J}\right) = \mathbf{0}. \tag{1.1}$$

The meanings of the symbols in Equation (1.1) are listed in Table 1.1. Reference [74] provides a detailed description of their precise definitions. Equation (1.1) applies uniformly to each individual point in the computational space. Each term in the equation is a vector of five components parameterized on the coordinates (ξ, η, ζ). To discretize the equation for numerical solution, the physical domain is represented by a three-dimensional curvilinear grid, and the continuous time is replaced by discrete time

Table 1.1.
Meanings of symbols in the governing equation of three-dimensional large eddy simulation in Equation (1.1)

Symbol	Meaning
t	Time
ξ, η, ζ	Generalized curvilinear coordinates in the computational space
J	Jacobian determinant of the coordinate transformation from the physical domain to the computational space
$\boldsymbol{Q}$	Vector of conservative flow variables (density, three components of momentum and energy)
$\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H}$	Inviscid flux vectors in the ξ-, η- and ζ-directions
$\boldsymbol{F_v}, \boldsymbol{G_v}, \boldsymbol{H_v}$	Viscous flux vectors in the ξ-, η- and ζ-directions

steps. In order to simplify the formulation of the discretized problem, it is a common practice to choose a mapping from the physical domain to the computational space such that the curvilinear grid in the physical domain is mapped to a uniform grid in the computational space with unit grid spacing. The immediate consequence of such discretization is that the spatial partial derivatives in Equation (1.1) are approximated by finite differences.

Equation (1.1) can be rearranged as

$$\frac{\partial \boldsymbol{Q}}{\partial t} = \mathbf{RHS}(\boldsymbol{Q}; t) \tag{1.2}$$

where

$$\mathbf{RHS}(\boldsymbol{Q}; t) = -J \left(\frac{\partial}{\partial \xi} \left(\frac{\boldsymbol{F} - \boldsymbol{F_v}}{J} \right) + \frac{\partial}{\partial \eta} \left(\frac{\boldsymbol{G} - \boldsymbol{G_v}}{J} \right) + \frac{\partial}{\partial \zeta} \left(\frac{\boldsymbol{H} - \boldsymbol{H_v}}{J} \right) \right). \tag{1.3}$$

In LES, Equation (1.1) is solved by integrating $\mathbf{RHS}(\boldsymbol{Q}; t)$ over time starting from a prescribed initial condition. For the purpose of jet engine noise prediction, the integration process is divided into two stages. The first stage propagates the effect of the inflow boundary condition in the downstream direction to drive the transient flow state induced by the initial condition out of the simulated region. It concludes when the flow field reaches a statistically stable state. The second stage takes up where the first stage left off and continues the integration, during which the state of the flow field is sampled periodically for later use in the acoustic postprocessing. It terminates when sufficient samples have been collected.

The choice of the method for integrating $\mathbf{RHS}(\boldsymbol{Q}; t)$ over time has fundamental ramifications on the rest of the numerical method of LES. Explicit methods, of which the Runge–Kutta family of integration schemes is a prime example, usually leads to greater overall simplicity in the numerical method as it requires only straightforward evaluation of $\mathbf{RHS}(\boldsymbol{Q}; t)$ with different values of $\boldsymbol{Q}$ and t. In contrast, implicit methods, of which the Beam–Warming scheme [8] is a well-known representative, requires an embedded iterative scheme to solve a system of nonlinear equations in order to advance from one time step to the next. In general, explicit methods are inferior in time-dependent numerical stability. Consequently, time integration must proceed in smaller time steps than in implicit methods and thus takes more time steps to reach the same simulation time. On the contrary, the computational cost per time step of implicit methods is more expensive than that of explicit methods. Linear systems arising from linearization of the nonlinear iteration system also to tend to be ill-conditioned. Their accurate, reliable and efficient solution is itself a major challenge. For the purpose of this study, we assume that time integration is performed using a Runge–Kutta method.

Depending on the parameters of the actual physical problem to be solved (e.g., heated or unheated jet, with or without a nozzle), the boundary conditions specified at domain boundaries change as does the size of the simulated region. In the downstream portion of the grid, the nonreflective boundary conditions from [9, 70] are frequently used. When the nozzle is not simulated, a laminar inflow velocity profile can be used, and a vortex ring forcing approach can be employed to promote the transition of the shear layer to turbulent flow [10]. When the nozzle is explicitly included in the simulation, there are adiabatic viscous [35, 36], extrapolation-based [44] and approximate turbulent wall model [2] boundary conditions that can be utilized; a digital filter-based turbulent inflow boundary condition can also be used to prescribe an initial turbulent wall boundary layer [19, 37, 73, 85].

Evaluation of $\mathbf{RHS}(\boldsymbol{Q}; t)$ as defined in Equation (1.3) relies on spatial partial differentiation along each of the ξ-, η- and ζ-directions. For jet engine noise prediction, compact finite difference schemes described in [39] are usually preferred over the traditional central difference schemes. As illustrated in Figure 1.1, compared to the corresponding central difference schemes of the same orders, compact difference schemes have lower dispersion errors as they better approximate exact differentiation under Fourier analysis. Unlike the explicit central difference schemes, compact difference

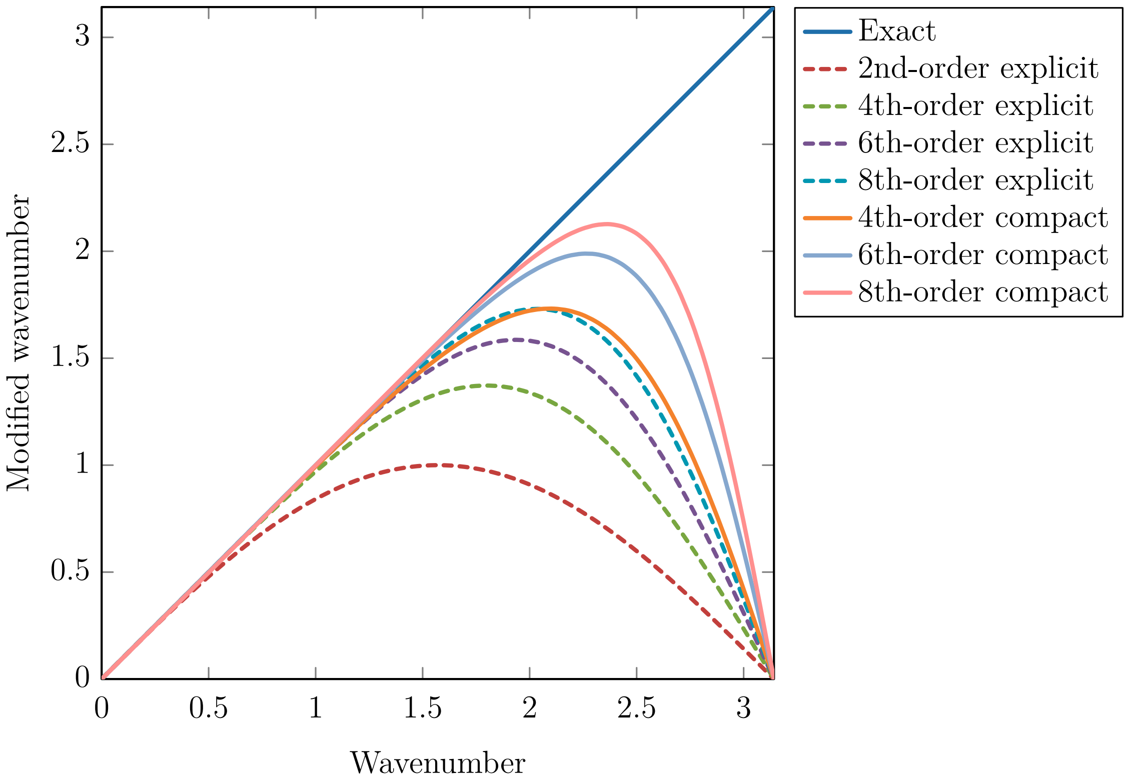


Figure 1.1. Wavenumber-based comparison of explicit and compact central difference schemes

schemes are implicit. Consider the nondissipative sixth-order compact difference scheme for example. Assuming unit grid spacing, it is formulated as

$$\frac{1}{3}f'_{i-1} + f'_i + \frac{1}{3}f'_{i+1} = \frac{7}{9}(f_{i+1} - f_{i-1}) + \frac{1}{36}(f_{i+2} - f_{i-2}) \tag{1.4}$$

where f_i is the value of the function f to be differentiated at the ith grid point in a grid line, and f'_i is the approximation of the first spatial partial derivative of f at the same grid point. Due to the five-point stencil in its right-hand side, Equation (1.4) cannot be applied to grid points at domain boundaries. For the boundary grid points at $i = 1, 2$, it is replaced by the following third-order one-sided and fourth-order central compact difference schemes [74]:

$$f'_1 + 2f'_2 = -\frac{5}{2}f_1 + 2f_2 + \frac{1}{2}f_3, \tag{1.5a}$$

$$\frac{1}{4}f'_1 + f'_2 + \frac{1}{4}f'_3 = \frac{3}{4}(f_3 - f_1). \tag{1.5b}$$

Correspondingly, the reflected formulations of Equations (1.5a) and (1.5b) are applied to the boundary grid points at $i = N-1, N$ where N is the number of grid points in the grid line. Equations (1.4) and (1.5) give rise to a tridiagonal linear system which must be solved to obtain the value of each f'_i. Similar tridiagonal linear systems also occur in the formulations of the fourth- and eight-order compact finite difference schemes.

Numerical artifacts may arise from boundary conditions, unresolved turbulent scales and mesh nonuniformity. They can exert negative impact on the numerical stability of LES as analyzed in [38]. Therefore, it is necessary to perform spatial filtering to suppress these unfavorable artifacts. Reference [81] suggests the following symmetric sixth-order three-term low-pass filter:

$$\alpha_f \bar{f}_{i-1} + \bar{f}_i + \alpha_f \bar{f}_{i+1} = \sum_{n=0}^{3} \frac{a_n}{2}(f_{i-n} + f_{i+n}) \tag{1.6}$$

where $\bar{f}_i$ is the filtered value of f_i, and α_f is a user-defined parameter. The coefficients a_n depend on α_f, and we refer the reader to [81] for the details of their definitions. The parameter α_f must satisfy the inequality $|\alpha_f| \leq 0.5$. The farther α_f is away from 0.5, the stronger the filter is; when $\alpha_f = 0.5$, the filter has no effect. As with the sixth-order compact difference scheme, Equation (1.6) cannot be applied to the grid points at domain boundaries due to the seven-point stencil in its right-hand side. Reference [81] suggests the following alternative formulation in the form of a biased seven-point stencil for the grid points at $i = 2, 3$:

$$\alpha_f \bar{f}_{i-1} + \bar{f}_i + \alpha_f \bar{f}_{i+1} = \sum_{n=1}^{7} a_{n,i} f_i \tag{1.7}$$

where $a_{n,i}$ are also coefficients which depend on α_f. Similarly, the reflected formulation of Equation (1.7) is applied to the grid points at $i = N-3, N-2$. The grid points at the very domain boundaries, i.e., where $i = 1, N$, are left unfiltered. In sum, the above spatial filtering scheme also gives rise to a tridiagonal linear system.

During simulations of supersonic jets, shock waves may develop in the flow field and cause the values of the flow variables to exhibit discontinuities in some locations. The high-order compact spatial partial differentiation and spatial filtering schemes described above, when applied without modifications, can introduce spurious oscillations near the discontinuities. In order to appropriately capture the shock waves, they can be extended to incorporate characteristic filters [1, 25, 45, 86] into their formulations. In brief terms, this involves locating discontinuities in the flow field using a shock detector and reducing the orders of the spatial filtering scheme in the neighborhoods of those locations. Compared to simulations of subsonic jets, where shock waves are absent, the linear systems resulting from these numerical schemes change in space and time. However, they still preserve the tridiagonal and stencil-based formulations of the original schemes.

1.3 Tridiagonal linear system solvers used in large eddy simulation

In three-dimensional LES for jet engine noise prediction, it is a major computational task to solve the tridiagonal linear systems arising from spatial partial differentiation and spatial filtering. To put it into perspective, consider the full numerical method of three-dimensional LES formulated in [74]. Reference [74] uses the classical fourth-order Runge–Kutta method to integrate $\mathbf{RHS}(\boldsymbol{Q};t)$ in Equation (1.3) over time. Within each time step, the three spatial partial differential operators, $\frac{\partial}{\partial\xi}$, $\frac{\partial}{\partial\eta}$ and $\frac{\partial}{\partial\zeta}$, are each evaluated nine times. At the end of the time step, the flow field is spatially filtered along each of the ξ-, η- and ζ-directions. Assuming that the dimensions of the grid are $N \times N \times N$, each of these operations is applied to $5N^2$ vectors of order N where the constant factor 5 comes from the number of flow variables. This translates to tridiagonal linear systems being solved thirty times per time step, each time with $5N^2$ right-hand side vectors. Hence, an accurate and efficient solver for those linear systems plays a critical role in the success of three-dimensional LES.

On sequential computing platforms, solving tridiagonal linear systems is straightforward as the classical Thomas algorithm, a variant of LU factorization specialized for tridiagonal linear systems, well serves the purpose. However, the computational power and storage space required by realistic jet engine noise prediction exceed the capacities of even today's most advanced monolithic computing devices by a wide margin and necessitate distribution among multiple processors comprising a parallel computing platform. Therefore, dedicated parallel tridiagonal linear system solvers must be employed to operate on the distributed data and fully exploit the aggregate computational power of the computing platform. To that end, CAA researchers have traditionally utilized methods including the transposition method [74], the multiblock method [87] and the Schur complement method [40].

1.3.1 Transposition method

Recognizing the success of the Thomas algorithm as an effective solver for tridiagonal linear systems, the transposition method seeks to use it without modification. However, this is immediately met with the hurdle that data belonging to one right-hand side vector can be scattered across multiple processors. The strategy adopted by the transposition method to overcome this difficulty is to temporarily gather the scattered data onto a single processor, solve the tridiagonal linear system and redistribute the data onto multiple processors.

Reference [74] describes an implementation of the transposition method applied to three-dimensional jet engine noise simulations with rectangular computational spaces. Given an $N_\xi \times N_\eta \times N_\zeta$ Cartesian grid and p processors, it partitions the grid into p slabs of dimensions $N_\xi \times N_\eta \times (N_\zeta/p)$ and assigns each slab to a distinct processor. Tridiagonal linear systems along the ξ- and η-directions can be solved by directly applying the Thomas algorithm to individual vectors with perfect parallelism because the vectors are not partitioned among multiple processors. However, the same method cannot be applied to the systems along the ζ-direction, where each right-hand side vector is distributed among the p processors. In order to make the Thomas algorithm applicable, the grid partitioning is temporarily transposed over the η–ζ plane as illustrated in Figure 1.2. After the transposition, the grid becomes partitioned in the η-direction. In the meantime, partitions of each right-hand side vector along the ζ-direction are gathered onto a single processor and concatenated, after which the Thomas algorithm becomes applicable. The transposition method then proceeds to solve the tridiagonal linear systems along the ζ-direction. Upon completion of the Thomas algorithm, the solution vectors go through a reverse transposition process to be redistributed among the processors so that they become partitioned in the ζ-direction, restoring the original data distribution scheme.

Reference [74] reports that the transposition method was able to attain linear scaling on up to 160 processors on LeMieux, a Compaq AlphaServer SC45 cluster

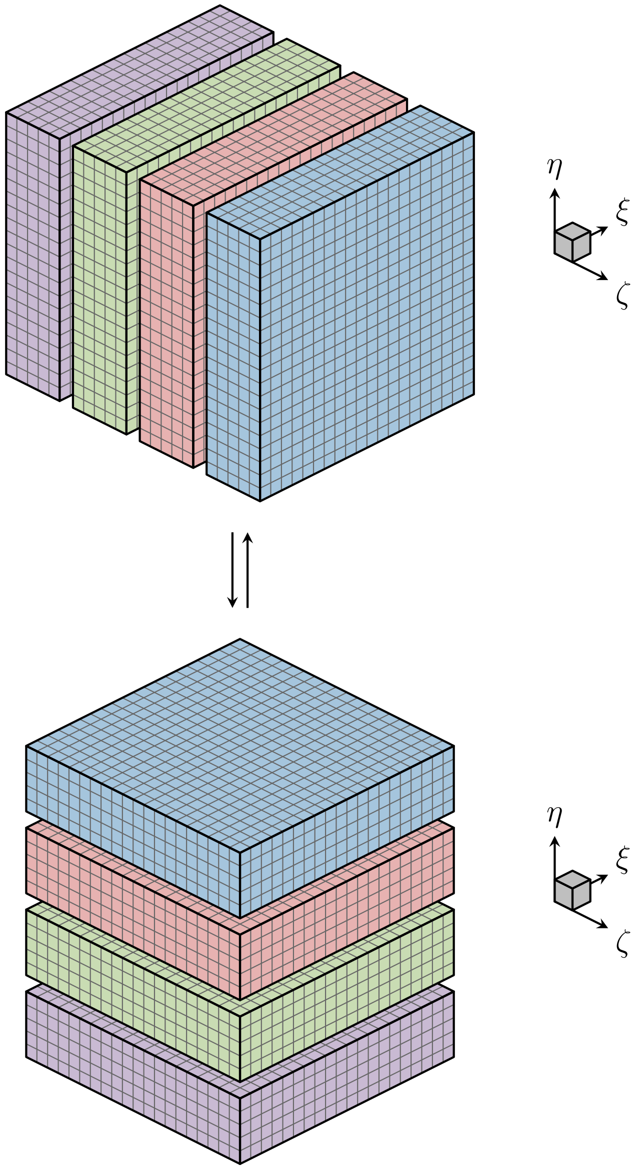


Figure 1.2. Transposition method

hosted at the Pittsburgh Supercomputing Center. In a scalability test performed on an unnamed IBM SP POWER3 cluster hosted at Indiana University, the method achieved a 76 % parallel efficiency on 160 processors relative to a baseline of 20 processors. However, over a decade has passed since those experiments were conducted, which necessarily damages the credibility of any performance claims derived from them in the context of today's supercomputing landscape. From a theoretical perspective, the transposition method has two major disadvantages. First, it imposes the constraint that $p \leq N_\zeta$ since each processor needs to possess at least one plane of grid points. This limits the amount of parallelsim that the method can exploit because N_ζ rarely exceeds 1,000 even in today's high-fidelity jet simulations. Second, the transposition process essentially shuffles data of the entire flow field among all processors, which implies a significant communication penalty. Even though the first disadvantage can be mitigated by partitioning the grid also along the ξ- and η-directions, the second disadvantage is not circumventable. In fact, [48] reports that the transposition-induced communication cost can be as high as 50 % of the total computational cost of LES, which corroborates the above theoretical argument.

1.3.2 Multiblock method

While the transposition method aims to solve the tridiagonal linear systems accurately even at the expense of global data shuffles among all processors, the multiblock method opts to trade numerical accuracy for program efficiency. The strategy which it adopts is to truncate the compact spatial partial differentiation and spatial filtering schemes to the boundaries of individual partitions of the grid, where they are replaced by one-sided and lower-order formulations. Due to such truncation, the resulting tridiagonal linear systems no longer span multiple grid partitions and thus can be solved straightforwardly using the Thomas algorithm. However, in its crudest form, the method will lead to unphysical simulation results because truncation of the spatial partial differentiation and spatial filtering schemes essentially disconnects the grid partitions from one another and effectively prevents turbulent fluctuations from propagating in the flow field.

In order to allow the turbulence information to be exchanged between grid partitions, a practical implementation of the multiblock method uses overlapping grid partitions. Typically, for the sixth-order compact spatial partial differentiation and spatial filtering schemes, neighboring grid partitions overlap by four planes of grid points as illustrated in Figure 1.3. Furthermore, within each grid partition, the four grid points at the boundaries of each grid line, two at each end, are declared as *fringe points*. The values of the flow variables and their spatial partial derivatives at the fringe points are replaced with those at their coinciding counterparts in neighboring grid partitions. This serves to let the turbulence information propagate from each grid partition to its neighbors. It also replaces the values computed by the one-sided and lower-order formulations for boundary grid points with more accurate values computed by the higher-order formulations for interior grid points.

The multiblock method is experimented in [87] using benchmark aeroacoustics problems and LES of a turbulent jet plume with good simulation results. It is also applied in [75] for simulation of a chevron nozzle jet flow, where neighboring grid partitions overlap by seven planes of grid points. However, preliminary experimental results in [48] show that the multiblock method is only slightly faster than the transposition method. This can be attributed to the redundant computation introduced by overlapping grid partitions, especially when the grid partition size is relatively small. For example, when neighboring grid partitions overlap by four planes of grid points, if the transposition method uses grid partitions of dimensions $32 \times 32 \times 32$, the dimensions of the grid partitions used by the multiblock method will become $36 \times 36 \times 36$, which is over 40 % larger in total size.

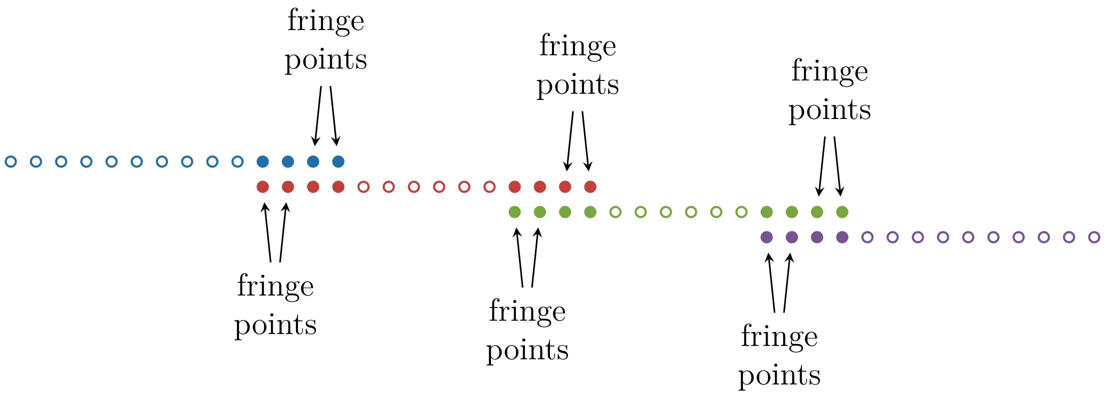

fringe points
fringe points
fringe points
fringe points
fringe points
fringe points

Figure 1.3. Overlapping grid lines in the multiblock method

1.3.3 Schur complement method

Development of the Schur complement method is driven by the desire to overcome the shortcomings of the transposition method and the multiblock method, namely the high communication cost of the former and the compromised numerical accuracy and redundant computation of the latter. Mathematically, the Schur complement method is equivalent to the traditional divide-and-conquer parallel narrow-banded linear system solver applied to tridiagonal linear systems. The algorithm used by the `PDDTTRF` and `PDDTTRS` subroutines of ScaLAPACK [15] is a variant of the method.

In order to solve a tridiagonal linear system, the Schur complement method designates the last (or first) grid point of a grid line within each grid partition as the interface for capturing the interaction between neighboring grid partitions. It then attempts to reduce the coupling between neighboring grid partitions to that between the interface grid points using LU factorization. Figures 1.4, 1.5 and 1.6 illustrate the process. The partitioned system is shown in Figure 1.4, where the elements directly related to the interface grid points are represented by filled circles. In Figure 1.5, those elements are permuted to the end of the system to separate them from the rest. In an actual implementation, such permutation and separation need to be carried out only conceptually. The upper left part of Figure 1.5 can obviously be LU-factorized with perfect parallelism. The result of the factorization is then used to form a Schur complement as shown in the lower right part of Figure 1.6. As with the original system, the Schur complement is also tridiagonal. It captures precisely the coupling between the interface grid points with the their interdependence with the other grid points eliminated. Hence, the Schur complement method effectively reduces a tridiagonal linear system of order N to a smaller one of order p where p is the number of processors. After the Schur complement system is solved, the values of the elements in the solution vector associated with the noninterface grid points can be retrieved again with perfect parallelism.

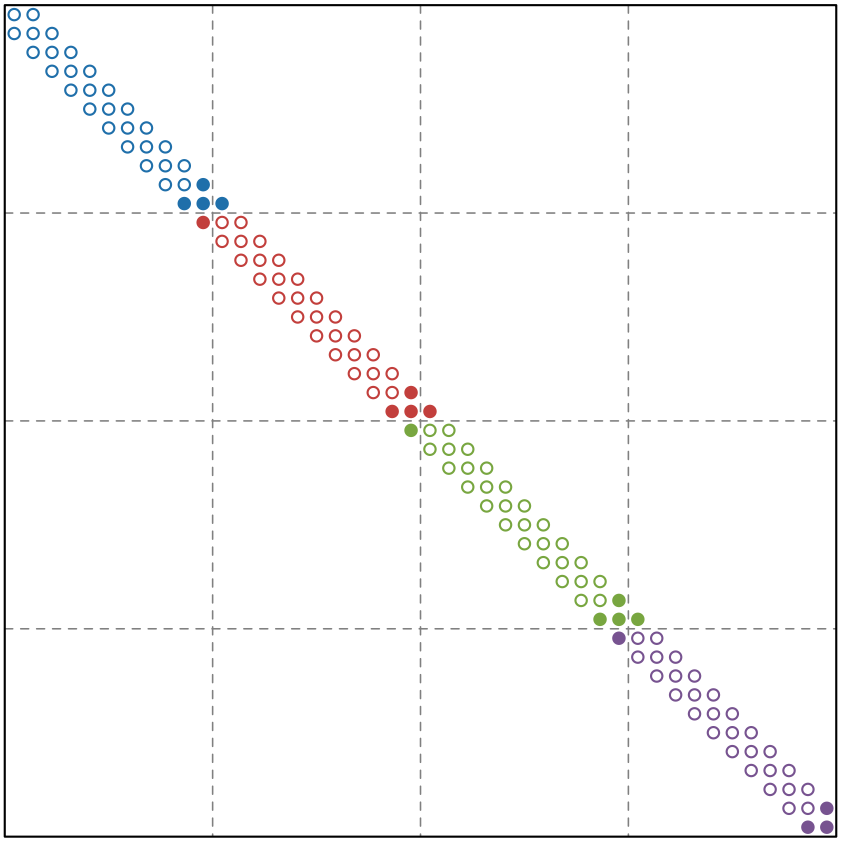


Figure 1.4. System to be solved by the Schur complement method

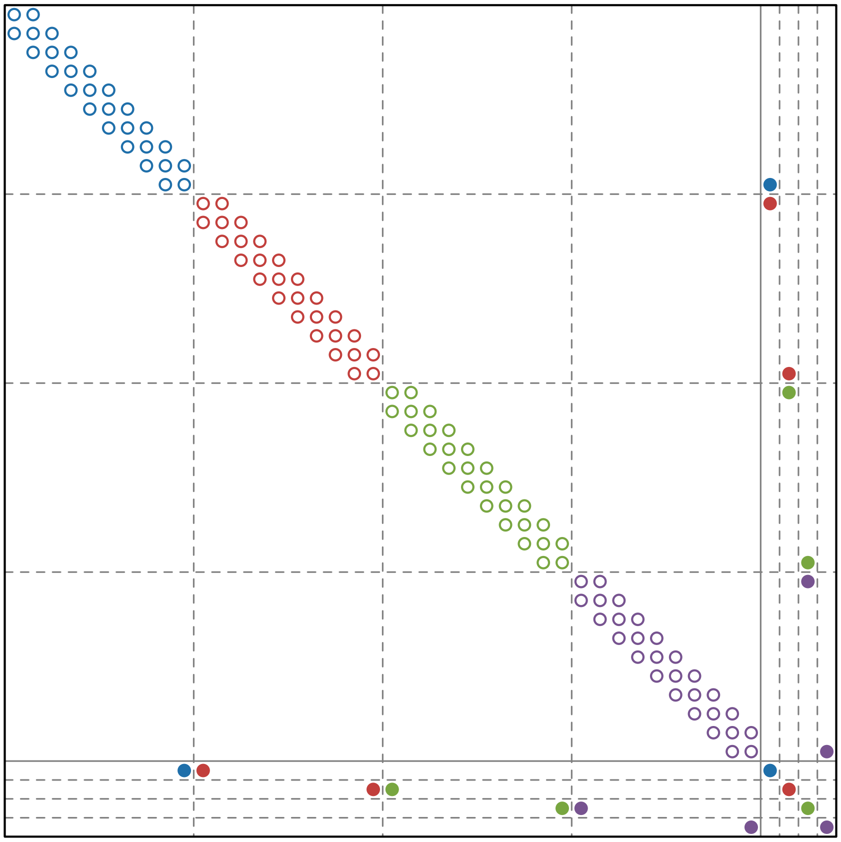


Figure 1.5. System in Figure 1.4 after permutation

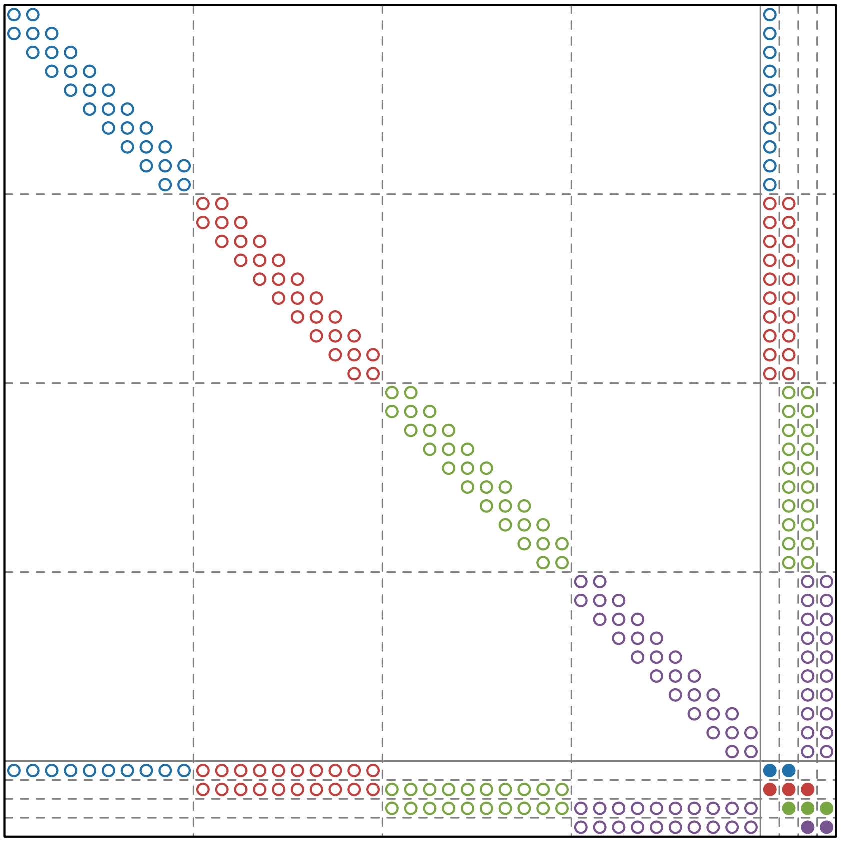


Figure 1.6. System in Figure 1.4 after permutation and partial LU factorization

The exact method used to solve the Schur complement system determines the efficiency of the Schur complement method for any fixed grid partition size. Reference [40] gathers the system onto a single processor and solves it sequentially. This is leads to a suboptimal $\mathcal{O}(p)$ asymptotic running time but can benefit from optimized software and hardware implementations of collective communication. Reference [52], which uses direct forward substitution instead of LU factorization to form the Schur complement system, implements the Thomas algorithm directly on multiple processors even though there is no parallelism at all. It also leads to an $\mathcal{O}(p)$ asymptotic running time. The `PDDTTRF` and `PDDTTRS` subroutines of ScaLAPACK use parallel cyclic reduction [31], which has an $\mathcal{O}(\log p)$ running time but must be manually synthesized from point-to-point communication.

1.3.4 Need for a more scalable tridiagonal linear system solver

While the transposition method, the multiblock method and the Schur complement method briefly described in sections 1.3.1, 1.3.2 and 1.3.3 have been successfully applied to jet engine noise prediction in existing literature, the emergence of petascale high-performance computing (HPC) platforms have recreated the need to develop new tridiagonal linear system solvers. The most conspicuous characteristics of today's petascale HPC platforms for scientific computing is their large numbers of processor cores. In the June 2014 edition of the Top500 List [72], every one of the top ten systems, which all achieve at least 3.14 PFLOPS in the LINPACK benchmark, is equipped with at least 220,000 processor cores except for the sixth-placed Piz Daint, which also boasts 115,984 processor cores. As the development of interconnect technologies continue to trail the growth of processor speeds, in the absence of algorithmic improvements, the dramatically increased numbers of processor cores in these petascale scientific computing platforms will significantly inflate the cost of communication between processors relative to the total computational cost.

By inspecting qualitatively the communication cost of the three methods mentioned in sections 1.3.1, 1.3.2 and 1.3.3 for solving tridiagonal linear systems on parallel computing platforms, we can conclude that the transposition method is very unlikely to successfully adapt to today's petascale computing systems due to the global data shuffles in the transposition process. On the contrary, the multiblock method is the most scalable because only a constant of data per right-hand side vector needs to be exchanged between neighboring grid partitions, and it does not rely on the globally synchronizing collective communication. The Schur complement method, assuming that the reduced Schur complement system is solved using parallel cyclic reduction, has a communication cost which lies between those of the previous two methods. The embedded parallel cyclic reduction has the same synchronizing effect and asymptotic cost as an all-to-all reduction. An ideal tridiagonal linear system solver for jet engine noise prediction should offer the best of the two worlds—the low communication cost of the multiblock method and the high numerical accuracy of the transposition method and the Schur complement method.

1.4 Programming finite difference-based large eddy simulation of jet engine noise

1.4.1 Programming challenges in authoring numerical applications

In the field of scientific computing, it is a common practice for researchers and engineers to rely on established external software libraries to achieve high performance as well as high reliability in their numerical applications. Examples of such software libraries include LAPACK [5] for dense numerical linear algebra problems and FFTW [24] for discrete Fourier transforms. In many situations, this can be a successful strategy because these reputed software libraries are typically fine-tuned for performance and broadly tested for reliability in real-world environments. However, as with any other code which originates from external sources, these software libraries also come with their own collections of limitations. Oftentimes, they tend to be general-purpose because they are generally intended to be competitive, in terms of both functionality and performance, for the generic classes of numerical problems which they target. In practice, however, specialized software tailored for the specific numerical problems occurring in the context of concrete numerical applications can usually take advantage of the special numerical properties exposed by the numerical applications and deliver performance which is not otherwise attainable by generic software. Furthermore, efficient integration of external software libraries into existing in-house code written by research and engineers can also prove to be plagued with difficulties. An example of the most prominent obstacles is the incompatibility of data structures, i.e., a numerical application stores its data in a memory layout which is not immediately accepted by an external software library. Bridging such an interface mismatch in a simplistic fashion may lead to additional data copying and shuffling, which incurs extra performance overhead and increases the memory footprint of the application.

These limitations which arise from utilization of external software libraries frequently force researchers and engineers to investigate new numerical algorithms and new implementations of their own which cater to the specific requirements of their numerical applications. However, neither is such algorithmic and implementation exploration free of difficulties of its own. Quite the contrary, it entails significant programming challenges to the researchers and engineers, especially those whose primary academic backgrounds are not computer science (CS). This is because converting a numerical algorithm from its descriptions in papers and/or textbooks into code which is ready to be integrated into a numerical application is a nontrivial undertaking. The conversion process involves translating the algorithm into a sequence of language constructs which is not only correct in semantics but also efficient in performance. In particular, the programmer needs to consider multiple issues which have crucial impact on the empirical performance of the algorithm:

- design of data structures which not only smoothly interface with existing data structures of the numerical application but also enable efficient implementation of the numerical algorithm,

- expression of computational operations in forms which induce the compiler into generating efficient code for the target computing platform,
- organization of communication operations in parallel algorithms in order that the computation–communication parallelism is effectively exploited.

Correct programming decisions regarding these issues rely on comprehensive knowledge about the operational behaviors and performance characteristics of the different components of diverse scales of the computing platform which include the processor microarchitecture, the memory hierarchy and the interconnection network. This renders them arguably difficult topics even for researchers and engineers with a primary academic background in CS.

The fact that non-CS researchers and engineers have to double as professional programmers for the sake of scientific computing-related research has proved to take its toll on the research itself. When we audited the codebases which backed the work in [40] and [74], we discovered programming issues ranging from maintainability-damaging antipatterns in software engineering to performance-punishing misuses of communication primitives. Furthermore, judging from the sheer sizes of the codebases, we can conclude that significant programming efforts were invested during their development. While the former point may be isolated occurrences specific to these two codebases, the latter point is very likely true for many other large-scale numerical applications. With the emergence and proliferation of petascale HPC platforms, we can expect to witness a continued trend of growth in the level of sophistication and complexity of numerical applications for scientific computing as researchers and engineers strive to adapt their applications to those computing platforms. The ensuing increase in the programming burden will contribute an even greater amount of distraction than it does now and prevent the researchers and engineers from focusing on their “real sciences”.

1.4.2 Need for dedicated programming tools for finite difference-based large eddy simulation of jet engine noise

Specifically to large eddy simulation-based jet engine noise prediction, the challenges which arise in the programming practices of researchers and engineers mainly originate from two sources. The first is the intrinsic complexity of the underlying numerical methods, while the second is the lack of dedicated programming tools. Simplification of the numerical methods of jet engine noise prediction depends on the advancement of the science of aeroacoustics. Comparing with the empirical measurements obtained from experiments using physical models of jet engines in anechoic chambers, the latest methodologies of high-fidelity LES of jet engine noise [1, 48] still produce simulation results which contain errors on the order of a few decibels in the predicted overall sound pressure level (OASPL). In terms of the perceived loudness of jet engine noise, predictions with errors of such magnitudes can be considered excellent approximations of the reality since the human perception system generally has a logarithmic response to the strength of stimuli, a phenomenon reflected by the expression of the OASPL in the logarithmic decibel scale. When it comes to the physical damage to the human body which the noise can cause, however, the same predictions have to be deemed in need of significant improvement. The reason lies in the fact that the amount of physical damage is determined by the total mechanical energy level, which is measured in the linear scale as opposed to the logarithmic scale. Given the lack of a fully developed theory of the generation mechanisms of jet engine noise in aeroacoustics of the current time, simplification of the numerical methods is unlikely to occur at least in the short term. This topic is also beyond the scope of this study. Consequently, in order to simplify the programming tasks in LES of jet engine noise from the perspective of CS, we have to investigate new programming tools which provide dedicated support for this particular type of numerical applications.

Currently there exist commercially and freely available software libraries and applications which attempt to provide one-stop solutions to CFD simulations. For instance, OpenFOAM [32] is a C++ library which provides almost fully automated functionalities encompassing grid generation, problem specification and simulation execution. However, attempting to rely on a fixed repertoire of black-box function units to assemble a CFD simulation program for jet engine noise prediction is unlikely to be a successful strategy. CAA is an art which combines science and engineering. While its essential theoretical foundation boils down to the Navier–Stokes equations, in practice, a functional CAA application depends on many more special numerical components. Just to name a few specific examples, [74] uses a randomized inflow vortex ring forcing method to promote the transition from laminar flow to turbulent flow. References [1, 40, 48, 74] attach an outflow sponge zone to the downstream end of the computational space to dampen the oscillations leaving the physical domain according to an artificial cubic fall-off profile. References [19] places an inflow sponge zone inside the simulated nozzle to suppress the spurious oscillations caused by the inflow boundary condition. None of these special-purpose treatments has a well-defined counterpart in the physical setting of a jet engine, but their inclusion in jet simulations is none the less crucial to ensuring that the numerical results are meaningful approximation to the real-world physics of jet engine noise and preventing the numerical artifacts inherent to the finite-precision floating-point arithmetic from leading to infinite values or NaNs. Due to their ad hoc nature, few, if there is any, of the packaged software libraries or applications would include them as ready-to-use components. Therefore, a programming tool dedicated to LES of jet engine noise must offer the possibility to manually specify the details of any involved numerical methods.

At the first glance, the fact that the programmer's need for the capability to specify arbitrary numerical methods may seem to reduce the researchers' and engineers' choices of programming tools back to traditional general-purpose programming tools such as Fortran, C and C++. This is not necessarily the case. In particular, we recognize one aspect of programming in which researchers and engineers can benefit from new programming tools. Recall that the governing equation of three-dimensional LES in Equation (1.1) applies uniformly to every individual point in the computational space. In numerical programming, this is referred to as elemental computation, which involves matching up the corresponding elements of multiple multidimensional arrays of identical shapes and carrying out a uniform sequence of scalar arithmetic operations on each individual combination of matched-up array elements. Besides elemental computation, Equation (1.1) also exhibits a second notable pattern of computation, which is represented by the three spatial partial differential operators $\frac{\partial}{\partial\xi}$, $\frac{\partial}{\partial\eta}$ and $\frac{\partial}{\partial\zeta}$. Mathematically, these three spatial partial differential operators are fundamentally identical in definition. As we have explained in section 1.2, they are also to be approximated in empirical jet engine noise prediction applications using the same compact spatial partial differentiation scheme. The only differences among them are their spatial orientations. Put in terms of the data structures on which they are to be implemented, the operations which comprise their definitions are applied along different array dimensions but are otherwise identical. If we construe elemental computation as a mechanism of repetition, then the latter pattern of computation can be regarded as a generalized form of elemental computation where slices of arbitrary dimensions, instead of merely scalar elements, of multiple multidimensional arrays are matched into combinations, and a uniform sequence of possibly nonscalar arithmetic operations are executed on each such combination. Viewed from an alternative perspective, it can also be interpreted as a procedure which maps an lower-dimensional numerical algorithm into a higher-dimensional computational space and augments the semantics of the algorithm with uniform repetition according to a certain dimension map.

Elemental computation, in its basic form, has long received support from programming languages and software libraries. As core syntactical features, the array section notation and elemental procedures have been available in Fortran since the Fortran 90 and 95 revisions, respectively. C and C++ have also recently received similar enhancements via external language extensions including OpenMP 4.0 [56] and Intel Cilk Plus [61]. By encapsulating arrays in specialized data types, programming languages which allow operator overloading such as C++ and Python can emulate the support of the array section notation for array expressions through library-only solutions such as the `std::valarray` class template in the C++ Standard Library and NumPy [21]. In particular, taking advantage of template metaprogramming, especially the technique of expression templates [79], `std::valarray` enables the emulated array expressions to avoid representing intermediate arithmetic results using temporary arrays, a performance optimization critical to the practical usability of the class template. Beyond the compile-time features of existing programming languages, Intel Array Building Blocks (ArBB) [54] and `ispc` [58] provide the programming capabilities for elemental computation by means of domain-specific languages (DSLs) and place emphasis on automatic vectorization and thread-level parallelization during code generation.

Compared to the basic form, the generalized form of elemental computation enjoys much more sporadic support from existing programming methodologies and tools. The ALPHA programming language [83, 84] supports a very generic form of elemental computation where array slices need not be rectangular-shaped or rectilinearly aligned and can be extracted from polyhedron-shaped multidimensional arrays via arbitrary affine maps. The axis control notation [29] of Single Assignment C (SAC) [28] follows a more canonical approach where array slices must be rectangular-shaped and rectilinearly aligned. Semanticswise, both ALPHA and SAC are sufficient for specifying the computational operations involved in LES of jet engine noise. Due to the fact that they belong to the category of functional programming languages, communication primitives are not included in their underlying programming models. While they are be construed as members of the family of implicitly distributed programming languages represented by High Performance Fortran (HPF) [46] and ZPL [43], and their implementations are responsible for partitioning the arrays appearing in programs and distributing them to multiple processors, without explicit communication operations, it is challenging, if that is possible at all, to express the tridiagonal linear system solvers needed by LES-based jet engine noise prediction applications. This is because efficient solvers such as those in [15, 66, 67] are usually developed based on the assumption that the programming model offers a single-program multiple-data (SPMD) programming style with an explicit notion of processors. They also actively exploit the assumption in the derivation of their respective numerical methods.

Based on the above arguments, we can summarize the desired features of any new programming tool dedicated to supporting composition of three-dimensional LES-based jet engine noise prediction applications as follows:

- It should provide a mechanism to express the computation pattern where lower-dimensional algorithms are applied in higher-dimensional contexts as previously described.
- It should support an SPMD programming style and encompass computation as well as communication in its expressive power.
- It should allow incremental adoption by researchers and engineers.

The last feature listed above is necessary because in many cases, the researchers and engineers already have readily functional applications at hand. It is impractical to require them to rewrite their applications from scratch just because they want to take advantage of a new programming tool.

2 AN EFFICIENT TRIDIAGONAL LINEAR SYSTEM SOLVER BASED ON THE TRUNCATED SPIKE ALGORITHM

2.1 General SPIKE algorithm applied to tridiagonal linear systems

The SPIKE algorithm [20, 59] is a parallel hybrid solver for narrow-banded linear systems. It is based on a philosophy similar to that of the Schur complement method described in section 1.3.3 in the sense that it also attempts to reduce the coupling between neighboring grid partitions to that between some designated interface grid points. Compared to the Schur complement method, it is different in that it uses both the first and the last grid points of a grid line within each grid partition as the interface grid points. It also avoids global LU factorization and employs an alternative method to eliminate the noninterface grid points from the coupling relations.

We specialize the general SPIKE algorithm for the purpose of solving the tridiagonal linear systems arising from LES of jet engine noise [69]. To illustrate the algorithm, we use the example of solving the linear system arising from computing spatial partial differentation on a grid line of 36 grid points evenly partitioned among six processors. Generalization is straightforward.

Let the tridiagonal linear system be

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f} \tag{2.1}$$

where the coefficient matrix $\boldsymbol{A}$ is depicted in Figure 2.1. The annotations "(P_1)" through "(P_6)" are symbolic names of the grid partitions. They can also be interpreted as the processors which own each of the grid partitions. Each small square in Figure 2.1 represents a nonzero element of $\boldsymbol{A}$ and is colored with a shade of gray whose visual darkness, as a fraction of pure black, is approximately porportional to the magnitude

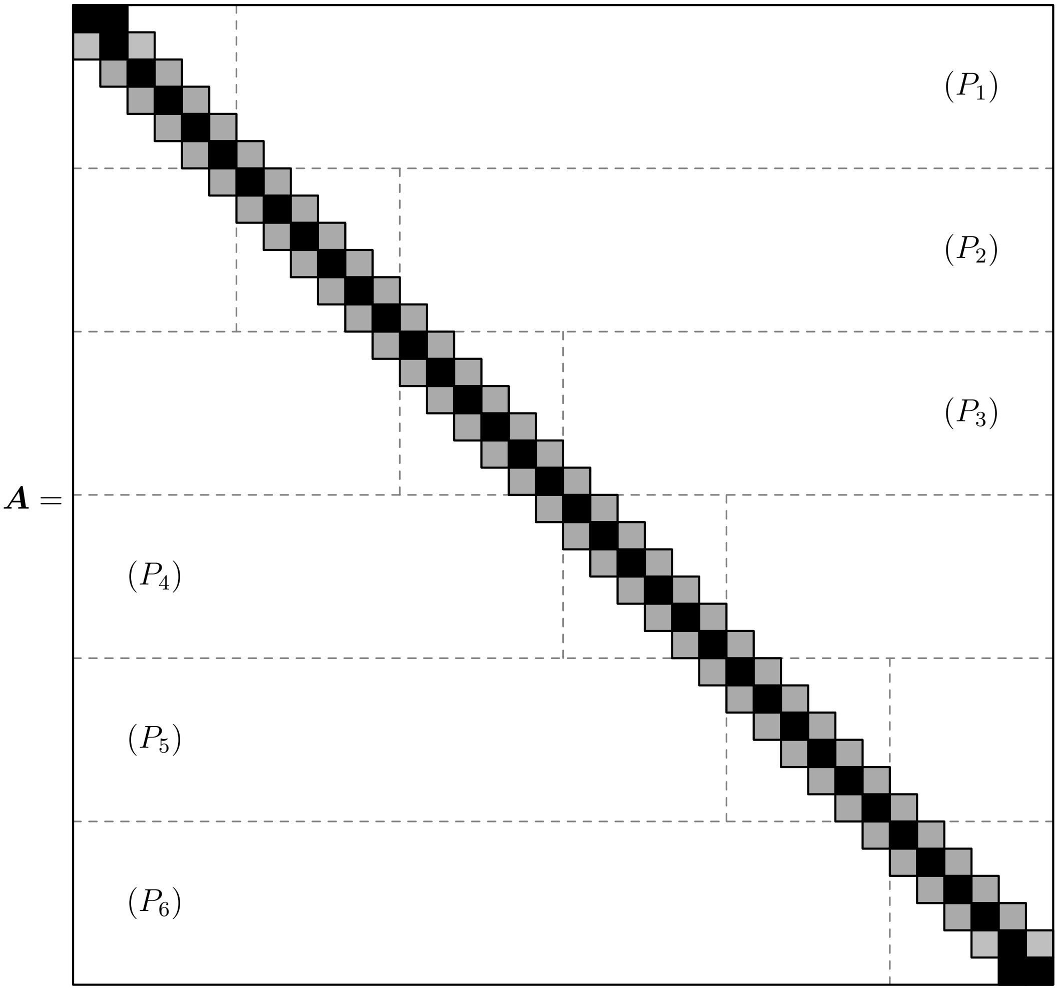

$\boldsymbol{A} =$
(P_1)
(P_2)
(P_3)
(P_4)
(P_5)
(P_6)

Figure 2.1. Coefficient matrix of 36-point spatial partial differentiation

of that element. Elements of magnitudes greater than one are drawn as fully black squares. We will adopt the same coloring scheme in other matrix plots to highlight the variation in the magnitudes of the matrix elements.

To solve the system $\boldsymbol{Ax} = \boldsymbol{f}$, the SPIKE algorithm first extracts the 6×6 diagonal blocks of $\boldsymbol{A}$ to form a block diagonal matrix $\boldsymbol{D}$ depicted in Figure 2.2. The removed elements in $\boldsymbol{D}$ compared to $\boldsymbol{A}$ are marked by dotted circles in Figure 2.2. The algorithm then computes a factorization

$$\boldsymbol{A} = \boldsymbol{DS}. \tag{2.2}$$

The factor $\boldsymbol{S} = \boldsymbol{D}^{-1}\boldsymbol{A}$ assumes the form shown in Figure 2.3. It has a unit main diagonal, from which single-column "spikes" extend both downwards and upwards. It is thus called the *spike matrix*, and Equation (2.2) is called the *spike factorization*. Using Equation (2.2), solving the tridiagonal linear system $\boldsymbol{Ax} = \boldsymbol{f}$ becomes equivalent to solving a new system $\boldsymbol{Sx} = \boldsymbol{g}$ where $\boldsymbol{g} = \boldsymbol{D}^{-1}\boldsymbol{f}$. Both $\boldsymbol{D}$ and $\boldsymbol{g}$ can be computed with perfect parallelism thanks to the block diagonal nature of $\boldsymbol{D}$.

A key insight of the SPIKE algorithm that the coupling between neighboring partitions of $\boldsymbol{S}$ is entirely captured by the first and the last elements of the partitions, i.e., the elements immediately above and below the partition boundaries. Figure 2.4 highlights these elements in red. These elements, along with the corresponding elements in $\boldsymbol{x}$ and $\boldsymbol{g}$, can be extracted from $\boldsymbol{Sx} = \boldsymbol{g}$ independently of the other elements to form a smaller linear system

$$\hat{\boldsymbol{S}}\hat{\boldsymbol{x}} = \hat{\boldsymbol{g}} \tag{2.3}$$

where the coefficient matrix $\hat{\boldsymbol{S}}$ is pentadiagonal. Figure 2.5 depicts the coefficient matrix $\hat{\boldsymbol{S}}$. We refer to Equation (2.3) as the *reduced system*. Once this reduced system is solved, the first and the last elements of $\boldsymbol{x}$ with each partition are known and can be backsubstituted into $\boldsymbol{Sx} = \boldsymbol{g}$ to retrieve the complete solution vector with perfect parallelism.

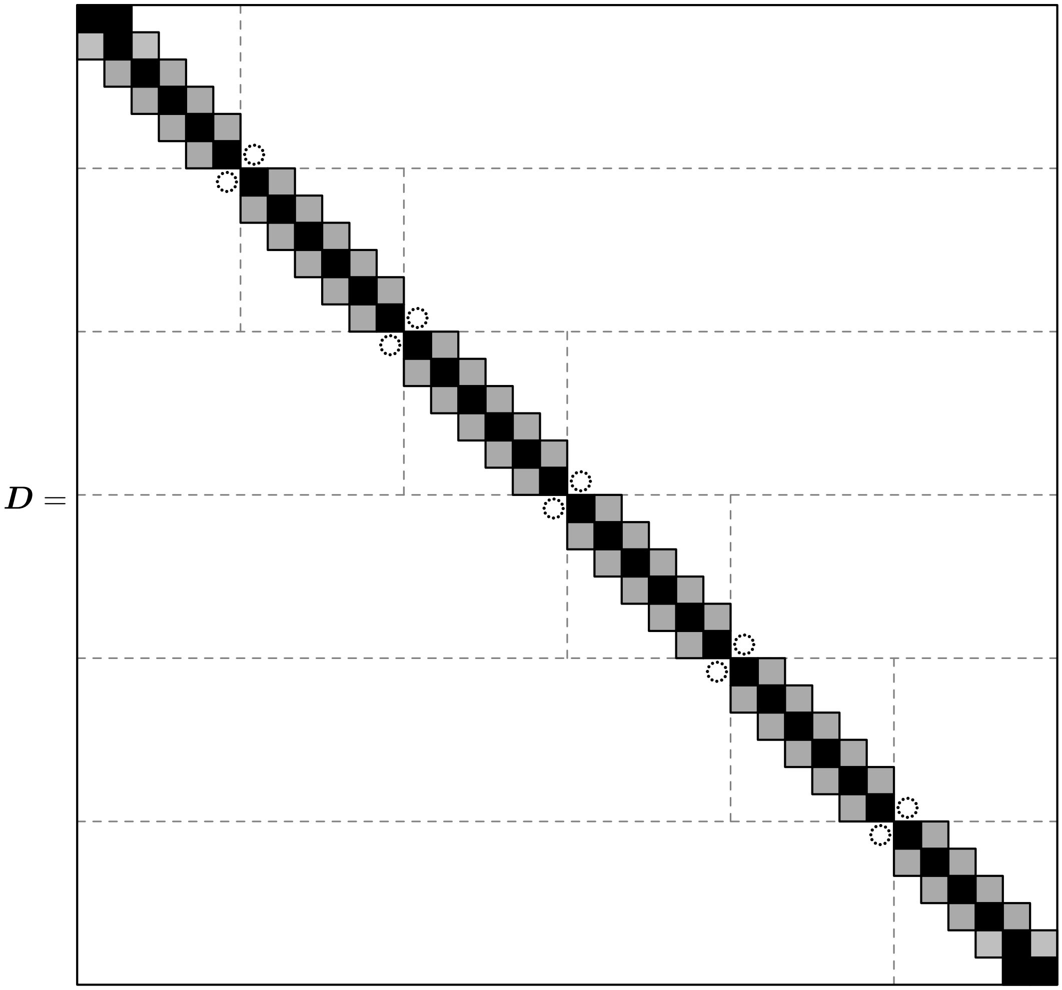


Figure 2.2. Diagonal blocks of the coefficient matrix $\boldsymbol{A}$ in Figure 2.1

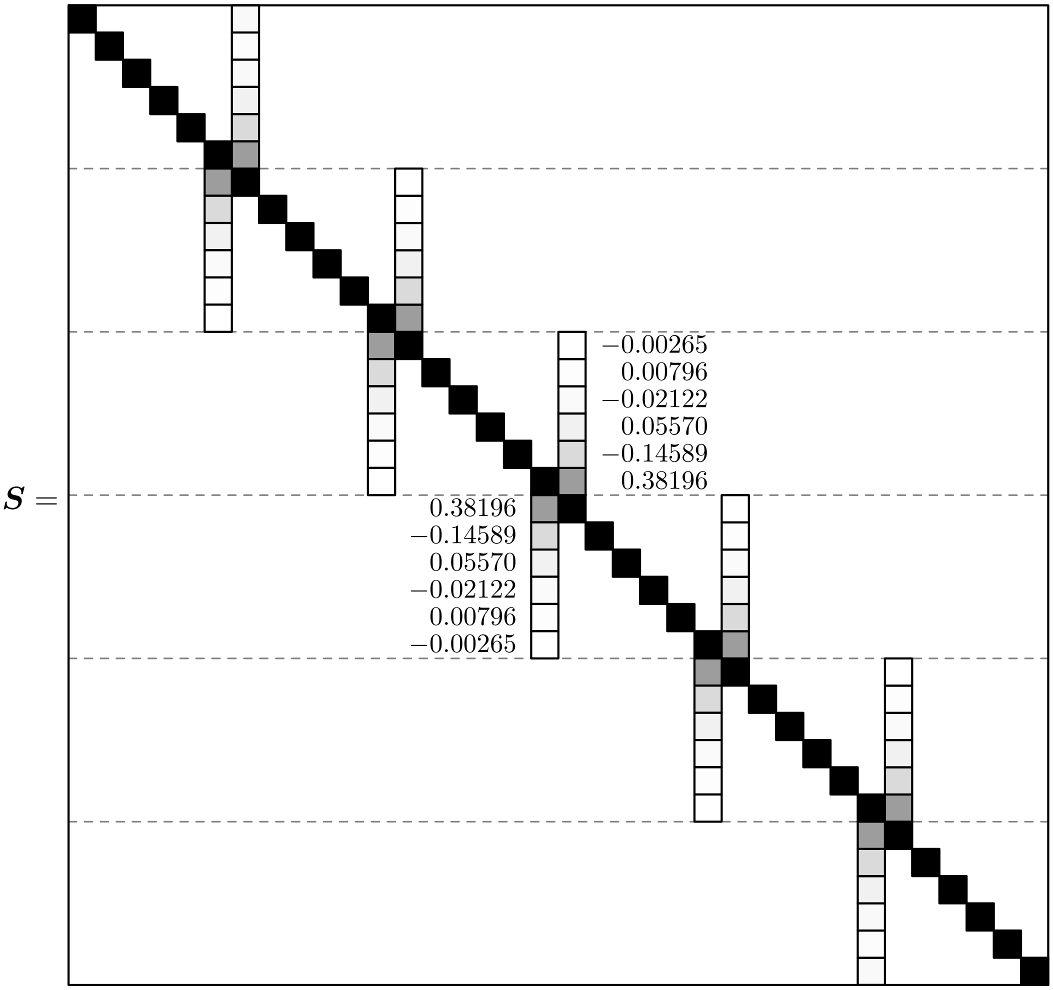

$S =$
−0.00265
0.00796
−0.02122
0.05570
−0.14589
0.38196
0.38196
−0.14589
0.05570
−0.02122
0.00796
−0.00265

Figure 2.3. Spike matrix $\boldsymbol{S}$ in the spike factorization of the matrix $\boldsymbol{A}$ in Figure 2.1

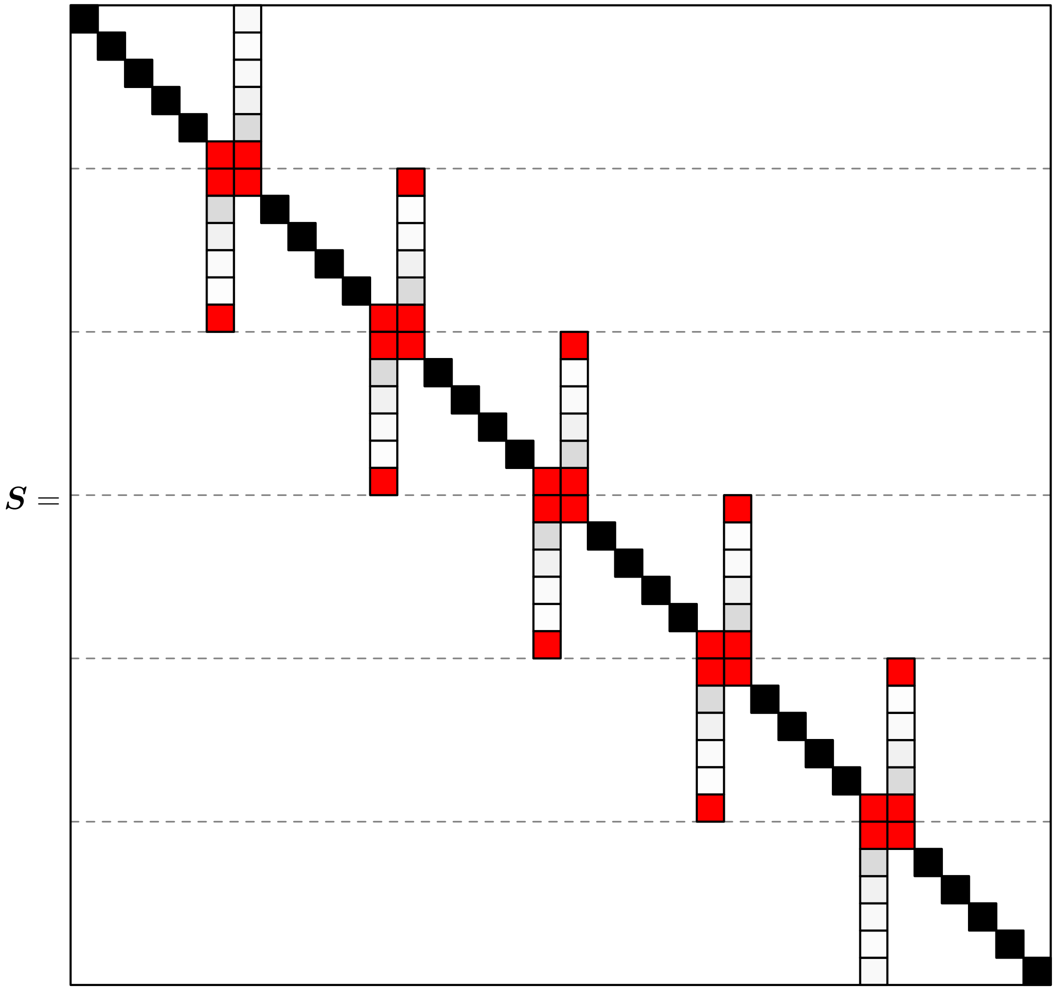

$\boldsymbol{S} =$

Figure 2.4. Elements of the spike matrix $\boldsymbol{S}$ in Figure 2.3 representing coupling between partitions

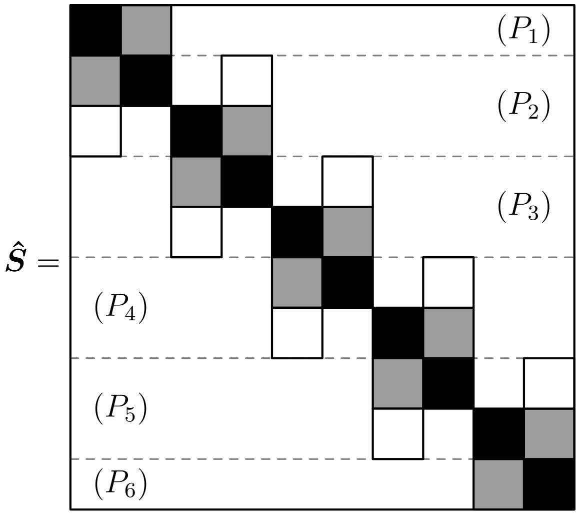

$\hat{\boldsymbol{S}} =$
(P_1)
(P_2)
(P_3)
(P_4)
(P_5)
(P_6)

Figure 2.5. Coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system in Equation (2.3)

Algorithm 2.1 provides a concise summary of the general SPIKE algorithm. It is worth noting that the general SPIKE algorithm leaves the precise method for solving the reduced system in Equation (2.3) unspecified. As a consequence, the choice of a particular method will lead to a specific variant of the SPIKE algorithm. Therefore, it is referred to as a *polyalgorithm* in [59, 60]. References [59, 60] describe three variants of the general SPIKE algorithm, namely the recursive, truncated and on-the-fly SPIKE algorithms. The recursive SPIKE algorithm is designed for narrow-banded linear systems which are dense within the band. It solves the reduced system via recursive halving by repeatedly merging pairs of neighboring partitions and in this sense bears resemblance to the parallel cyclic reduction used in the Schur complement method described in section 1.3.3. On the contrary, the on-the-fly SPIKE algorithm is designed for narrow-banded linear systems which are sparse within the band. In order to preserve the sparsity, it relies on a direct sparse linear system solver such as PARDISO [64], MUMPS [4] and SuperLU [18]. It also avoids explicitly forming the spike matrix and the reduced system and uses an iterative method to solve the reduced system. However, neither of these two variants of the general SPIKE algorithm is likely to lead to efficient solvers for tridiagonal linear systems arising from LES of jet engine noise. The recursive SPIKE algorithm takes an $\mathcal{O}((\log p)^2)$ asymptotic running time to solve the reduced system for a single right-hand side vector where p is the number of processors. This immediately renders it inferior to the Schur complement method since the asymptotic running time of parallel cyclic reduction is only $\mathcal{O}(\log p)$. The on-the-fly SPIKE algorithm is simply unnatural to be applied to tridiagonal linear systems to due latter's simple but dense-within-the-band structure. This effectively leaves the truncated SPIKE algorithm as the only option among the three variants of the general SPIKE algorithm to explore for deriving an efficient solver for tridiagonal linear systems. As it turns out, the truncated SPIKE algorithm does lead to a tridiagonal linear system solver which offers the desired features of an ideal solver mentioned in section 1.3.4. This solver is to be detailed in Section 2.2.

Algorithm 2.1. General SPIKE algorithm

```
procedure SPIKE-FACTORIZE($\boldsymbol{A}$)
    Extract the diagonal blocks from $\boldsymbol{A}$ to form $\boldsymbol{D}$
    Compute the spike matrix $\boldsymbol{S} = \boldsymbol{D}^{-1}\boldsymbol{A}$
    Form the coefficient matrix $\hat{\boldsymbol{S}}$ from $\boldsymbol{S}$ of the reduced system
    Preprocess $\hat{\boldsymbol{S}}$ for solution of the reduced system
end procedure

procedure SPIKE-SOLVE($\boldsymbol{D}, \boldsymbol{S}, \hat{\boldsymbol{S}}, \boldsymbol{f}$)
    Compute $\boldsymbol{g} = \boldsymbol{D}^{-1}\boldsymbol{f}$
    Form and solve the reduced system $\hat{\boldsymbol{S}}\hat{\boldsymbol{x}} = \hat{\boldsymbol{g}}$
    Backsubstitute $\hat{\boldsymbol{x}}$ into $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{g}$ to retrieve $\boldsymbol{x}$
end procedure
```

2.2 Efficient solution of tridiagonal linear systems using the truncated SPIKE algorithm

2.2.1 Basic truncated SPIKE algorithm

Recall that the tridiagonal linear systems arising from LES of jet engine noise come from the compact spatial partial differentiation and spatial filtering schemes. Equations (1.4), (1.5), (1.6) and (1.7) are the formulations of the sixth-order formulations of these schemes. An important feature of these formulations is that their left-hand sides all lead to diagonally dominant rows in the resulting coefficient matrices with the sole exception of the formulae for the first and the last points of a grid line in the case of spatial partial differentiation. Not limited to just the sixth-order formulations, this feature is also shared by formulations of other orders. Such diagonal dominance has a consequential implication on the magnitudes of the elements on the spikes in the spike matrix $\boldsymbol{S}$ in Figure 2.3. Observe that in Figure 2.3, as the elements lie further away from the main diagonal, the shade of gray of the small squares which represent them quickly diminishes into pure white, which indicates a steady decay in their magnitudes. As revealed by the element value annotations in the same figure, the decay in the element magnitude is in fact exponential. More rigorously speaking, as proved in [51], the rate of decay is lower-bounded by the degree of diagonal dominance of the coefficient matrix $\boldsymbol{A}$ in Figure 2.1, which is defined as

$$d = \min_i \left\{ \frac{|a_{ii}|}{\sum_{j \neq i} |a_{ij}|} \right\}. \tag{2.4}$$

In practice, the actual rate of decay is oftentimes a lot higher than what is predicted by Equation (2.4). For example, for the two spikes at the center of Figure 2.3, the magnitudes of the two elements at the spike tips are approximately -0.0026525. In the meantime, estimation based on Equation (2.4) using $d = 3/2$ evaluates to a very loose upper bound on their magnitudes of $1/d^6 \approx 0.087791$.

The exponentially small magnitudes of the elements at the spike tips in the spike matrix $\boldsymbol{S}$ makes it reasonable to ignore those elements. This effectively truncates the spikes by one element and leads to the *truncated SPIKE algorithm*. Translating the effect of such truncation from $\boldsymbol{S}$ to the coefficient matrix $\hat{\boldsymbol{S}}$ of the reduced system, the elements which lie outside of the 2×2 diagonal blocks are removed, which simplifies the reduced system from pentadiagonal to block diagonal with 2×2 diagonal blocks. The resulting coefficient matrix $\tilde{\boldsymbol{S}}$ is depicted in Figure 2.6. To solve this modified reduced system, only a constant amount of data needs to be exchanged between neighboring partitions, and the following computation can be performed with perfect parallelism. This enables the truncated SPIKE algorithm to achieve the same performance characteristics as the multiblock method described in section 1.3.2.

2.2.2 Truncated SPIKE algorithm enhanced with block Jacobi iteration

Truncation of the spikes in the spike matrix $\boldsymbol{S}$ and the ensuing effect on the coefficient matrix $\hat{\boldsymbol{S}}$ of the reduced system necessary cause the reduced system to be solved only approximately. The degree of approximation depends on the actual magnitudes of the elements at the truncated spike tips. In the case of the tridiagonal linear systems arising from LES of jet engine noise, it is fully determined by the orders of the compact spatial partial differential and spatial filtering schemes, the parameter α_f of the spatial filtering scheme and the partition size. As we have explained in Section 1.1, LES, or CAA in a broader scope, imposes very stringent requirements on the numerical accuracy of the underlying numerical methods. Therefore, the numerical inaccuracy introduced into the linear systems by the truncation of the spike tips must be remedied.

In [59], it is suggested that, when high numerical accuracy is desired, the truncated SPIKE algorithm can be wrapped inside an outer iterative scheme. In this case, the truncated SPIKE algorithm functions as a preconditioner of the outer iterative scheme. Popular choices of the outer iterative scheme include GMRES [62], QMR [23] and

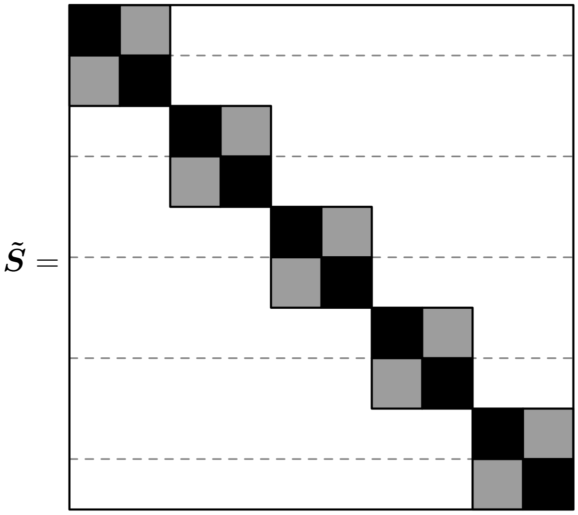

$\tilde{\boldsymbol{S}} =$

Figure 2.6. Coefficient matrix $\tilde{\boldsymbol{S}}$ of the SPIKE reduced system after truncation

BiCGSTAB [82]. However, since they all belong to the family of Krylov subspace methods, they share the vector dot product as a common fundamental component in their definitions and inevitably rely on the globally synchronizing all-to-all reduction communication primitive. If the truncated SPIKE algorithm is wrapped inside any of these methods, although it is reasonable to assume that the number of iterations required for convergence is relatively small, the all-to-all reductions will nevertheless degrade the asymptotic running time of the truncated SPIKE algorithm to the same level as the Schur complement method with embedded parallel cyclic reduction described in section 1.3.3. In order to avoid the all-to-all reductions, we must use an outer iterative scheme which does not rely on vector dot products. For our purposes, we choose the much less sophisticated block Jacobi iteration as the outer iterative scheme, which requires only local-scoped, nonsynchronizing communication between neighboring partitions. We perform 2×2 block Jacobi iteration on the reduced system.

An important element of any iterative scheme is determining when to terminate the iteration process. Usually, the relative residual corresponding to the iterate vector is monitored during every iteration. Once it drops below a predefined tolerance level, the iteration process is terminated, and the iterate vector of the last executed iteration is declared as converged. Unfortunately, the relative residual is defined in terms of the norm of the iterate vector, which is in turn defined in terms of the vector dot product. Hence, attempting to keep track of the relative residual at run-time will end up reintroducing the all-to-all reductions eliminated by adoption of the block Jacobi iteration. Instead, we resort to an alternative method based on the matrix norm to estimate *a priori* the minimum number of iterations after which the relative residual is guaranteed to drop below the predefined tolerance level. The feasibility of this method relies on two crucial properties of the tridiagonal linear systems arising from LES of jet engine noise. First, these linear systems are diagonally dominant except for the first and the last rows in the case of the compact spatial partial differentiation schemes. This ensures that the block Jacobi iteration is free from numerical instability and requires only a few iterations to converge. Second, these linear systems have a very simple and regular structure. This provides us with a convenient way to compute the matrix norm of interest.

We estimate the minimum of block Jacobi iterations needed to guarantee convergence as follows. Denote the iterate vector of the reduced system after n iterations by $\hat{\boldsymbol{x}}^{(n)}$ and the corresponding residual vector by $\hat{\boldsymbol{r}}^{(n)} = \hat{\boldsymbol{g}} - \hat{\boldsymbol{S}}\hat{\boldsymbol{x}}^{(n)}$. Each step of the block Jacobi iteration updates the iterate and residual vectors via the recurrence relation

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \tilde{\boldsymbol{S}}^{-1}\hat{\boldsymbol{r}}^{(n)}, \tag{2.5a}$$

$$\hat{\boldsymbol{r}}^{(n+1)} = \hat{\boldsymbol{g}} - \hat{\boldsymbol{S}}\hat{\boldsymbol{x}}^{(n+1)}. \tag{2.5b}$$

Eliminating $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{x}}^{(n+1)}$ from Equation (2.5b) gives

$$\boldsymbol{r}^{(n+1)} = (\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1})\boldsymbol{r}^{(n)}, \tag{2.6}$$

which implies

$$\begin{aligned} \|\hat{\boldsymbol{r}}^{(n+1)}\| &\leq \|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\| \cdot \|\hat{\boldsymbol{r}}^{(n)}\| \\ &\leq \|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|^2 \cdot \|\hat{\boldsymbol{r}}^{(n-1)}\| \\ &\vdots \\ &\leq \|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|^{n+1} \cdot \|\hat{\boldsymbol{r}}^{(0)}\|. \end{aligned} \tag{2.7}$$

Hence, in general, the residual vector after n iterations, $\hat{\boldsymbol{r}}^{(n)}$, satisfies

$$\frac{\|\hat{\boldsymbol{r}}^{(n)}\|}{\|\hat{\boldsymbol{r}}^{(0)}\|} \leq \|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|^{n}. \tag{2.8}$$

If we let the initial guess $\hat{\boldsymbol{x}}^{(0)}$ be a zero vector, then the initial residual vector $\hat{\boldsymbol{r}}^{(0)}$ is equal to $\hat{\boldsymbol{g}}$, and thus the initial relative residual is simply one. As long as we can determine $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$, given any tolerance level ϵ, we are guaranteed that the relative residual will not exceed ϵ after

$$\tau = \left\lceil \frac{\log \epsilon}{\log \|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|} \right\rceil \tag{2.9}$$

iterations. In an actual implementation of the truncated SPIKE algorithm for the purpose of LES of jet engine noise using the IEEE-754 double-precision floating-point arithmetic, we use $\epsilon = 2^{-52} \approx 2.2204 \times 10^{-16}$, the machine epsilon, to ensure that the numerical accuracy of the block Jacobi iteration is comparable to that of a direct solver based on the LU factorization.

In order to compute $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$, we first shift the boundaries of the partitioning of $\hat{\boldsymbol{S}}$ upwards by one row, leading to the form shown in Figure 2.7, then transpose the partitioning so that $\hat{\boldsymbol{S}}$ becomes partitioned by columns, ending in the form shown in Figure 2.8. The corresponding $\tilde{\boldsymbol{S}}$ also receives the same data redistribution treatment. This process ensures that none of the 2×2 diagonal blocks of $\hat{\boldsymbol{S}}$ and $\tilde{\boldsymbol{S}}$ straddles two neighboring partitions. It also allows the matrix $\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}$, depicted in Figure 2.9, to be formed conveniently. Observe that as illustrated in Figure 2.9, $\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}$ is comprised of 1×2 blocks each occupying a distinct row. Hence, for commonly used matrix norms including $\|\cdot\|_1$, $\|\cdot\|_2$ and $\|\cdot\|_\infty$, each processor can compact the one or two 1×2 blocks in its possession into a 1×2 or 2×2 small matrix, compute the norm of that matrix and perform an all-to-all reduction with all other processors to determine $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$. Once $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$ is known, τ can be calculated according to Equation (2.9) and saved for later use.

The value of τ determines the actual performance of the truncated SPIKE algorithm enhanced with block Jacobi iteration in practice. Table 2.1 lists the computed values of τ for scenarios of the sixth-order compact spatial partial differentiation and spatial filtering schemes where a grid line is divided into partitions of 8, 16, 32 and 64 grid points. The parameter α_f of the spatial filtering scheme is set to 0.47. The data are computed from cases where the grid line consists of 128, 256, 512 and 1,024 grid points. All these cases lead to the same value of τ for each partition size. Hence, we show only one value in the table for all the cases. As is evident in Table 2.1, the number of block Jacobi iterations needed to guarantee convergence for scenarios where each partition consists of 16 or more grid points are so small that iterative methods

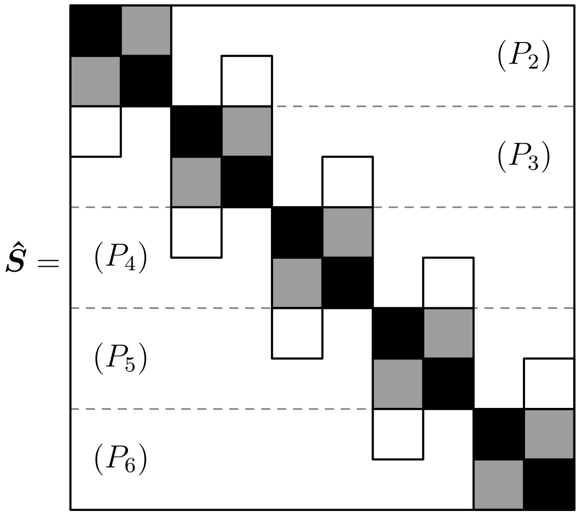

$\hat{\boldsymbol{S}} =$
(P_2)
(P_3)
(P_4)
(P_5)
(P_6)

Figure 2.7. Shifted partitioning of the coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system

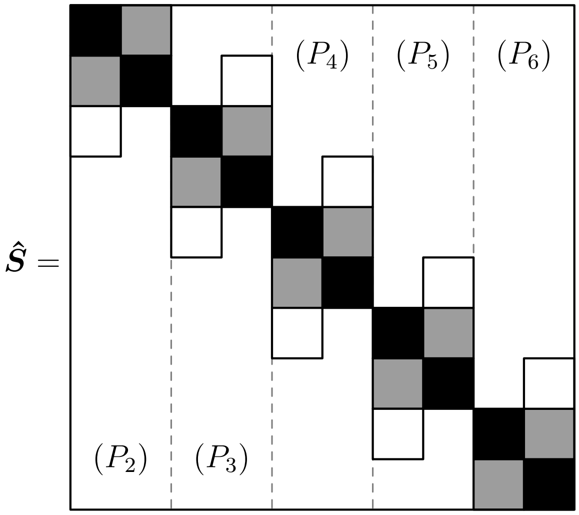

$\hat{\boldsymbol{S}} =$
(P_4)
(P_5)
(P_6)
(P_2)
(P_3)

Figure 2.8. Transposed partitioning of the coefficient matrix $\hat{\boldsymbol{S}}$ of the SPIKE reduced system

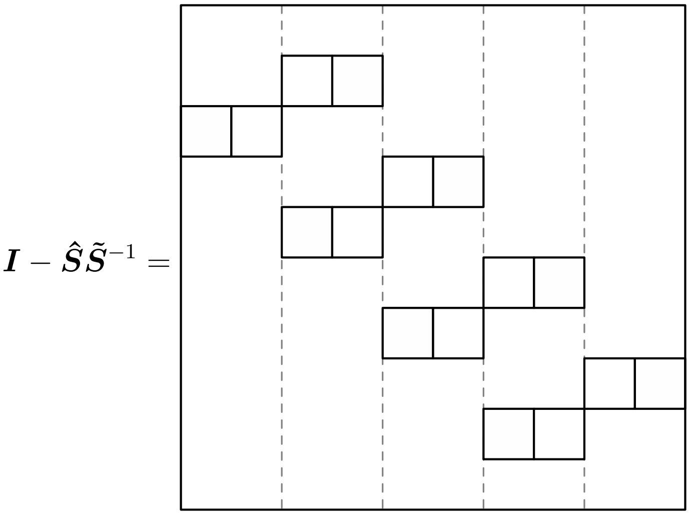

$\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1} =$

Figure 2.9. Iteration matrix $\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}$ of the residual vector of the SPIKE reduced system

Table 2.1.
Computed numbers of block Jacobi iterations needed by the truncated SPIKE algorithm

Points/partition	Differentiation	Filtering
8	6	15
16	2	6
32	1	3
64	0	1

more sophisticated than the block Jacobi iteration such as the previously mentioned GMRES, QMR and BiCGSTAB are rendered completely unnecessary.

2.3 Theoretical scalability analysis of the truncated SPIKE algorithm

We analyze the theoretical scalability of the truncated SPIKE algorithm enhanced with block Jacobi iteration in the context of three-dimensional LES of jet engine noise following the approach of isoefficiency analysis proposed in [27]. We compare the analysis result of the truncated SPIKE algorithm with those of the transposition method and the Schur complement method with embedded parallel cyclic reduction described in sections 1.3.1 and 1.3.3.

In the isoefficiency analysis of a parallel algorithm, one attempts to determine N, the size of the computational problem, as a function of p, the number of processors, which maintains a constant parallel efficiency as the latter increases. The parallel efficiency of an algorithm is defined as

$$E(p) = \frac{T(1)}{p \cdot T(p)} \tag{2.10}$$

where $T(p)$ is the running time of the algorithm when it is executed on p processors. The product $p \cdot T(p)$ represents the aggregate running time of the algorithm across all p processors. In particular, when $p = 1$, $T(1)$ represents its sequential running time. A parallel algorithm is deemed scalable if N needs to grow only mildly with respect to p in order to maintain a constant $E(p)$.

For sake of clarity in our analysis of the three tridiagonal linear system solvers for three-dimensional LES, we make the following assumptions and simplications. First, we assume that an $N \times N \times N$ grid is used to represent the computational space, and p^3 processors arranged in a $p \times p \times p$ grid is used to perform the simulation. We further assume that the interconnection between the processors has a perfect three-dimensional Cartesian topology. This is a realistic approximation of the processor grid topologies which can be commonly achieved on today's petascale scientific computing platforms. Furthermore, although the problem size of three-dimensional LES and the number of processors are N^3 and p^3, respectively, we attempt to determine a relation between N and p instead. This alternative approach is mathematically equivalent but notationally simpler. Finally, due to the presence of platform-dependent constants such as the relative costs of the computation and communication operations, we consider the isoefficiency analysis in the asymptotic sense. Specifically, we consider only the asymptotic growth of N with respect to p such that $E(p^3) = \mathcal{O}(1)$.

Before proceeding to the more complex parts of the individual analyses of the three tridiagonal linear system solvers, we point out that they share the common asymptotic sequential running time $T(1) = \mathcal{O}(N^3)$ because they are all reduced to the Thomas algorithm when executed sequentially.

2.3.1 Analysis of the truncated SPIKE algorithm

In the truncated SPIKE algorithm, the total aggregate running time of obtaining the reduced systems and retrieving the complete solution vectors after the reduced systems are solved is $\mathcal{O}(N^3)$ because the computation is completely local to each processor, and linear-time numerical methods are used for these parts. The aggregate running time of solving the reduced systems is $\mathcal{O}(\tau \times p^3 \times (N/p)^2) = \mathcal{O}(\tau N^2 p)$ because there are τ block Jacobi iterations, and within each iteration, $(N/p)^2$ numbers are exchanged between neighboring grid partitions and used in constant-time computation. Therefore, the total aggregate running time of the truncated SPIKE algorithm is

$$T(p) = \mathcal{O}(N^3 + \tau N^2 p). \tag{2.11}$$

In order to establish a relation between N and p, we need to eliminate τ from Equation (2.11). Given the definition of τ in Equation (2.9), we have to determine an expression which defines $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$ in terms of N and p where $\|\cdot\|$ can be any of $\|\cdot\|_1$, $\|\cdot\|_2$ and $\|\cdot\|_\infty$. More concretely speaking, we want to find an asymptotic upper bound on $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$. For that purpose, we first rewrite $\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}$ as $(\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}})\tilde{\boldsymbol{S}}^{-1}$ and then estimate $\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\|$ via

$$\begin{aligned}\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\| &= \|(\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}})\tilde{\boldsymbol{S}}^{-1}\| \\ &\leq \|\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}\| \cdot \|\tilde{\boldsymbol{S}}^{-1}\|. \end{aligned} \tag{2.12}$$

The matrix $\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}$, depicted in Figure 2.10, consists of the elements which originate from the spike tips in the spike matrix $\boldsymbol{S}$ illustrated in Figure 2.3. As a result, the magnitude of each of the individual nonzero elements of $\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}$ is $\mathcal{O}(1/d^{N/p})$ where d is the degree of diagonal dominance defined in Equation (2.4). Observe that in Figure 2.10, each of the nonzero elements of $\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}$ occupies a distinct row and a distinct column. Therefore, it is possible to permute the matrix into a diagonal matrix, and thus $\|\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}\|$ is simply the absolute value of its largest-magnitude nonzero element, which is $\mathcal{O}(1/d^{N/p})$. For $\|\tilde{\boldsymbol{S}}^{-1}\|$, since $\tilde{\boldsymbol{S}}$ is a block diagonal matrix, $\tilde{\boldsymbol{S}}^{-1}$ is also block diagonal. Let a diagonal block of $\tilde{\boldsymbol{S}}$ be

$$\begin{bmatrix} 1 & v \\ w & 1 \end{bmatrix},$$

then both v and w are $\mathcal{O}(1/d)$ due to the exponential decay in element magnitudes along the spikes in the spike matrix $\boldsymbol{S}$, and the corresponding diagonal block in $\tilde{\boldsymbol{S}}^{-1}$ is

$$\begin{bmatrix} 1 & v \\ w & 1 \end{bmatrix}^{-1} = \frac{1}{1 - vw} \begin{bmatrix} 1 & -v \\ -w & 1 \end{bmatrix}. \tag{2.13}$$

Hence, assuming that Equation (2.13) represents the largest-norm diagonal block of $\tilde{\boldsymbol{S}}^{-1}$,

$$\begin{aligned}\|\tilde{\boldsymbol{S}}\| &= \left\| \frac{1}{1 - vw} \begin{bmatrix} 1 & -v \\ -w & 1 \end{bmatrix} \right\| \\ &= \mathcal{O}\left(\frac{1}{1 - (1/d)^2} \cdot (1 + 1/d) \right) \\ &= \mathcal{O}(1) \end{aligned} \tag{2.14}$$

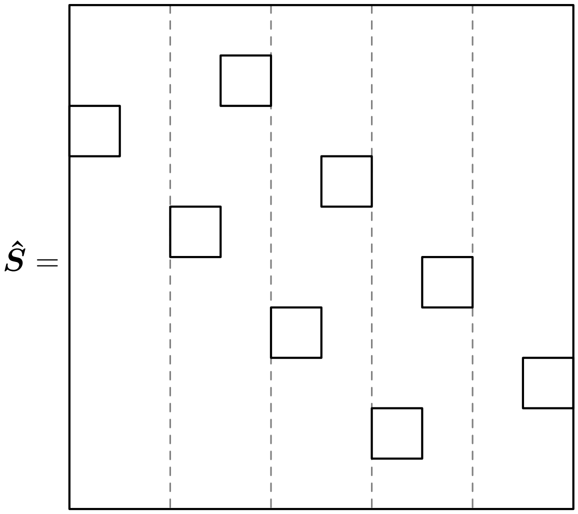

$\hat{\boldsymbol{S}} =$

Figure 2.10. Matrix $\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}$ representing the truncated spike tips removed from matrix $\tilde{\boldsymbol{S}}$

since d is a constant for any given tridiagonal linear system. Plugging

$$\|\tilde{\boldsymbol{S}} - \hat{\boldsymbol{S}}\| = \mathcal{O}(1/d^{N/p}), \tag{2.15a}$$

$$\|\tilde{\boldsymbol{S}}^{-1}\| = \mathcal{O}(1) \tag{2.15b}$$

into Inequality 2.12, we conclude that

$$\|\boldsymbol{I} - \hat{\boldsymbol{S}}\tilde{\boldsymbol{S}}^{-1}\| = \mathcal{O}(1/d^{N/p}), \tag{2.16}$$

and consequently,

$$\tau = \mathcal{O}(p/N), \tag{2.17}$$

$$T(p) = \mathcal{O}(N^3 + Np^2). \tag{2.18}$$

With these results, we arrive at

$$\begin{aligned} E(p^3) &= \mathcal{O}\left(\frac{N^3}{N^3 + Np^2}\right) \\ &= \mathcal{O}\left(\frac{1}{1 + (p/N)^2}\right), \end{aligned} \tag{2.19}$$

which becomes $\mathcal{O}(1)$ when $N = \mathcal{O}(p)$. This means that the truncated SPIKE algorithm, even when augmented with the block Jacobi iteration, attains an asymptotically constant parallel efficiency under weak scaling conditions. It is therefore as scalable as the multiblock method described in section 1.3.2 yet solves tridiagonal linear systems accurately.

2.3.2 Analysis of the transposition method

In the transposition method, tridiagonal linear systems are essentially solved by the Thomas algorithm, and transposition is used for only data redistribution which ensures that each right-hand side vector is not partitioned across multiple processors as required by the Thomas algorithm. The aggregate running time of the Thomas algorithm across all processors is obviously $\mathcal{O}(N^3)$. Now we need to account the aggregate running time of the transposition process.

Figure 2.11 illustrates the transposition process in the transposition method. Without loss of generality, we assume that the transposition is to be carried out over the ξ–ζ plane. Before the transposition takes place, each processor owns a separate grid partition of dimension $(N/p) \times (N/p) \times (N/p)$. During the transposition process, each processor first slices the grid partition in its possession into p slabs of dimensions $(N/p) \times (N/p) \times (N/p^2)$. Among these slabs, exactly one remains in the possession of its current owner, and each of the other slabs are exchanged with a distinct peer processor. When all pairwise slab exchanges between the processors have completed, each processor keeps one of its original slabs and receives $p - 1$ new slabs from the other processors. These p slabs are concatenated to form a larger slab of dimensions $N \times (N/p) \times (N/p^2)$ which contains all data belonging to N^2/p^3 right-hand side vectors to be supplied to the Thomas algorithm. After the tridiagonal linear systems are solved, the transposition process is performed in the reverse order to restore the original data distribution.

To calculate the communication cost of the transposition process, we consider a bisection of the $p \times p \times p$ processor grid into two halves each of dimensions $(p/2) \times p \times p$. The aggregate network bandwidth across these two halves of the processor grid is $\mathcal{O}(p^2)$ since they are connected by p^2 network links. Due to the nature of the transposition process being an aggregation of all-to-all communication operations among processors belonging to each individual grid line, $\mathcal{O}(N^3)$ data need to transferred across the interface between the two halves. Hence, the aggregate running time of the transposition process is $\mathcal{O}(p^3 \times N^3/p^2) = \mathcal{O}(N^3 p)$. As a result, the total aggregate running time of the transposition method is $\mathcal{O}(N^3 p)$. Correspondingly, its parallel efficiency is

$$\begin{aligned} E(p^3) &= \mathcal{O}\left(\frac{N^3}{N^3 p}\right) \\ &= \mathcal{O}(1/p), \end{aligned} \tag{2.20}$$

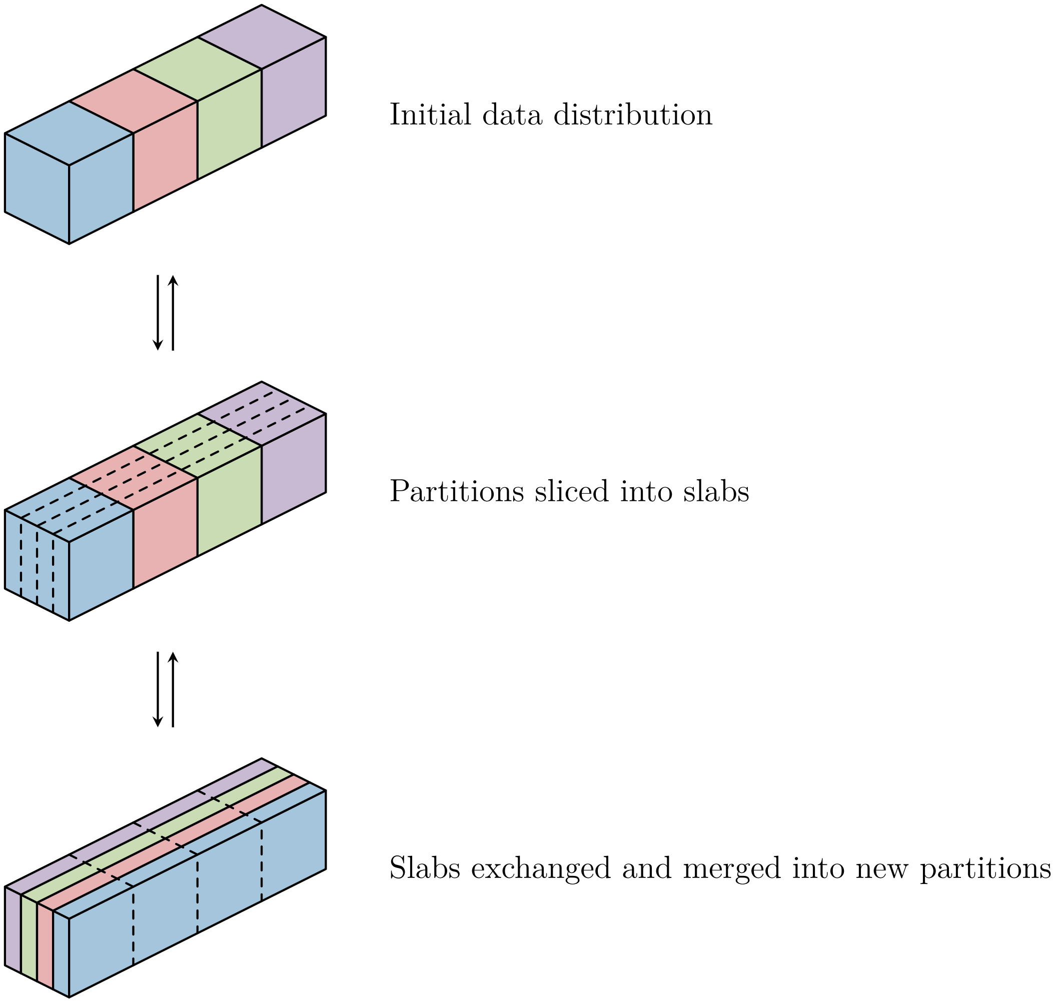

Initial data distribution
Partitions sliced into slabs
Slabs exchanged and merged into new partitions

Figure 2.11. Transposition process in the transposition method

which is always asymptotically smaller than $\mathcal{O}(1)$ and in fact diminishes to zero as p increases. This indicates that the transposition method can never maintain an asymptotically constant parallel efficiency no matter how large the computational problem becomes.

2.3.3 Analysis of the Schur complement method

In the Schur complement method, the coefficient matrix of the tridiagonal linear systems to be solved is permuted and partially LU-factorized to form a smaller tridiagonal Schur complement system of order p. Due to the form of the partial LU factorization, which is illustrated in Figure 1.6, the total aggregate running time across all processors of any computation not directly related to solving the Schur complement system is $\mathcal{O}(N^3)$. Since we assume that the Schur complement system is solved using parallel cyclic reduction, which has $\mathcal{O}(\log p)$ stages each taking constant time for a single right-hand side vector, the associated aggregate running time is $\mathcal{O}(p^3 \times (N^2/p^2) \times \log p) = \mathcal{O}(N^2 p \log p)$. Therefore, the total aggregate running time of the Schur complement method is $\mathcal{O}(N^3 + N^2 p \log p)$, and corresponding parallel efficiency is

$$\begin{aligned} E(p^3) &= \mathcal{O}\left(\frac{N^3}{N^3 + N^2 p \log p}\right) \\ &= \mathcal{O}\left(\frac{1}{1 + (p \log p)/N}\right), \end{aligned} \tag{2.21}$$

which becomes $\mathcal{O}(1)$ when $N = \mathcal{O}(p \log p)$. Compared to the truncated SPIKE algorithm enhanced with block Jacobi iteration, this is asymptotically larger by a logarithmic factor.

2.4 Empirical scalability verification

To verify our theoretical argument, we conduct a series of weak scaling experiments on Kraken, a Cray XT5 cluster hosted at the National Institute of Computational

Sciences (NICS) at the University of Tennessee. We implement the algorithm using MPI as the communication API. The experiments share a common grid partition size of $32 \times 32 \times 32$ but use different numbers of processor cores ranging from 2,744 to 42,875. Each processor cores runs a separate MPI rank. Experimental results are collected separately for each of the ξ-, η- and ζ-directions.

Table 2.2 lists the measured empirical running times of the truncated SPIKE algorithm. We note that the performance differences between the ξ-, η- and ζ-directions are mainly due to the fact that the `MPI_CART_CREATE` subroutine, which we use to establish the mapping from MPI processes to processor cores, happens to favor the ζ-direction while disadvantaging the ξ-direction. Although the figures in Table 2.2 show fluctuations as the number of processor cores changes, those anomalies can be attributed to nondeterministic factors in the experiments including the run-time allocation of compute nodes and the network traffic of other jobs executing concurrently on the cluster. Overall, the running times do not exhibit significant increasing trends as the number of processor cores increases, which is consistent with the theoretically predicted scalability results.

Table 2.2.
Empirical running times of the truncated SPIKE algorithm in weak scaling experiments on Kraken

Core configuration	ξ-direction (ms)	η-direction (ms)	ζ-direction (ms)
$14 \times 14 \times 14$	4.52	3.72	2.37
$15 \times 15 \times 15$	8.54	5.43	1.87
$18 \times 18 \times 18$	6.03	3.86	2.08
$21 \times 21 \times 21$	6.11	3.86	2.45
$30 \times 30 \times 30$	8.74	5.31	2.00
$35 \times 35 \times 35$	8.44	4.28	1.02

3 EFFICIENT IMPLEMENTATION OF FINITE DIFFERENCE-BASED THREE-DIMENSIONAL LARGE EDDY SIMULATION OF JET ENGINE NOISE

3.1 Software engineering considerations

During earlier stages of this study [66, 67], we inherited a codebase from [74] written largely in a mix of Fortran 77 and Fortran 90. The codebase was based on the transposition method described in section 1.3.1. With this codebase, we experimented with the truncated SPIKE algorithm described in detail in Chapter 2. We followed an incremental code development methodology, modifying only the parts of interest while leaving the majority of the code intact. This strategy proved feasible because we focused our attention on a comparison of the truncated SPIKE algorithm against the transposition method. Furthermore, the codebase dealt with only a simple one-dimensional computational space partitioning. However, as our research progresses towards the more realistic three-dimensional computational space partitioning, even though we have access to an alternative codebase from [40] based on the Schur complement method described in section 1.3.3, the complexity of the two codebases and the numerical methods of LES renders the incremental approach difficult to carry forward. Therefore, we decide to rewrite the entire LES-based jet engine noise prediction application from scratch with code portability, maintainability and reusability in mind.

To be specific, we use Fortran 2003 as the primary programming language and MPI as the communication API. We restrict the use of the programming language and software libraries to those which have mature support across the multiple computing platforms which we use for development and experiments. The emphasis on code portability enables us to develop and test our new codebase using small multicore servers and carry out large-scale experiments on petascale clusters. We extensively use broadly supported modern Fortran language features such as modules and derived types with the intention to decompose the code so that it can be maintained in small functional units. The result of these efforts is a highly modularized implementation of LES-based jet engine noise prediction. We expect many of its components to remain reusable even after the current research project concludes.

3.2 Managing the three-dimensional computational spacing partitioning

We partition the three-dimensional computational space using the most straightforward strategy. Given a computational space which is discretized into a computational grid of dimensions $N_\xi \times N_\eta \times N_\zeta$, we form a logical grid of processor cores of dimensions $p_\xi \times p_\eta \times p_\zeta$. We divide the computational grid into $p_\xi \times p_\eta \times p_\zeta$ contiguous grid partitions of dimensions $(N_\xi/p_\xi) \times (N_\eta/p_\zeta) \times (N_\zeta/p_\zeta)$ and assign each grid partition to the processor core located at the corresponding position in the logical grid of processor cores. Given the total number of processor cores $p = p_\xi p_\eta p_\zeta$, we always choose a combination (p_ξ, p_η, p_ζ) such that N_ξ/p_ξ, N_η/p_η and N_ζ/p_ζ are as close to one another as possible. In other words, we always try to make the shape of each grid partition as close to a cube as possible. This decreases the surface-to-volume ratio of the computational space partitioning. Due to the communication patterns of the truncated SPIKE algorithm and the stencil computation involved in the compact spatial partial differentiation and spatial filtering schemes, this surface-to-volume ratio determines the total communication volume of the jet engine noise prediction application. Decreasing the surface-to-volume ratio also reduces the communication cost.

We rely on the communication API and the run-time environment to establish the mapping from physical processor cores to their corresponding locations in the logical grid of processor cores. In the case of MPI, this is realized using the `MPI_CART_CREATE` subroutine. The `MPI_CART_CREATE` subroutine returns at each processor core an MPI communicator which represents the topology of the entire logical grid of processor cores and is then split into three separate communicators using the `MPI_CART_SUB` subroutine. Each of these three communicators represents a grid line of processor cores along one of the ξ-, η- and ζ-directions. They are the actual MPI communicators used for the communication in the compact spatial partial differentiation and spatial filtering schemes.

3.3 Communication optimizations

3.3.1 Overlapping computation and communication in stencil computation

Overlapping computation and communication to hide the communication latency by means of nonblocking communication primitives is a common optimization technique from which many scientific computing applications can benefit. Our LES-based jet engine noise prediction application is no exception. In LES, the primary source of opportunities of computation–communication overlapping is computation of the right-hand side vectors of the tridiagonal linear systems arising from the compact spatial partial differentiation and spatial filtering schemes.

Due to the manner in which the computational space is partitioned, computation of the right-hand side vectors of the tridiagonal linear systems for spatial partial differentiation and spatial filtering requires only communication between processor cores which are neighbors in the logical grid of processor cores. Depending on the operation being spatial partial differentiation or spatial filtering, for each right-hand side vector, a processor core needs to exchange the flow variable values at grid points belonging to the two or three boundary planes at each face of the grid partition in its possession with the corresponding neighboring processor core. Furthermore, since only the values of the right-hand side vectors at the same boundary locations depend on flow variable values from the neighboring processor cores, the other values can be computed when while the messages being exchanged are in-flight.

Algorithm 3.1 provides a precise listing of operations assuming that the grid lines of grid points in the flow field and of the involved processor cores are aligned along the vertical direction.

3.3.2 Reducing communication overhead in block Jacobi iteration

In modern MPI implementations, messages transmitted during point-to-point communication (as opposed to collective communication) in the standard mode are delivered using either an eager protocol or a rendezvous protocol. In the eager protocol, the sender delivers a message to the receiver without waiting for the latter to post a request to receive data. The delivered message is buffered internally by the MPI implementation and copied to the receive buffer supplied by the programmer after the matching receive operation has started. Meanwhile, in the rendezvous protocol, the sender not does deliver the message to the receiver until the latter has posted the request to receive data. Comparing the two protocols, the eager protocol avoids any handshake between the sender and the receiver but requires additional buffering, whereas the rendezvous protocol allows the MPI implementation to omit any internal buffering and deliver the data directly to the receive buffer supplied by the programmer. Due to their respective characteristics, the eager protocol is typically used for short messages, while the rendezvous protocol is usually used for long messages.

The trade-off between the overhead of internal buffering and that of handshake is necessitated by the fact that in standard-mode point-to-point communication, the sender does not make any assumption about when the receiver is ready to receive the message and thus must be prepared for all possible scenarios. However, in situations where the sender knows that the receiver is ready, it is desirable to have the functionality to deliver the message without delay as in the eager protocol and also avoid message buffering as in the rendezvous protocol. The ready-mode point-to-point communication of MPI provides exactly such a capability.

Algorithm 3.1. Stencil computation of right-hand side vectors in spatial partial differentiation and spatial filtering

```
Post nonblocking receive requests to neighboring processor cores immediately
    above and below
Initiate nonblocking send operations to both neighbors
Complete all local portions of stencil computation
repeat
    Wait for either receive operation to complete
    if the completed receive operation was from above then
        Fix up the values at grid points at the top boundary
    else                        ▷ The completed receive operation was from below
        Fix up the values at grid points at the bottom boundary
    end if
until both receive operations have completed
```

As detailed in Chapter 2, the truncated SPIKE algorithm uses block Jacobi iteration to solve the reduced system in Equation (2.3). During each block Jacobi iteration, each pair of neighboring processor cores exchanges a pair of messages. If we prepost the requests to receive data for all iterations before any iteration is executed, starting with the second iteration, we are assured that the receivers are always ready to receive data and can thus take advantage of ready-mode point-to-point communication to reduce the communication overhead. In practice, however, preposting all requests to receive data at once will require the receive buffers for all iterations to coexist. This inflates the memory consumption. It is also unnecessary because the send operations of every second iterations cannot overlap due to the data dependence in the block Jacobi iteration. Therefore, we allocate only two receive buffers for each destination neighboring processor core and use them in an alternating fashion. Algorithm 3.2 lists the exact algorithmic steps.

3.4 Computation optimizations

We use the standard structure-of-arrays (SOA) data layout the store the values of the five flow variables at all grid points. The data structure is implemented as collections of four-dimensional arrays of dimensions $(N_\xi/p_\xi) \times (N_\eta/p_\eta) \times (N_\zeta/p_\zeta) \times 5$. Under the column-major dimension order of Fortran arrays, the first three dimensions of each four-dimensional array correspond to the coordinates (ξ, η, ζ) of the computational space, while the last dimension serves to distinguish the different flow variables.

Since computation other than the compact spatial partial differentiation and spatial filtering schemes is entirely pointwise and typically accounts for a relatively small portion of the total computational cost, it allows little headroom for improvement. Therefore, we make only some minor efforts in its optimization. This includes collapsing each loop nest iterating over the entire coordinate space into a single flattened loop and annotating it as free of loop-carried dependence using compiler directives. Both measures help the compiler better vectorize those loops.

Algorithm 3.2. Block Jacobi iteration using the ready-mode point-to-point communication of MPI

```
Allocate four receive buffers $b_1^{(\mathrm{T})}$, $b_1^{(\mathrm{B})}$, $b_2^{(\mathrm{T})}$ and $b_2^{(\mathrm{B})}$
for $i \leftarrow 1, 2$ do
    if $i \leq \tau$ then
        Post the receive requests of iteration $i$ using $b_i^{(\mathrm{T})}$ and $b_i^{(\mathrm{B})}$ as the receive buffers
    end if
end for
for $i \leftarrow 1, 2, \ldots, \tau$ do
    if $i = 1$ then
        Initiate the nonblocking send operations of iteration $i$ to the neighboring processor cores using the standard mode
    else
        Initiate the nonblocking send operations of iteration $i$ to the neighboring processor cores using the ready mode
    end if
    Complete the receive operations of iteration $i$
    if $i \leq \tau - 2$ then
        $j \leftarrow (i - 1) \bmod 2 + 1$                    ▷ $j$ alternates between 1 and 2
        Post the receive requests of iteration $i + 2$ using $b_j^{(\mathrm{T})}$ and $b_j^{(\mathrm{B})}$ as the receive buffers
    end if
    Perform all computation of iteration $i$
end for
Complete the send operations still in-progress
```

We invest the majority of our computation optimization efforts in the compact spatial partial differentiation and spatial filtering schemes. Conceptually, both spatial partial differentiation and spatial filtering entail the same sequence of operations:

1. evaluation of right-hand side vectors via stencil computation;
2. solution of tridiagonal linear systems using the truncated SPIKE algorithm.

Computation local to each processor core and independent of any communication can be categorized into the following three main kind:

- local portions of stencil computation during evaluation of the right-hand side vectors;
- the embedded Thomas algorithm in the truncated SPIKE algorithm;
- retrieval of complete solution vectors in the truncated SPIKE algorithm.

The first kind is part of the computation of the right-hand side vectors in Algorithm 3.1, whereas the other two kinds comprise the first and the last steps of the SPIKE-SOLVE procedure in Algorithm 2.1. We manually optimize the first two kinds while leaving the simpler third kind to the compiler.

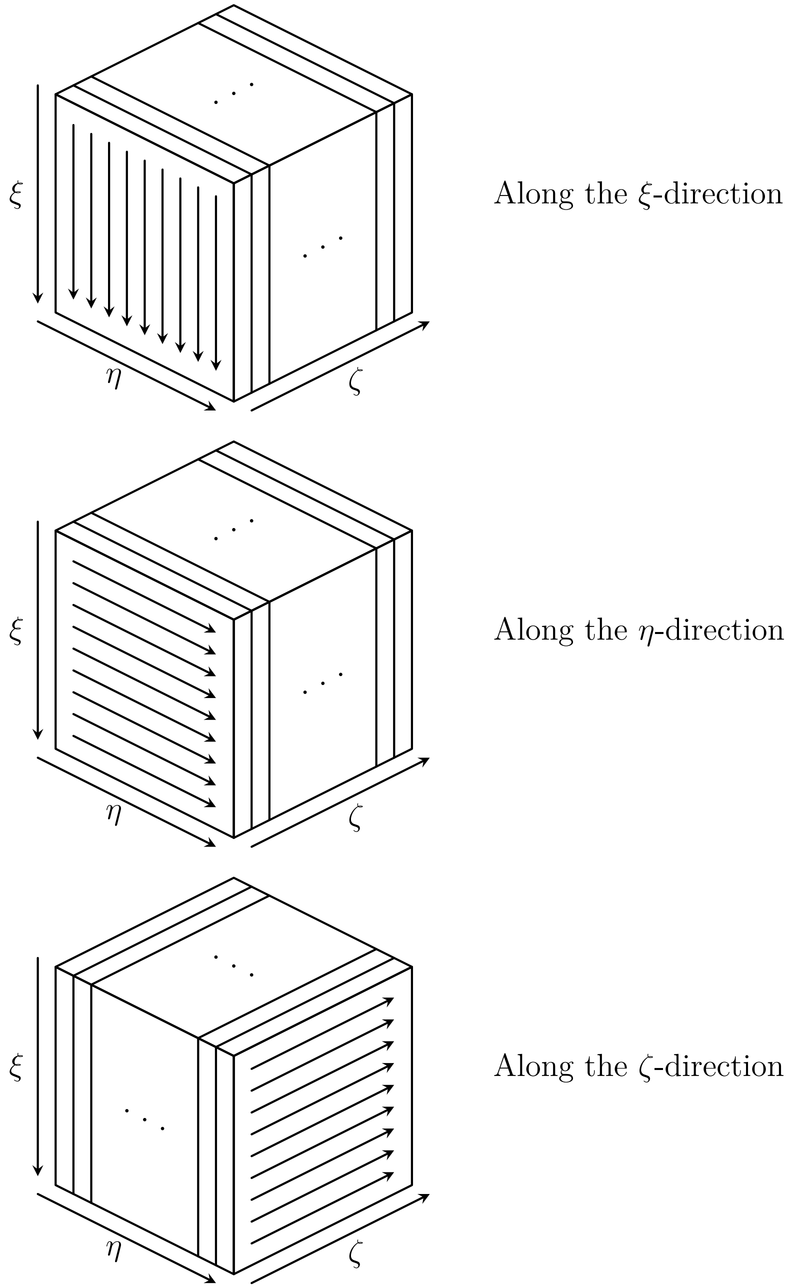
3.4.1 Manipulating four-dimensional arrays using two-dimensional computational kernels

Although the compact spatial partial differentiation and spatial filtering schemes are defined on individual vectors, in three-dimensional LES, they are applied to collections of vectors stored in four-dimensional arrays. The vectors in one collection share the same size and the same spatial orientation, but that orientation can be aligned with any of the ξ-, η- and ζ-directions of the computational space. From the perspective of their application to the data structure, on a column-major four-dimensional array used to store the flow variable values, the aforementioned operations are applied to along one of the first three dimensions and repeated uniformly across the remaining dimensions. In theory, taking advantage of the built-in support for array slicing of Fortran in the form of its array section notation, we can straightforwardly implement the communication-free computation of the two operations using one-dimensional computation kernels and let the compiler optimize the application of those one-dimensional computation kernels on four-dimensional arrays. In practice, however, the compiler is usually incapable of performing reliable and effective code optimizations for such scenarios because that involves a series of nontrivial loop nest transformations. In particular, it is typically unable to interleave the operations applied to multiple vectors to promote data locality and exploit opportunities of vectorization because such a capability is generally limited to elemental computation.

This situation necessitates manual optimization. Our manually optimized implementation is based on two-dimensional computation kernels. For each computation kernel, we prepare two versions, namely a columnwise version and a rowwise version. When the computation implemented by a computation kernel is applied along the ξ-direction, we divide each four-dimensional array to operate on into slabs of dimensions $(N_\xi/p_\xi) \times (N_\eta/p_\eta)$ and repeatedly apply the columnwise version of the computation kernel on each slab. When the computation is applied along the η- and ζ-directions, we divide each four-dimensional array into slabs of dimensions $(N_\xi/p_\xi) \times (N_\eta/p_\eta)$ and $(N_\xi/p_\xi) \times (N_\zeta/p_\zeta)$, respectively, and repeatedly apply the rowwise version of the computation kernel on each slab. Figure 3.1 illustrates the three cases.

3.4.2 Optimizing the two-dimensional computation kernels

In all cases, the two-dimensional slabs retain the first dimension of the four-dimensional array as their first dimensions and thus are stored contiguously in memory in those dimensions. We optimize the two-dimensional computation kernels involved in the compact spatial partial differentiation and spatial filtering schemes from the perspectives of vectorization and loop tiling.



Along the ξ-direction

Along the η-direction

Along the ζ-direction

Figure 3.1. Application of two-dimensional computation kernels along the three coordinate directions

Optimization of the rowwise versions of the computation kernels is relatively simple because they operate uniformly on array elements belonging to the same columns of the two-dimensional slabs. Those array elements occupy contiguous storage in memory and thus enable the computational kernels to be straightforwardly vectorizable. Such vectorization is well within the reach of the capabilities of modern optimizing compilers. Hence, we only need to organize the loop nests into forms such that automatic vectorization is not impeded. In comparison, optimization of the columnwise versions of the computation kernels is more complicated. To begin with, naïve vectorization which accesses all columns of a two-dimensional slab will induce very inefficient strided memory access patterns. On the other hand, complete avoidance of vectorization misses the opportunity to take advantage of the single-instruction multiple-data (SIMD) capabilities of modern high-performance processors. Therefore, an eclectic approach is necessary. The strategy which we adopt is to use explicit loop tiling to limit the vector length and reduce the impact of the strided memory access patterns.

Consider for example the optimized implementation of the columnwise version of the Thomas algorithm illustrated in Listing 3.1. Observe that it operates on b vectors simultaneously where b is a constant tiling factor. By introducing scalar temporary variables to induce the compiler to generate the minimum number of memory access instructions, the compiled code of the procedure `TiledThomasSolve` in Listing 3.1 performs $1.5b$ floating-point operations and $3b + 1.5$ memory accesses per iteration of the two inner `i`-loops on average. If $b = 1$, the code degenerates into an untiled version where each floating-point operation requires three memory accesses. On modern processor microarchitectures, where floating-point operations and memory accesses hitting the top-level data cache have similar throughput and latency, this memory-to-floating point ratio is far more than the top-level data cache can keep up with. In contrast, if $b = 4$, every floating-point operation requires only 2.25 memory accesses, which is much closer to the theoretical lower bound of two. However, the value of b should not be made arbitrarily large since the available vector registers are

Listing 3.1. Loop-tiled implementation of the columnwise version of the Thomas algorithm

```
SUBROUTINE TiledThomasSolve(l, d, u, X)

  DOUBLE PRECISION, INTENT(IN)    :: l(:), d(:), u(:)
  DOUBLE PRECISION, INTENT(INOUT) :: X(:, :)

  INTEGER                         :: n, m, i, j, k

  n = SIZE(X, 1)
  m = SIZE(X, 2)

  ! Let b be a constant tiling factor

  DO j = 1, m, b
    DO i = 2, n
      DO k = 0, b - 1
        X(i, j + k) = X(i, j + k) - l(i) * X(i - 1, j + k)
      END DO
    END DO
    DO k = 0, b - 1
      X(n, j + k) = X(n, j + k) / d(n)
    END DO
    DO i = n - 1, 1, -1
      DO k = 0, b - 1
        X(i, j + k) = (X(i, j + k) - u(i) * X(i + 1, j + k)) / d(i)
      END DO
    END DO
  END DO

END SUBROUTINE
```

limited in both length and number. Too large a value of b will cause spills to memory and introduce extra memory accesses. Furthermore, when the number of vectors in a slab is not an exact multiple of b, there can be any number from zero up to $b-1$ vectors which need to be handled by an untiled remainder loop. Also, the marginal decrease in the memory-to-floating point ratio,

$$\frac{3b+1.5}{1.5b} - \frac{3(b+1)+1.5}{1.5(b+1)} = \frac{1}{b(b+1)}, \tag{3.1}$$

diminishes quadratically as b increases.

The optimal value of the tiling factor b depends on the computing platform and the computation kernel and thus needs to be determined via benchmarking. In Figure 3.2, we plot the speedup achieved by the loop-tiled implementation of the columnwise version of the Thomas algorithm using loop tiling factors $b = 1, 2, \ldots, 20$ with respect to the case where $b = 1$. We consider two scenarios where the four-dimensional arrays are of dimensions $90 \times 90 \times 90 \times 5$ and $28 \times 28 \times 28 \times 5$, respectively. The measurements are conducted using an AMD Opteron 8350 processor. We can make the following observations regarding Figure 3.2:

- In general, optimal performance is achieved when b is a divisor of N_η/p_η since the untiled remainder loop is avoided. To the contrary, when $(N_\eta/p_\eta) \bmod b$ is close to $b-1$, the performance is degraded because the remainder loop needs to execute many iterations.

- In the range where b is relatively small, larger values of b achieve higher performance because cross-column vectorization helps hide the memory access latency.

- In the range where b is relatively large, the performance is suboptimal even when the remainder loop is avoided due to the greater register pressure and spills to memory.

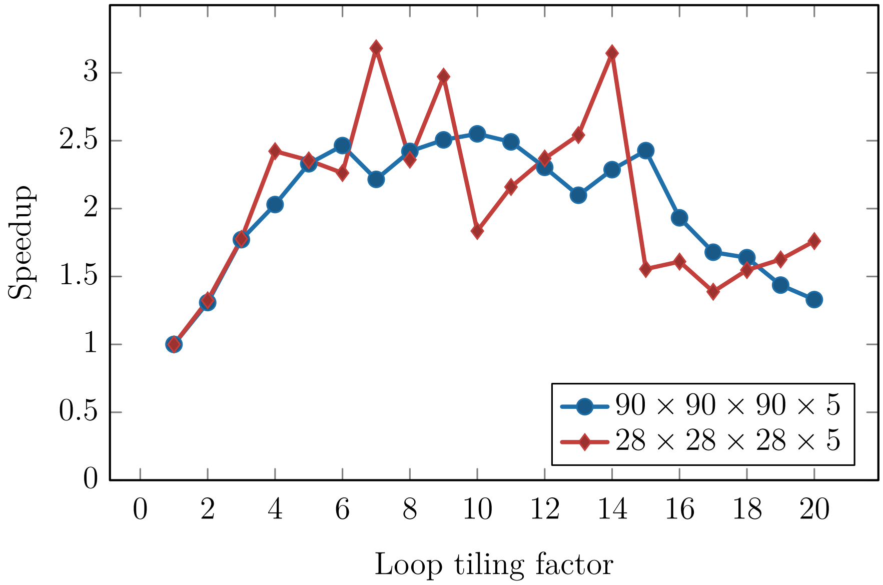


Figure 3.2. Speedup achieved by the columnwise Thomas algorithm using different loop tiling factors

Since no single value of b works the best for all cases, in our actual implementation, we use four different loop tiling factors. The vectors are processed using an eight-wide vectorized code path, and the remainder loop is implemented in terms of a four-wide vectorized, a two-wide vectorized and an unvectorized code path.

Although the above discussion is focused on the Thomas algorithm, the same optimization technique can also be applied to the stencil computation involved in evaluation of the right-hand side vectors in the compact spatial partial differentiation and spatial filtering schemes. Other than vectorization, we apply an additional level of loop tiling along the the span direction of the stencil. By tiling stencil computation by the half width of the stencil, we ensure that each array element is loaded and stored exactly once.

3.5 Implementation validation

We validate our implementation of the three-dimensional LES-based jet engine noise prediction application on Kraken, Ranger and Carter, three clusters hosted at the National Institute of Computational Sciences (NICS) at the University of Tennessee, the Texas Advanced Computing Center (TACC) at the University of Texas at Austin, and the Rosen Center for Advanced Computing (RCAC) at Purdue University, respectively.

We consider nine validation problems based on the experimental case SP07 from [71] with grids of dimensions $292 \times 128 \times 128$, $500 \times 218 \times 218$ and $810 \times 354 \times 354$. The largest problems use grids consisting of over 100 million grid points and are considered production-class problems. We run each of the validation problems to completion, taking between 210,000 and 600,000 time steps during time integration depending on the grid size and the problem configuration. The experimental results validate the implementation. We refer the reader to [49] for details of the experimental setup and interpretation of the numerical results.

3.6 Performance experiments

3.6.1 Experimental setup

We evaluate the performance of our implementation of the LES-based jet engine noise prediction application using a series of strong scaling experiments. We use a test problem which has a grid of dimensions 1,260 × 1,260 × 1,260 consisting of just over two billion grid points. Although a full jet simulation on a grid of this large a size typically needs to last for between 500,000 and 1,000,000 time integration steps and requires checkpointing to guard against hardware failures, we limit the number of time integration steps and disable checkpointing to shorten the turnaround times of the performance experiments.

We conduct the performance experiments on Kraken and Ranger. Their hardware configurations are listed in Table 3.1. On Kraken, we vary the total number of processor cores participating in computation in a wide range and use seven configurations of between 2,744 and 91,125 processor cores. On Ranger, we use the same configurations except the largest one due to its lower total count of processor cores. We replace that configuration with one of 54,872 processor cores. The 2,744-core configuration is considered the baseline for performance evaluation. In every configuration, the processor cores are organized into a three-dimensional Cartesian logical grid with the same number of processor cores in each dimension. Performance statistics are measured using a combination of various performance monitoring tools including the `MPI_WTIME` function, PAPI [53], CrayPat [17] and a custom performance monitoring module.

3.6.2 Experimental results

Figures 3.3 and 3.4 plot the parallel speedup and efficiency achieved by our implementation of the LES-based jet engine noise prediction application. As is evident from Figure 3.3, the application maintains a steady trend of speedup growth as the

Table 3.1.
Hardware configurations of Kraken and Ranger

	Kraken	Ranger
System	Cray XT5	Sun Blade
Nodes	9,408	3,936
Cores	112,896	62,976
Processor model	AMD Opteron 2435	AMD Opteron 8354
Frequency	2.6 GHz	2.3 GHz
L1 data cache/core	64 KB	64 KB
L2 cache/core	512 KB	512 KB
L3 cache/core	6 MB	2 MB
Cores/socket	6	4
Sockets/node	2	4
Memory/node	16 GB	32 GB
Interconnect	Cray SeaStar2+	Sun Constellation

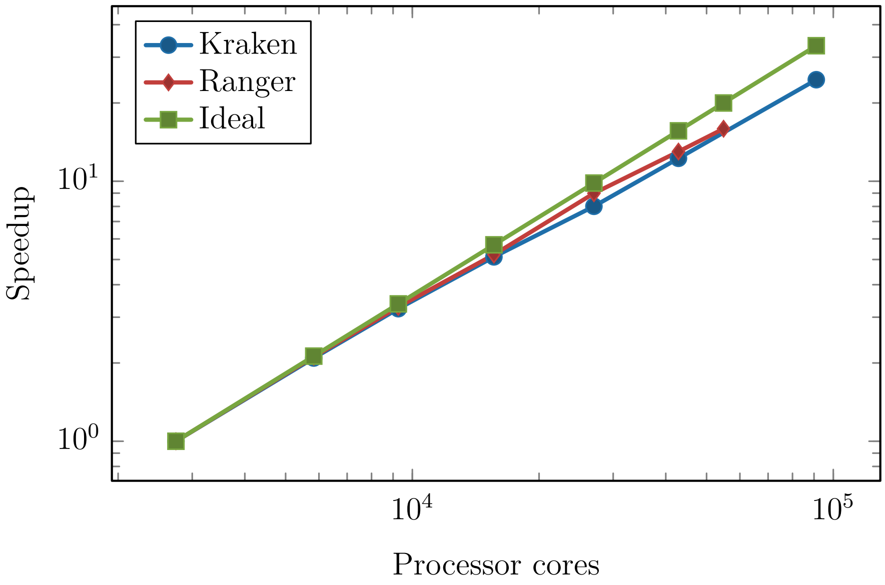


Figure 3.3. Parallel speedup achieved by the large eddy simulation-based jet engine noise prediction application

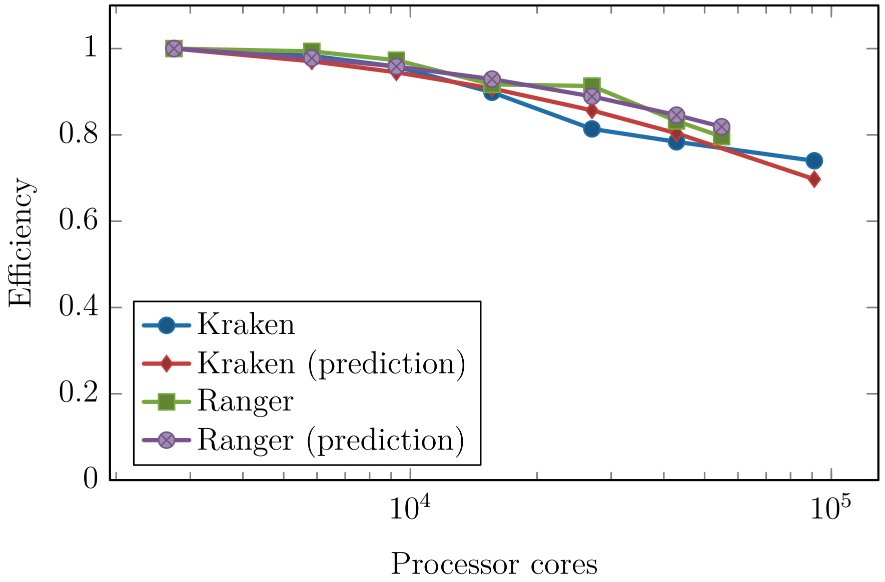


Figure 3.4. Parallel efficiency achieved by the large eddy simulation-based jet engine noise prediction application

number of processor cores increases. Figure 3.4 provides an even clearer picture of the efficiency of the application in utilizing large numbers of processor cores. Compared to the 2,744-core baseline cases, the parallel efficiency decreases only modestly, reaching 74 % at 91,125 processor cores on Kraken and 80 % at 54,872 processor cores on Ranger.

In addition to the measured parallel efficiency, in Figure 3.4, we also include the theoretically predicted relative parallel efficiency derived from fitting

$$E_{\text{rel}}(p^3) = \left(\frac{1}{1 + c(p/N)^2}\right) \bigg/ \left(\frac{1}{1 + c(p_{\text{base}}/N)^2}\right) \tag{3.2}$$

to the measured data where $N = 1{,}260$ and p are the respective sizes of the dimensions of the computational grid and the logical grid of processor cores, and p_{base} corresponds to the baseline cases where $p = 12$. Equation (3.2) originates from the isoefficiency function of the truncated SPIKE algorithm in Equation (2.19) and takes into account the fact that the calculated parallel efficiency is relative to the baseline cases. In Equation (3.2), c is a constant factor determined by minimizing

$$\left|\prod E_{\text{rel}}(p_k^3) - 1\right|$$

where p_k assumes all values of p. As Figure 3.4 shows, the parallel efficiency predicted by the theoretical scalability analysis in section 2.3 closely matches the empirical measurements.

Finally, we compare the performance of the truncated SPIKE algorithm against that of the transposition method described in Section 1.3.1 in Figure 3.5. Since, the implementation of the transposition method in the codebase from [74] is based on a one-dimensional computational partitioning, we have to create our own reference implementation of the method. We modify our implementation of the LES-based jet engine noise prediction application to replace the truncated SPIKE algorithm with the transposition method in the compact spatial partial differentiation module. We leave the spatial filtering module unchanged even though it is also based on the

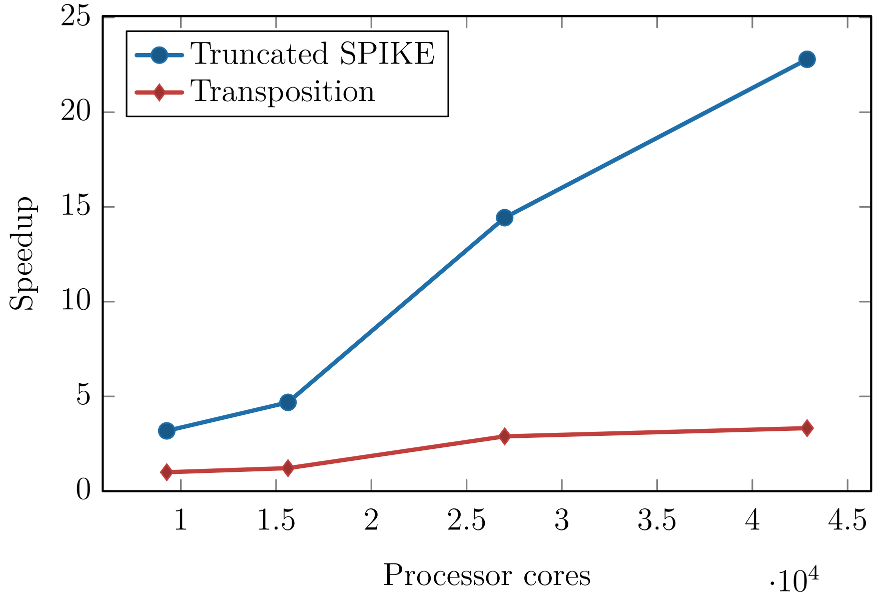


Figure 3.5. Performance comparison of the truncated SPIKE algorithm and the transposition method in the large eddy simulation-based jet engine noise prediction application

truncated SPIKE algorithm because its contribution to the total computational cost is insignificant. We measure the running times of the two implementations of ours in four strong scaling experiments using between 9,261 and 42,875 processor cores. We normalize the measured data with respect to the running time of the implementation based on the transposition method in the 9,261-core experiment. As Figure 3.5 shows, the speedup growth of the transposition method quickly flatlines as the number of processor cores increases, while the truncated SPIKE algorithm maintains a steady upward trend and exhibits an increasing performance advantage over the transposition method.

4 A PROGRAMMING MODEL BASED ON GENERALIZED ELEMENTAL SUBROUTINES FOR REGULAR GRID-BASED NUMERICAL APPLICATIONS

4.1 Introduction

Three-dimensional LES-based jet engine noise prediction belongs to a category of numerical applications which we can refer to as the *regular grid-based numerical applications*. In a regular grid-based numerical application, the computational grid which represents the computational space is divided into grid partitions each of which assumes the topology of a regular grid. The regularity of the structure of the grid partitions enables the application, when implemented in the SPMD programming style, to effectively take advantage of multidimensional arrays, which in term paves the ground for generalizing elemental computation, a programming model feature the desire for which has been explained in section 1.4.2.

In this chapter, we describe a programming model which realizes the generalization of elemental subroutines in a straightforward fashion [68]. We discuss the semantic model of the generalized elemental subroutines in detail and present the code transformation processes for implementing them through source code rewriting.

4.2 Semantic model and proof-of-concept implementation of generalized elemental subroutines

4.2.1 Semantic model

Consider a subroutine f accepting n parameters $\langle x_1, x_2, \ldots, x_n \rangle$ which is intended to be used as a generalized elemental subroutine. For our purposes, we allow $x_1, x_2, \ldots, x_n$ to be arrays in addition to scalars. Subroutine f can access the parameters as well as local variables defined in its subroutine body. The lifetime of any local variable is required to terminate when the control flow leaves the subroutine. Within the subroutine body, we allow the common arithmetic operations as well as constructs which affect the control flow including `DO` loops and `IF` conditions. We also allow invocation of other generalized elemental subroutines, but recursion in any direct or indirect forms is not supported. Finally, some forms of communication operations which do not impact the persistent program state other than the parameters (e.g., global variables and local variables with static lifetimes) are supported. As a syntactic mechanism to enforce this last restriction, we require that matching pairs of send and receive operations must be contained in f. This requirement ensures that it is not possible to have a dangling send or receive operation when execution of f terminates.

The semantics of invoking the generalized elemental subroutine f with arguments $\langle Y_1, Y_2, \ldots, Y_n \rangle$ is defined by a process which repeatedly invokes f regarding it as an ordinary subroutine. Each combinations of arguments $\langle y_1, y_2, \ldots, y_n \rangle$ to be used as $\langle x_1, x_2, \ldots, x_n \rangle$ during the repeated invocation is taken from $\langle Y_1, Y_2, \ldots, Y_n \rangle$ following the rules explained below. To define the rules precisely, we first introduce several formal notations. For each x_k, we identify its dimensions with a unique sequence of symbols

$$I_k = \langle i_1^{(x_k)}, i_2^{(x_k)}, \ldots, i_{d_k}^{(x_k)} \rangle \tag{4.1}$$

where d_k is its dimensionality. We call these symbols *dimension identifiers*. Depending on the occasion, we also interpret I_k as a set for the sake of simplicity in notations. We also introduce an extra set of dimension identifiers

$$J = \{\, j_1, j_2, \ldots, j_m \,\} \tag{4.2}$$

which are not tied to any of the dimensions of $x_1, x_2, \ldots, x_n$. When f is invoked with the arguments $\langle Y_1, Y_2, \ldots, Y_n \rangle$, for each Y_k, the programmer assigns a distinct dimension identifier from the set $I_k \cup J$ to each of its dimensions. We require that all elements of I_k appear in the dimension identifier assignment for Y_k. This implies that the dimensionality of Y_k must be no less than that of x_k. The dimension identifier assignment defines an injective dimension map from the dimensions of x_k to those of Y_k. By choosing an index for each dimension of Y_k which is not the image corresponding to a dimension of x_k in the dimension map, i.e., each dimension of Y_k which is assigned a dimension identifier from J, an array slice y_k can be extracted from Y_k that can be used as x_k for during invocation of f. Furthermore, since $Y_1, Y_2, \ldots, Y_n$ share the same set of extra dimension identifiers, we can naturally consider several dimensions from different members of $\{ Y_1, Y_2, \ldots, Y_n \}$ which are assigned the same dimension identifiers to be linked and traverse them using a single index.

Given the above construction, we are now ready to define the precise semantics of invoking the generalized elemental subroutine f with the arguments $\langle Y_1, Y_2, \ldots, Y_n \rangle$. For each dimension identifier $j_k \in J$, the programmer selects an index range

$$L_k = \{ 1, 2, \ldots, l_k \} . \tag{4.3}$$

Then for each multidimensional index in the Cartesian range $L_1 \times L_2 \times \cdots \times L_m$, we extract an array slice y_k from Y_k following the aforementioned rules and invoke f with arguments $\langle y_1, y_2, \ldots, y_n \rangle$, using it as an ordinary subroutine. To enable aggressive code optimizations, we allow instances of repeated invocation of f to be executed in any order and operations belonging to different instances of repeated invocation to be interleaved arbitrarily. Both of these two behavioral characteristics are inherited from the semantics of traditional elemental subroutines.

Here we provide a concrete example to illustrate the above description. We choose matrix multiplication as our example. While matrix multiplication is not generally considered a regular grid-based numerical application, it suffices for the purpose of demonstrating the concepts of generalized elemental subroutines. Listing 4.1 shows a simple generalized elemental subroutine `Dot` for the vector dot product written using the Fortran syntax. The arguments have the following dimension identifiers,

Listing 4.1. Generalized elemental subroutine for the vector dot product

```
SUBROUTINE Dot(n, a, b, c)

  INTEGER :: n
  REAL    :: a(:), b(:), c

  INTEGER :: i

  DO i = 1, n
    c = c + a(i) * b(i)
  END DO

END SUBROUTINE
```

respectively:

$$\begin{aligned}\mathtt{n} &\to \emptyset,\\ \mathtt{a} &\to \left\langle i_1^{(\mathtt{a})}\right\rangle,\\ \mathtt{b} &\to \left\langle i_1^{(\mathtt{b})}\right\rangle,\\ \mathtt{c} &\to \emptyset.\end{aligned}$$

In order to use this subroutine to implement the matrix multiplication `C = A * B + C` where `A`, `B` and `C` are all two-dimensional arrays, we specify the following dimension identifier assignment:

$$\begin{aligned}\mathtt{n} &\to \emptyset,\\ \mathtt{a} &\to \left\langle j_1, i_1^{(\mathtt{a})}\right\rangle,\\ \mathtt{b} &\to \left\langle i_1^{(\mathtt{b})}, j_2\right\rangle,\\ \mathtt{c} &\to \left\langle j_1, j_2\right\rangle.\end{aligned}$$

Notice that this dimension identifier assignment links the two dimensions of `C` to the first dimension of `A` and the second dimension of `B`, respectively, while mapping the remaining dimensions of `A` and `B` to the only dimensions of `a` and `b`, respectively. Thus, each element of `C` is defined by an appropriate dot product of a row of `A` and a column of `B`, and the desired semantics of matrix multiplication is delivered.

4.2.2 Proof-of-concept implementation

To demonstrate the capabilities of generalized elemental subroutines, we implement a proof-of-concept programming tool which generates code that can be immediately integrated into regular grid-based numerical applications. The programming tool is implemented as a source code rewriter with a front end generated by ANTLR [57] and a custom code optimizer and generator written in Java. It processes generalized elemental subroutines defined in a domain-specific input language according to the dimension identifier assignments specified by the programmer and generates standard-compliant Fortran 90 code which implements the semantics of their invocations.

We borrow the core language syntax and semantics from Fortran 90 to form the foundation of our domain-specific input language. In this proof-of-concept implementation, we support only the intrinsic data types and ignore the derived types. To further simplify the language, we remove language constructs which either can be replaced by simple equivalents (e.g., the `DO WHILE` loops) or have disallowed semantics (e.g., the `ALLOCATABLE` and `SAVE` attributes). The support for communication is modeled after MPI. Communication primitives such as `COMM_SIZE`, `COMM_RANK`, `SEND` and `RECV` are provided as extension intrinsic subroutines for querying information about communicators as well as carrying out the actual communication operations. The programming tool understands the semantics of the communication primitives and is capable of performing optimizations on their invocations by assuming them as generalized elemental subroutines as well. To facilitate the recognition of the matching pairs of send and receive operations, we require the programmer to annotate each occurrence of the send and receive primitives with a compile-time constant tag. Pairs of send and receive primitives annotated with the same tag are considered to be matching. Of course, for broadcast and reduce primitives, which contain both send and receive semantics, such tags are unnecessary. An external runtime library provides implementations of these extension intrinsic subroutines for integration into applications.

4.3 Generating optimized code for generalized elemental subroutines

As can be inferred intuitively from the semantic model of generalized elemental subroutines, invoking them with arguments of higher dimensionalities than their parameters can be achieved trivially by constructing several multiply nested loops to iterate over the Cartesian index ranges and repeatedly executing their original subroutine bodies on array slices taken from the arguments. However, this simplistic approach is very likely to lead to unsatisfactory performance levels in practical scenarios due to a number of reasons including inefficient memory access patterns and high communication overhead. Hence, in order to ensure that our proof-of-concept programming tool delivers sufficient performance so that they are practical for use in implementation of regular-grid based numerical applications, we devise and implement several code optimizations which take advantage of the domain-specific high-level knowledge about the semantic model of generalized elemental subroutines. To be more specific, the code optimization process proceeds in three stages, namely loop nest generation, local variable transformation and subroutine invocation aggregation. We elaborate on each of these code optimizations in the following sections.

4.3.1 Loop nest generation

As mentioned above, one possible method of generating code that implements the semantics of an invocation of a generalized elemental subroutine is to simply wrap the subroutine body inside a loop nest which iterates over the Cartesian index range selected by the programmer. Although the resulting code will likely be inefficient in general, it nevertheless serves as a good starting point for a series of loop nest transformations which can enable highly efficient code.

Algorithm 4.1 lists the main algorithmic steps of the loop nest generation process. To disambiguate the terminologies, in Algorithm 4.1, we refer to a loop specified by the programmer in the generalized elemental subroutine as an *existing loop* and a loop generated for iteration over the Cartesian index range as a *new loop*. The algorithm assumes that each local variable in the subroutine has been implicitly augmented with a new dimension for each new loop to accommodate the potentially different array element values generated by different loop iterations. As the first step, it establishes a single initial loop nest consisting of all new loops, which forms the basis for further loop transformations. Then it carries out the the loop transformations in two phases. The first phase of loop nest generation performs loop distribution on the initial loop

Algorithm 4.1. Loop nest generation for generalized elemental subroutines

```
for each new loop in order starting from the innermost do        ▷ Perform loop distribution on each new loop
    Determine the admissible distribution locations for the current new loop
    for each top-level construct of the loop body do
        if the current construct contains communication or strided memory accesses related to the current new loop then
            Mark locations immediately before and after the current construct as distribution-requested
        end if
    end for
    Mark locations with an enforce-distribution directive as distribution-requested
    Unmark locations that have a prevent-distribution directive or are themselves inadmissible for loop distribution
    Perform loop distribution on the current new loop at the marked locations
end for
for each new loop nest enclosing a single top-level construct of the original subroutine body do
    if the construct is an existing loop nest then
        Perform loop permutation on the combined loop nest of both new and existing loops
        if the body of the loop nest is now immediately enclosed by one or more new loops then
            Invoke Algorithm 4.1 recursively on the body of the loop nest
        end if
    else if the construct is an IF construct then
        Perform IF condition hoisting
        Invoke Algorithm 4.1 recursively on the IF branches now immediately enclosed by new loops
    end if
end for
```

nest. The algorithm processes the new loops one by one starting with the innermost one. In general, the algorithm adopts a comparatively conservative strategy for loop distribution because as we will discuss in section 4.3.2, excessively aggressive loop distribution can unnecessarily inflate the memory footprint of the generated code. However, the algorithm does attempt to be aggressive when communication is involved, or the memory access pattern is unfavorable. Loop distribution in these cases can create opportunities for communication aggregation as we will discuss in section 4.3.3 and avoid strided memory accesses. The second phase of loop nest generation performs loop permutation and loop tiling as well as `IF` condition hosting. It also attempts to discover more code optimization opportunities by recursively invoking the entire Algorithm 4.1 on the loop bodies and `IF` branches which became immediately enclosed in new loops after the loop transformations.

In order to allow the programmer to explicitly influence the loop nest generation process, we provide four special directives in the programming tool. First, we have a pair of directives for enforcing and preventing distribution of the new loops at certain locations in a generalized elemental subroutine. They can be specified to be selective and apply to only a subset of the new loops but not the remaining ones. While the prevent-distribution directive is always semantically valid, the enforce-distribution directive may violate the semantics of generalized elemental subroutines in some occasions and is automatically ignored in such cases. The other two directives enable the programmer to declare semantic assumptions concerning the existing loops in the subroutine. One is used to indicate that an existing loop nest behaves like a generalized elemental subroutine, i.e., its iterations can be arbitrarily ordered and interleaved. The other one is used to specify a tiling factor for an existing loop. Together, these four directives provide the programmer with the ability to request certain loop transformations at specific locations when our programming tool fails to recognize some optimization opportunities.

4.3.2 Local variable transformation

In section 4.3.1, the methodology of loop nest generation assumes that each local variable of the generalized elemental subroutine is augmented with implicit new dimensions so that the semantics of repeated invocation is preserved during the process in which the initial loop nest is transformed. For the purpose of code generation, these implicit dimensions need to be made explicit, but not in all cases are they actually needed. For example, a local variable can have an implicit dimension of its ignored if its element values are private with respect to all distributed instances of the corresponding new loop. Furthermore, the dimension identifier assignment specified by the programmer may cause the dimensions of the parameters in the generated code to be in a different order than in the original subroutine. In this case, the dimensions of the local variables also have to be reordered in order to improve the memory access efficiency. We devise two separate algorithms to tackle these two problems, respectively.

We first consider the simpler problem of reordering the dimensions of the local variables. Algorithm 4.2 summarizes the basic methodology for our algorithm for the problem. To unify the terminologies, in the following discussion, we refer to both the parameters and the local variables of the generalized elemental subroutine as *variables*. To understand the method, consider the following simple example. Suppose that `A` and `B` are two two-dimensional arrays appearing in the same statement in two nested loops whose loop variables are `i` and `j`, respectively. If `A` is referenced in the form `A(i, j)`, intuitively, we would prefer that the dimensions of `B` be ordered in a way such that `B` is referenced in the form `B(i, j)`, i.e., its subscripts are in the same order as those of `A`, because otherwise strided memory accesses will be incurred on at least one of `A` and `B` regardless of the order in which the two loops are nested. Viewed from an alternative perspective, if we regard `A` as a reference variable for `B` when considering the dimension order of the latter, compared to `B(j, i)`, `B(i, j)` has a smaller number of reversely ordered dimension pairs with respect to `A` and is thus favored. Extending this idea

Algorithm 4.2. Local variable dimension reordering for generalized elemental subroutines

```
1: Construct an undirected graph containing all variables appearing in the generalized elemental subroutine as vertices where two variables are connected if they appear in the same context
2: Partition the variables into layers by their distances in the graph from any of the parameters
3: Reorder dimensions of each local variable in layer order using neighbors in the previous layer as reference variables
```

to cases where variables have more than two dimensions, for each local variable `X`, if we can define a set of reference variables whose dimension orders have been fixed and recognize a correspondence relation between the dimensions of `X` and those of its reference variables, then we can determine the dimension order of `X` by minimizing the total number of reversely ordered dimension pairs with respect to the reference variables.

In Algorithm 4.2, we choose the set of reference variables for each local variable by considering pairs of variables which appear in the same contexts. A context for this purpose can be either a single statement or a matching pair of send and receive primitives. The *appear-in-the-same-context* relation between the variables defines an undirected graph. We can propagate the dimension ordering information from the parameters to all local variables along the edges of the graph. To do this, we perform a breadth-first search (BFS) on the graph starting from the parameters and partition the variables into layers according to their distances from the parameters in the graph. Then variables in one layer can used as the reference variables for their neighbors in the next layer.

Having solved the problem of local variable dimension reordering, we now consider the more complex problem of eliminating the unneeded implicit new dimensions from the local variables. To eliminate a new dimension implicitly assumed for a local variable during loop nest generation, we consider the following two conditions:

Constantness: The element values of the local variable are identical across that dimension.

Privateness: The local variable can be considered private to the distributed instances of the new loop corresponding to that dimension.

Apart from these two intuitive conditions, we need an additional condition to ensure that the semantics of generalized elemental subroutines is not violated:

No upward exposure: The element value of the local variable generated in any iteration of a distributed instance of the new loop is not upward exposed to later iterations.

In order for an implicit dimension to be eligible for elimination, it must satisfy at least one of the constantness and the privateness conditions as well as the no upward exposure condition.

Using the standard data flow analysis techniques including the variable liveness, reaching definition and program dependence analyses, the three aforementioned conditions can be checked using Algorithm 4.3. Notice that as reflected by the last line of Algorithm 4.3, passing the tests of the algorithm does not mean that the implicit dimension in question can be immediately eliminated. In order for the elimination to be actually viable, all distributed instances of the new loop containing references of the local variable must be eligible for removal. Conversely, in order for those loop instances to be eligible for removal, every variable referenced in their loop bodies must either not have a corresponding dimension or have that dimension deemed eligible for elimination by Algorithm 4.3. This is the structural requirement for local variable dimension elimination. Forcing the involved loop instances to be removed ensures that dimension elimination cannot incidentally result in a reduction semantics on the local variable.

4.3.3 Subroutine invocation aggregation

Subroutine invocation aggregation is a relatively simple code transformation. It looks for each tight loop nest whose loop body consists of a single `CALL` statement and aggregate the repeated subroutine invocations into a single invocation. Since only invocations of generalized elemental subroutines are allowed in the programming model, the target subroutine of the aggregated invocation can be automatically generated by inferring a dimension identifier assignment from the loop nest and the `CALL` statement and carrying out the code transformation process recursively. An important function of

Algorithm 4.3. Local variable dimension elimination for generalized elemental subroutines

```
for each distributed instance of the new loop do
    if the local variable is not live at any entry or exit point of the current loop instance then
                                        ▷ The variable is private to the current loop instance
    else if any definition of the variable within the loop instance is program-dependent on any parameter with a dimension
        corresponding to the new loop then
        Reject elimination              ▷ The variable is neither constant nor private
        return
    end if
    if the variable is live at the entry point of the current loop instance and has a definition reaching any exit point of
        the loop instance then
        Reject elimination              ▷ The variable is upward exposed
        return
    end if
end for
Accept elimination if it is structurally viable
```

this subroutine invocation aggregation is that it aggregates all applicable invocations of the communication primitives, which are handled as if they were generalized elemental subroutines by our programming tool.

4.3.4 Example

Now we demonstrate the effect of the aforementioned code transformations using a simple but complete example. Listing 4.2 shows a generalized elemental subroutine `Stencil` for computing the following accumulative stencil operation:

$$y_i \coloneqq y_i + \begin{cases} x_2 & \text{if } i = 1 \\ x_{i-1} + x_{i+1} & \text{if } 1 < i < n, \\ x_{n-1} & \text{if } i = n. \end{cases} \tag{4.6}$$

The subroutine accepts a communicator `comm`, a vector length `n`, an input vector `x` and an accumulator vector `y` as its parameters. Suppose that we want to apply then stencil operation to vectors aligned along the third dimension of the computational space of a three-dimensional regular grid-based numerical application. In this case, the arguments `X` and `Y` to be supplied to the subroutine as `x` and `y` are three-dimensional arrays. For this purpose, with respect to the dimension identifiers

$$\begin{aligned} \texttt{comm} &\to \emptyset, \\ \texttt{n} &\to \emptyset, \\ \texttt{x} &\to \left\langle i_1^{(\texttt{x})} \right\rangle, \\ \texttt{y} &\to \left\langle i_1^{(\texttt{y})} \right\rangle, \end{aligned}$$

Listing 4.2. Generalized elemental subroutine for stencil computation

```
SUBROUTINE Stencil(comm, n, x, y)

  INTEGER          :: comm, n
  DOUBLE PRECISION :: x(:), y(:)

  INTEGER          :: nprc, rank, k
  DOUBLE PRECISION :: xprev, xnext

  CALL COMM_SIZE(comm, nprc)
  CALL COMM_RANK(comm, rank)

  IF (rank /= nprc - 1) THEN
    CALL SEND(x(n), rank + 1, comm, 0)  ! 0 is a tag
  END IF
  IF (rank /= 0) THEN
    CALL SEND(x(1), rank - 1, comm, 1)  ! 1 is a tag
  END IF

  y(1) = y(1) + x(2)
  DO k = 2, n - 1
    y(k) = y(k) + x(k - 1) + x(k + 1)
  END DO
  y(n) = y(n) + x(n - 1)

  IF (rank /= 0) THEN
    CALL RECV(xprev, rank - 1, comm, 0)  ! 0 is a tag
    y(1) = y(1) + xprev
  END IF
  IF (rank /= nprc - 1) THEN
    CALL RECV(xnext, rank + 1, comm, 1)  ! 1 is a tag
    y(n) = y(n) + xnext
  END IF

END SUBROUTINE
```

we specify the following dimension identifier assignment:

$$
\begin{aligned}
\texttt{comm} &\rightarrow \emptyset, \\
\texttt{n} &\rightarrow \emptyset, \\
\texttt{X} &\rightarrow \left\langle j_1, j_2, i_1^{(\texttt{x})} \right\rangle, \\
\texttt{Y} &\rightarrow \left\langle j_1, j_2, i_1^{(\texttt{y})} \right\rangle.
\end{aligned}
$$

In this dimension identifier assignment, the third dimensions of `X` and `Y` are mapped to `x` and `y`, respectively. The stencil operation is repeated uniformly across the first dimensions of `x` and `y`.

Listing 4.3 shows the generated code which implements the semantics of the invocation of the generalized elemental subroutine `Stencil`. We have edited the code to replace the automatically generated variable names with easier-to-understand ones and removed the unnecessary loop labels. Other than these editorial modifications, the code remains identical. Our programming tool introduces two extra parameter `m1` and `m2` to represent the sizes of the first two dimensions of the transformed `x` and `y`. They also define the Cartesian index range to be iterated over. The scalar local variables `xprev` and `xnext` are changed into two-dimensional arrays. In the meantime, the other local variables are kept as scalars because they are proved to have constant values during repeated invocation. Two new `i`- and `j`-loops are introduced for the semantics of repeated execution. They enclose all the computation statements but are collapsed around the communication primitives to become aggregated subroutine invocations. Without any algorithmic changes, the generated code is exactly what the programmer is expected to produce by hand.

4.4 Empirical evaluation

We evaluate the practical feasibility of generalized elemental subroutines in the context of the three-dimensional LES-based jet engine noise prediction application

Listing 4.3. Code for stencil computation in the three-dimensional space generated using generalized elemental subroutines

```
SUBROUTINE Stencil(comm, n, x, y, m1, m2)

  INTEGER          :: comm, n, m1, m2
  DOUBLE PRECISION :: x(:, :, :), y(:, :, :)

  INTEGER          :: nprc, rank, k, i, j
  DOUBLE PRECISION :: xprev(m1, m2), xnext(m1, m2)

  CALL COMM_SIZE(comm, nprc)
  CALL COMM_RANK(comm, rank)

  IF (rank /= nprc - 1) THEN
    CALL SEND(x(1 : m1, 1 : m2, n), rank + 1, comm, 0)
  END IF
  IF (rank /= 0) THEN
    CALL SEND(x(1 : m1, 1 : m2, 1), rank - 1, comm, 1)
  END IF

  DO j = 1, m2; DO i = 1, m1
    y(i, j, 1) = y(i, j, 1) + x(i, j, 2)
  END DO; END DO
  DO k = 2, n - 1; DO j = 1, m2; DO i = 1, m1
    y(i, j, k) = y(i, j, k) + x(i, j, k - 1) + x(i, j, k + 1)
  END DO
  DO j = 1, m2; DO i = 1, m1
    y(i, j, n) = y(i, j, n) + x(i, j, n - 1)
  END DO; END DO

  IF (rank /= 0) THEN
    CALL RECV(xprev(1 : m1, 1 : m2), rank - 1, comm, 0)
    DO j = 1, m2; DO i = 1, m1
      y(i, j, 1) = y(i, j, 1) + xprev(i, j)
    END DO; END DO
  END IF
  IF (rank /= nprc - 1) THEN
    CALL RECV(xnext(1 : m1, 1 : m2), rank + 1, comm, 1)
    DO j = 1, m2; DO i = 1, m1
      y(i, j, n) = y(i, j, n) + xnext(i, j)
    END DO; END DO
  END IF

END SUBROUTINE
```

described in [49]. We consider implementation of the Thomas algorithm for solving tridiagonal linear systems and the sixth-order compact spatial partial differentiation scheme using generalized elemental subroutines and their integration into the existing codebase from [49]. We use two metrics in the evaluation. First, we measure the performance of the automatically generated code. We need to confirm that our programming tool is capable of delivering a competitive level of performance so that it is practical to use in composing regular grid-based numerical applications. Second, we estimate the difficulty of programming. Specifically, we use of the count of lines of code as our metric. Since we implement all the test cases ourselves and include a pretty-printer in the code generator of our programming tool which closely imitates our own code style, this metric should fairly accurately reflect the difficulty of programming.

4.4.1 Thomas algorithm

First, we consider implementing the Thomas algorithm for solving tridiagonal linear systems. As has been explained in section 1.2, in a three-dimensional LES-based jet engine noise prediction application, tridiagonal linear systems are solved with right-hand side vectors aligned along each of the ξ-, η- and ζ-directions. Although the Thomas algorithm consists of a factorization step and a solve step, since the factorization step is executed only once and also computationally inexpensive, we consider only the solve step. Using generalized elemental subroutines, we only need to provide an implementation of the algorithm which assumes a single right-hand side vector. The code for the implementation is shown in Listing 4.4.

We use a test case with a grid of dimensions $500 \times 500 \times 500$. We run the algorithm along each of the three dimensions. Thus, each run involves 500×500 right-hand side vectors each of order 500. We compare two implementations of the Thomas algorithm, namely one generated by our programming tool by processing the generalized elemental subroutine `ThomasSolve` in Listing 4.4 and another one which repeatedly executes `ThomasSolve` in a simplistic manner without the code optimizations presented in

Listing 4.4. Generalized elemental subroutine for the Thomas algorithm

```
SUBROUTINE ThomasSolve(n, dl, d, du, x)

    INTEGER          :: n
    DOUBLE PRECISION :: dl(:), d(:), du(:), x(:)

    INTEGER          :: i

    DO i = 1, n - 1
        x(i + 1) = x(i + 1) - dl(i) * x(i)
    END DO
    x(n) = d(n) * x(n)
    DO i = n - 1, 1, -1
        x(i) = d(i) * x(i) - du(i) * x(i + 1)
    END DO

END SUBROUTINE
```

section 4.3. As illustrated in Figure 4.1, our programming tool generates code which significant outperforms the simplest possible handwritten code while taking mostly the same amount of programming effort. The speedup ranges from 1.9 to 8.1 depending on along which direction the algorithm is run. The performance gain mainly comes from the appropriate interleaving of the arithmetic operations pertaining to multiple vectors, which reduces cache misses and enables automatic vectorization.

4.4.2 Sixth-order compact spatial partial differentiation scheme

Next, we consider the sixth-order compact spatial partial differentiation scheme in Equation (1.4). Implementation of the five-point stencil in the right-hand side of Equation (1.4) is straightforward. The tridiagonal linear system defined by the left-hand side of Equation (1.4) is solved using the truncated SPIKE algorithm detailed in Chapter 2.

We implement the sixth-order compact spatial partial differentiation scheme using the following six generalized elemental subroutines:

- differentiation driver,
- right-hand side stencil,
- SPIKE driver,
- Thomas algorithm,
- SPIKE reduced system,
- SPIKE solution retrieval.

All of these subroutines are implemented to handle just a single vector, and we use our programming tool to generate code that handles multiple vectors aligned along each of the three dimensions of the computational space.

As with the previous evaluation scenario, we also compare an implementation generated by our programming tool with an implementation which uses simple repeated

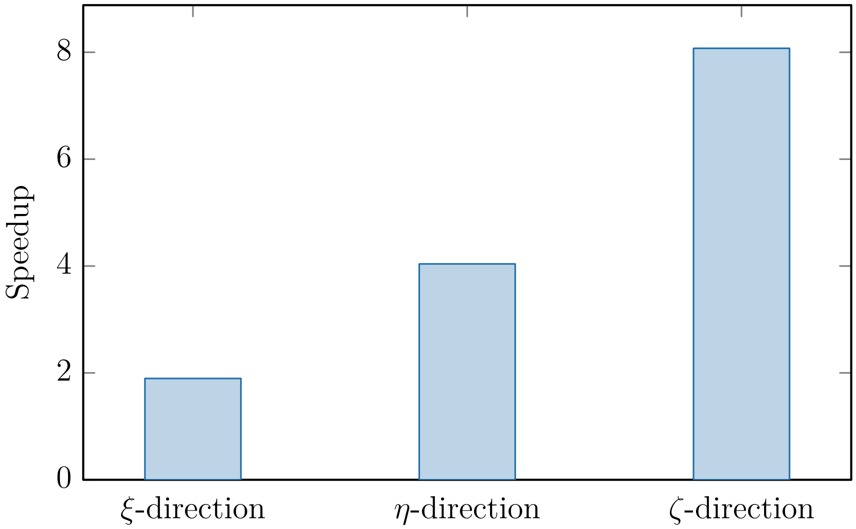


Figure 4.1. Speedup achieved by generalized elemental subroutines for the Thomas algorithm over simple repeated subroutine invocation

subroutine invocation. We run both implementations on a grid of dimensions $400 \times 400 \times 400$ using 512 processor cores arranged into a logical grid of dimensions $8 \times 8 \times 8$, assigning a distinct grid partition of dimensions $50 \times 50 \times 50$ to each processor core. Figure 4.2 shows the speedup achieved by the automatically optimized implementation with respect to the unoptimized implementation. Notice that the code generated by our programming tool is over 100 times faster than the simplest possible handwritten code for each of the three directions. The main contributor in such impressive boost in performance is communication aggregation. Instead of sending and receiving the data pertaining to a single vector at a time, the optimized implementation performs communication in batches of 400×400 vectors and thus is much more efficient in terms of the communication initiation cost and network bandwidth utilization.

4.4.3 Application in finite difference-based three-dimensional large eddy simulation of jet engine noise

Finally, we look at the application of generalized elemental subroutines in three-dimensional LES of jet engine noise. We again focus on the compact spatial partial differentiation scheme because in the implementation of the application from [49], it accounts for about two thirds of the total computational cost, of which one half comes from arithmetic operations, and the other half is due to communication. Compared to the synthetic scenario in section 4.4.2, spatial partial differentiation in this real-world application is applied on four-dimensional arrays whose added fourth dimensions are used to distinguish the different flow variables. Because our programming tool for generalized elemental subroutines outputs Fortran code, we are able to modify the application incrementally by substituting only the modules which implement the compact spatial partial differentiation scheme.

For empirical performance experiments, we consider two test scenarios. First, we use a microbenchmark which executes 100 successive spatial partial differentiation tasks along each of the three dimensions of the computational space. We use a grid

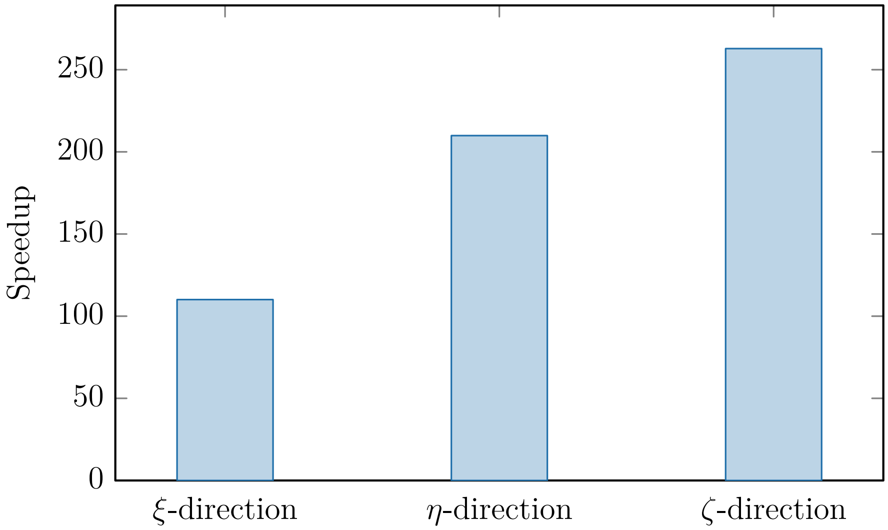


Figure 4.2. Speedup achieved by generalized elemental subroutines for the sixth-order compact spatial partial differentiation scheme over simple repeated subroutine invocation

of dimensions 512 × 512 × 512 and 4,096 processor cores arranged into a logical grid of dimensions 16 × 16 × 16 for this microbenchmark. Second, we use a realistic jet simulation problem with a grid of dimensions 896 × 384 × 384. We run the problem using 4,032 processor cores arranged into a logical grid of dimensions 28 × 12 × 12. In both scenarios, we assign a distinct grid partition of dimensions 32 × 32 × 32 to each processor core. Because we focus on only the computational efficiency of the simulation proper, we limit the simulation to last for just ten time integration steps, disable checkpointing and exclude the cost of initiation and termination. We measure all performance statistics using PAPI.

We compare the performance of using three different implementations of the compact spatial partial differentiation scheme in the LES-based jet engine noise prediction application. The first implementation uses code generated by our programming tool using an approach similar to that in section 4.4.2. The second implementation is the one presented in [49]. Its code is completely handwritten and include extensive optimizations specific to Kraken, the computing platform used for these performance experiments, to induce the compiler to generate the desired sequences of instructions. The third implementation uses the transposition method described in section 1.3.1 and serves as the baseline for performance comparison. We name these three implementations GES, HAND and XPOSE, respectively, to simplify references to them below.

Figure 4.3 shows the running times of the spatial partial differentiation microbenchmark. We note that the performance discrepancy between spatial partial differentiation along the three directions of the computational spaces is caused by the MPI process-to-processor core mapping computed by the `MPI_CART_CREATE` subroutine, which happens to favor the ζ-direction on Kraken. From Figure 4.3, it is evident that the truncated SPIKE algorithm delivers much better performance than the transposition method. Furthermore, the implementation of the truncated SPIKE algorithm generated by our programming tool displays performance advantages of 6.3 % and 7.0 % over the handwritten implementation for spatial partial differentiation tasks

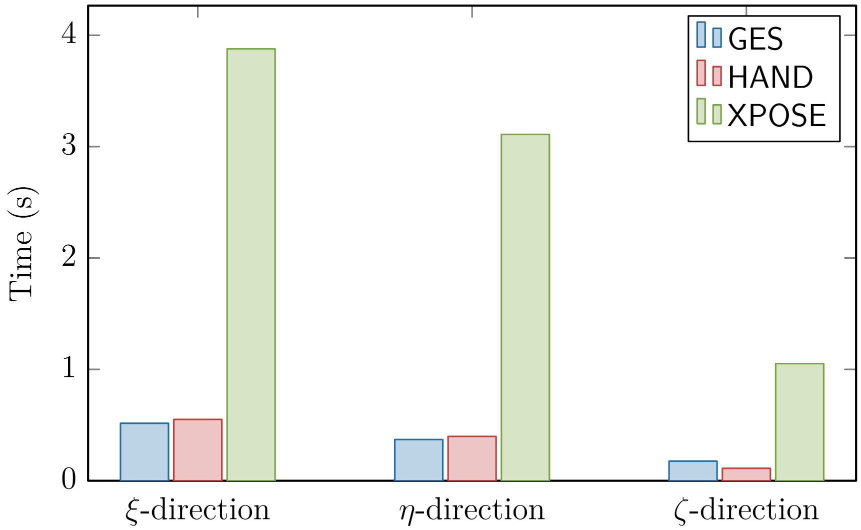


Figure 4.3. Running times of the three implementations of the sixth-order compact spatial partial differentiation scheme

along the ξ- and η-directions, respectively. For the ζ-direction, the generated code is 58 % slower because the handwritten code chooses an unusual loop nesting order for the embedded Thomas algorithm which puts a loop which iterates over the third dimensions of four-dimensional arrays inside another loop which iterates over the second dimensions of the same arrays. Such a loop nesting order greatly improves the L1 data cache reuse because it reduces the maximum size of the working set so that the backward sweep of the Thomas algorithm can take advantage of the data which have already been loaded into the L1 data cache by the forward sweep. The loop nest generation algorithm in Algorithm 4.1 is unable to discover this loop nesting order. But since this computation pattern is specific to the Thomas algorithm, we believe that the general effectiveness of the loop nest generation algorithm, as demonstrated by the spatial partial differentiation performance along the first two directions, is not significantly impacted.

Figure 4.4 shows the performance of the LES-based jet engine noise prediction application using each of the three implementations of the compact spatial partial differentiation scheme in the jet simulation problem. Again, the truncated SPIKE algorithm is shown to have a significant performance advantage over the transposition method. When integrated into the full application, compared to the handwritten code, the code generated by our programming tool is only 9.6 % and 14 % slower in computation and communication, respectively, and 12 % slower overall, which represents a decent level of performance considering the greatly reduced programming effort required as to be discussed below.

Lastly, we look at the programming effort involved in creating the three implementations of the sixth-order compact spatial partial differentiation scheme. Table 4.1 lists the counts of lines of code needed by the individual components of the compact spatial partial differentiation scheme in the three implementations. It is evident from Table 4.1 that the implementation using generalized elemental subroutines is very concise compared to the other two implementations. It is also much more maintainable thanks to its simplicity because it essentially implements the underlying numerical

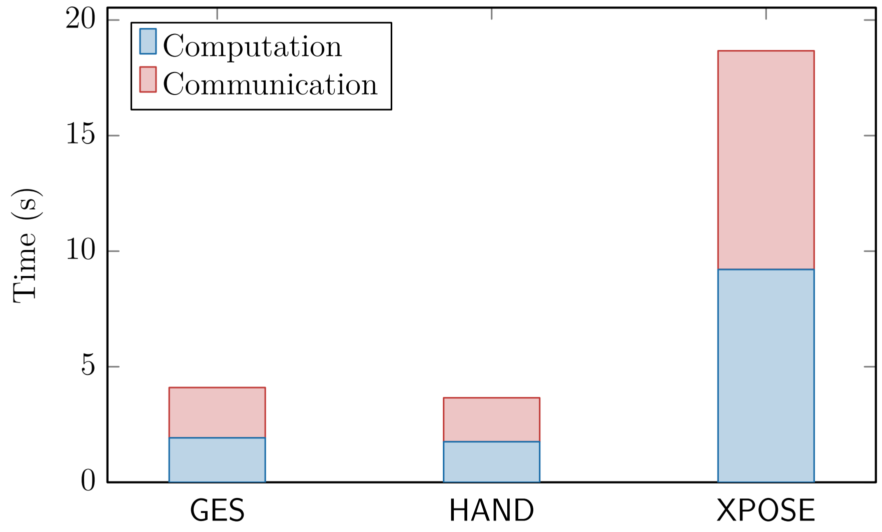


Figure 4.4. Running times of the three-dimensional large eddy simulation-based jet engine noise prediction using the three implementations of the sixth-order compact spatial partial differentiation scheme

Table 4.1.
Lines of code needed by the individual components of the sixth-order compact spatial partial differentiation scheme in the three implementations

	GES	HAND	XPOSE
Differentiation driver	17	10	
Right-hand side stencil	84	433	
SPIKE driver	25	39	
Thomas algorithm	17	123	
SPIKE reduced system	55	299	
SPIKE solution retrieval	13	131	
Transposition method			750
Total	211	1,035	750

algorithms for only single vectors. In comparison, the handwritten code requires almost five times as many lines of code to deliver an extra 12 % of performance.

5 A FUNCTIONAL ARRAY PROGRAMMING MODEL FOR REGULAR GRID-BASED NUMERICAL APPLICATIONS

5.1 Introduction

In Chapter 4, we discussed a programming model based on the idea of generalizing the elemental subroutines found in programming languages traditionally used in scientific computing to simplify the programming tasks of authoring regular grid-based numerical applications. We also demonstrated its practical feasibility through a proof-of-concept implementation in the form of a programming tool which rewrites Fortran source code. As shown in section 4.4, the programming tool is capable of delivering a decent level of performance, generating code which trails the equivalent handwritten code by only about 10 % in performance, while requiring a significantly reduced amount of programming effort. However, when it comes to the pursuit for the highest possible program efficiency, it is desirable that this performance gap between an automated programming tool and manual programming be closed.

There are two areas in which the programming model based on generalized elemental subroutines is arguably imperfect. First, the expressiveness of the input language is excessively broad. Due to the imperative nature of the programming model, it is difficult to constrain the expressiveness of the input language. As implemented by the proof-of-concept programming tool, despite the syntactic restrictions compared to the full Fortran 90 language, it is still possible to specify arbitrary patterns of computation. As a consequence, the code optimization process is relatively conservative with regard to the existing elements of the generalized elemental subroutine. For example, it makes no attempt to adjust the relative order of the existing statements in the subroutines. Second, while the programming model has some built-in knowledge about communication which enables automatic communication aggregation, the programmer still largely has to manually handle many other aspects of communication such as managing communicators and annotating matching pairs of send and receive operations. In this chapter, we present an SPMD-style functional array programming model [65] which attempts to address these problems of generalized elemental subroutines. By being functional and array-based, it has sufficient flexibility to express the computation patterns occurring in many regular grid-based numerical applications, especially LES-based jet engine noise prediction, but also enables more aggressive code optimizations. It also has communication fully integrated into its construction and liberates the programmer from the mundane details such as managing communicators. We describe an accompanying code generation process which translates programs specified using the programming model into imperative code for incremental adoption in existing applications.

5.2 Basic concepts

A regular grid-based numerical application considered by our functional array programming model is defined in a d-dimensional computational space whose dimensions are numbered from 1 to d and denoted by

$$\Delta = \{ \delta_1, \delta_2, \ldots, \delta_d \} . \tag{5.1}$$

These dimensions need not have spatial counterparts in the physical setting of the computational problem which the application targets. Nonspatial dimensions can be defined to distinguish homogeneous attributes associated with the points in the computational space. For example, the different flow variables in a flow field can be represented by a fourth dimension in addition to the three dimensions of the physical space. The computational space is discretized into a grid that is in turn divided into partitions each of which assumes the topology of a regular grid. For each grid partition, we denote its size in each $\delta_k \in \Delta$ by n_k, a symbolic constant whose exact value is to be determined at run-time. A grid partition can be connected to another grid partition at each of its $2d$ faces. It is required that any two neighboring grid partitions, when considered as a whole, must form a larger regular grid. As a consequence, starting from any grid partition, it is possible to traverse in both directions of a dimension and identify a chain of grid partitions which form either a rectangle or a torus. The application assigns each grid partition to a separate processor. Due to this one-to-one correspondence between grid partitions and processors, the topology of the connectivity between the grid partitions is isomorphic to that of the logical grid of processors. Hence, for each processor, it is possible to similarly identify in each dimension a either linear or circular chain of processors to which it belongs. Such chains will serve as the basis for defining communication in the programming model.

A program specified using this programming model consists of a collection of functional procedures. Each procedure uses a tuple of input variables to compute a tuple of output variables and may use a set of local variables to represent some intermediate results. The only available data type is multidimensional arrays of primitive scalar types such as real, integer and boolean. An array A can have no or exactly one dimension which corresponds to each $\delta_k \in \Delta$. As a result, there is no need to distinguish the dimensions of A and those of the computational space. Thus, we denote the set of dimensions of A by $\mathrm{dims}(A) \subseteq \Delta$. The size of A in each $\delta_k \in \Delta$, denoted by $\mathrm{size}_k(A)$, can be either a compile-time integral constant or n_k. If $\mathrm{dims}(A) = \emptyset$, A degenerates into a scalar variable. The semantics of a procedure is defined via definitions of array values by means of arithmetic, communication, procedure invocation and procedure iteration. Since the procedures are functional, each array can have only one definition, and its value cannot be changed once that has been computed.

5.3 Types of array definitions

5.3.1 Definition by arithmetic

The value of an array A can be defined by elemental arithmetic expressions. For this purpose, for each $\delta_k \in \mathrm{dims}(A)$, we allow two nonnegative compile-time integral constants $\mathrm{top}_k(A)$ and $\mathrm{bottom}_k(A)$ satisfying

$$\mathrm{top}_k(A) + \mathrm{bottom}_k(A) \leq \mathrm{size}_k(A) \tag{5.2}$$

to be specified which partition the indices of A in δ_k into

$$\mathrm{idx}_k(A) = \mathrm{idx}_k^{\mathrm{T}}(A) \cup \{\, i_k \,\} \cup \mathrm{idx}_k^{\mathrm{B}}(A) \tag{5.3}$$

where

$$\mathrm{idx}_k^{\mathrm{T}}(A) = \{\, i \mid 1 \leq i \leq \mathrm{top}_k(A) \,\}\,, \tag{5.4}$$

$$\mathrm{idx}_k^{\mathrm{B}}(A) = \{\, i \mid \mathrm{size}_k(A) - \mathrm{bottom}_k(A) + 1 \leq i \leq \mathrm{size}_k(A) \,\}\,, \tag{5.5}$$

and i_k is a symbolic index which represents all indices not in $\mathrm{idx}_k^{\mathrm{T}}(A) \cup \mathrm{idx}_k^{\mathrm{B}}(A)$. This makes it possible to separate the elements located at the array boundaries from those in the interior and use different expressions than that for the latter to define their values. This is a common need which arise from the boundary conditions of the numerical methods of many regular grid-based numerical applications.

For each index

$$\begin{aligned} j^{(A)} &= \langle j_{k_1}^{(A)}, j_{k_2}^{(A)}, \ldots, j_{k_m}^{(A)} \rangle \\ &\in \prod_{\delta_k \in \mathrm{dims}(A)} \mathrm{idx}_k(A), \end{aligned} \tag{5.6}$$

the expression for the element $A[j^{(A)}]$ can contain commonplace mathematical operators and functions. Conditional expressions are supported by incorporating the `MERGE` intrinsic function of Fortran. Operands in an expression can be compile-time constants, symbolic run-time constants and array elements. Symbolic run-time constants are mainly used to convey information whose exact contents can only be determined at run-time such as the location of the processor in the logical grid of processors. If an array element $B[j^{(B)}]$ with

$$\begin{aligned} j^{(B)} &= \langle j^{(B)}_{k_1}, j^{(B)}_{k_2}, \ldots, j^{(B)}_{k_m} \rangle \\ &\in \prod_{\delta_k \in \mathrm{dims}(B)} \mathrm{idx}_k(B), \end{aligned} \tag{5.7}$$

is referenced in the expression, the index component $j^{(B)}_k$ should be a linear combination of $\{\, \mathrm{size}_k(B), j^{(A)}_k, 1 \,\}$ if $\delta_k \in \mathrm{dims}(A)$ or a linear combination of $\{\, \mathrm{size}_k(B), 1 \,\}$ if $\delta_k \notin \mathrm{dims}(A)$.

An array A can also be defined arithmetically via reduction on another array B such that $\mathrm{dims}(A) \subset \mathrm{dims}(B)$. It is required that $\mathrm{size}_k(A) = \mathrm{size}_k(B)$ for all $\delta_k \in \mathrm{dims}(A)$. Then using a commutative and associative reduction operator $\odot$ such as addition and multiplication, each element $A[j^{(A)}]$ is assigned the value

$$A[j^{(A)}] = \bigodot_{j^{(B)}_{k_1}=1}^{\mathrm{size}_{k_1}(B)} \bigodot_{j^{(B)}_{k_2}=1}^{\mathrm{size}_{k_2}(B)} \cdots \bigodot_{j^{(B)}_{k_m}=1}^{\mathrm{size}_{k_m}(B)} B[j^{(B)}] \tag{5.8}$$

where

$$\{\, \delta_{k_1}, \delta_{k_2}, \ldots, \delta_{k_m} \,\} = \mathrm{dims}(B) \setminus \mathrm{dims}(A), \tag{5.9}$$

$$j^{(B)}_k = j^{(A)}_k, \quad \forall \delta_k \in \mathrm{dims}(A). \tag{5.10}$$

Definition by arithmetic is the only type of array definitions that allows circular references among arrays. In general, an array A can directly or indrectly use some of its elements to define the other elements, but circular references at the level of individual array elements are prohibited.

5.3.2 Definition by communication

For our functional array programming model, without imposing extra assumptions on the grid topology, we consider only one-dimensional communication primitives. If further assumptions can be made regarding the grid topology such as existence of embedded higher-dimensional Cartesian topologies, corresponding higher-dimensional communication primitives can also be incorporated.

Two types of one-dimensional communication primitives are provided. The first type is shift. Given a direction aligned with one of the dimensions of the computational space and a hop count, the shift primitive defines a destination array on a remote processor as an exact copy of a source array on the local processor. Both the remote and the local processors belong to the same chain of processors that lies in the dimension in which the shift occurs. A shift can either linear or circular. The linear shift is inapplicable if the chain of processors is circular. In contrast, the circular shift can be applied when the chain of processors is linear by hypothetically joining the two ends of the chain.

The second type of one-dimensional communication primitives is all-to-all reduce. As with shift, all-to-all reduce also needs a direction aligned with a dimension of the computational space. It collapses the values of the source arrays on processors belonging to the same chain that lies in that dimension using a commutative and associative elemental reduction operator and duplicates the result in the destination arrays on the same set of processors.

As a natural constraint, the source arrays and the destination array should share the same dimensions and the same size in each of those dimensions.

5.3.3 Definition by procedure invocation

We allow nonrecursive procedure invocation in our functional array programming model. For a procedure P, we denote its tuples of input and output parameters by

$$\mathrm{in}(P) = \langle I'_1, I'_2, \ldots, I'_{|\mathrm{in}(P)|} \rangle \,, \tag{5.11}$$

$$\mathrm{out}(P) = \langle O'_1, O'_2, \ldots, O'_{|\mathrm{out}(P)|} \rangle \,, \tag{5.12}$$

respectively. We can invoke P with the input $\langle I_1, I_2, \ldots, I_{|\mathrm{in}(P)|}\rangle$ to obtain the output $\langle O_1, O_2, \ldots, O_{|\mathrm{out}(P)|}\rangle$, provided that each input or output argument–parameter pair $\langle I_m, I'_m\rangle$ or $\langle O_l, O'_l\rangle$, if represented commonly by $\langle A, A'\rangle$, satisfies that

$$\mathrm{dims}(A) = \mathrm{dims}(A'), \tag{5.13}$$

$$\mathrm{size}_k(A) = \mathrm{size}_k(A'), \quad \forall \delta_k \in \mathrm{dims}(A). \tag{5.14}$$

5.3.4 Definition by procedure iteration

Procedure iteration is an augmented version of procedure invocation designed for expressing iterative algorithms. Suppose that for a procedure P, $|\mathrm{in}(P)| \geq |\mathrm{out}(P)|$. Procedure iteration defines a recurrence relation

$$\begin{aligned}&\langle O_1^{(\tau)}, O_2^{(\tau)}, \ldots, O_{|\mathrm{out}(P)|}^{(\tau)}\rangle \\ &\quad = \begin{cases}\langle I_1, I_2, \ldots, I_{|\mathrm{out}(P)|}\rangle & \text{if } \tau = 0, \\ P(O_1^{(\tau-1)}, O_2^{(\tau-1)}, \ldots, O_{|\mathrm{out}(P)|}^{(\tau-1)}, I_{|\mathrm{out}(P)|+1}, I_{|\mathrm{out}(P)|+2}, \ldots, I_{|\mathrm{in}(P)|}) & \text{if } \tau \geq 1.\end{cases}\end{aligned} \tag{5.15}$$

The iteration terminates at some $\tau = \tau^*$ such that

$$\langle O_1, O_2, \ldots, O_{|\mathrm{out}(P)|}\rangle = \langle O_1^{(\tau^*)}, O_2^{(\tau^*)}, \ldots, O_{|\mathrm{out}(P)|}^{(\tau^*)}\rangle \tag{5.16}$$

satisfies a stopping criterion specified as a boolean expression.

5.3.5 Examples

Using an expository syntax for our functional array programming model, Listings 5.1 and 5.2 illustrate the implementations of the Thomas algorithm for tridiagonal linear systems and the first-order central difference scheme

$$y_i = \frac{x_{i+1} - x_{i-1}}{2h} \tag{5.17}$$

in a periodic one-dimensional computational space. The "`top:size:bottom`" notation such as "`1:n:0`" in the declaration of an array A specifies the values of $\text{top}_k(A)$, $\text{size}_k(A)$ and $\text{bottom}_k(A)$ for each $\delta_k \in \text{dims}(A)$; if the "`top`" and "`bottom`" parts are omitted, it is implied that $\text{top}_k(A) = \text{bottom}_k(A) = 0$. If the notation is replaced by an "`*`", the corresponding dimension is not in $\text{dims}(A)$. The "`cshift`" keyword indicates an array definition by circular shift with the following signed number in square brackets specifying the shift direction and hop count.

5.4 Extending procedure invocation and procedure iteration to enable application of lower-dimensional algorithms in higher-dimensional contexts

So far, we have not included the capability to apply lower-dimensional algorithms in higher-dimensional contexts in our functional array programming model. We separate its description from that of the basic construction of the programming model so that it enjoys an emphasized discourse. Our strategy for enabling such a functionality is to extend the semantics of procedure invocation and procedure iteration.

5.4.1 Extension of procedure invocation

In order to allow expression application of a lower-dimensional algorithm in a higher-dimensional computational space, we need a mechanism to map the computational space used in defining the former onto the latter and introduce the necessary semantics of repeated invocation. To achieve this goal, we first let each procedure have its own

Listing 5.1. Specification of the Thomas algorithm in the functional array programming model

```
PROCEDURE ThomasFactorize
  INPUT { l[n], d[n], u[n] : real }
  OUTPUT { l_[1:n:0], d_[1:n:0], u_[0:n:1] : real }

  l_[i] = l[i] / d_[i-1]           # 2 <= i <= n
  d_[1] = d[1]
  d_[i] = d[i] - l_[i] * u_[i-1]   # 2 <= i <= n
  u_[i] = u[i]                     # 1 <= i <= n-1


PROCEDURE ThomasSolve
  INPUT { y[n], l[n], d[n], u[n] : real }
  LOCAL { z[1:n:0] : real }
  OUTPUT { x[0:n:1] : real }

  z[1] = y[1]
  z[i] = y[i] - l[i] * z[i-1]              # 2 <= i <= n
  x[i] = (z[i] - u[i] * x[i+1]) / d[i]   # 1 <= i <= n-1
  x[n] = z[n] / d[n]


PROCEDURE Thomas
  INPUT { y[n], l[n], d[n], u[n] : real }
  LOCAL { l_[n], d_[n], u_[n] : real }
  OUTPUT { x[n] : real }

  {l_, d_, u_} = ThomasFactorize(l, d, u)
  {x} = ThomasSolve(y, l_, d_, u_)
```

Listing 5.2. Specification of the periodic first-order central difference scheme in the functional array programming model

```
PROCEDURE PeriodicCentralDifference
  INPUT { x[n], h[*] : real }
  LOCAL { x1[*], xn[*], x0[*], xn1[*] : real }
  OUTPUT { y[1:n:1] : real }

  x1 = x[1]
  xn = x[n]
  x0 = cshift[+1](xn)
  xn1 = cshift[-1](x1)
  y[1] = (x[2] - x0) / (2 * h)
  y[i] = (x[i+1] - x[i-1]) / (2 * h)  # 2 <= i <= n-1
  y[n] = (xn1 - x[n-1]) / (2 * h)
```

computational space instead of assuming that of the entire application. The reference computational space of each array definition then becomes that of its containing procedure. Furthermore, because symbolic run-time constants can be used to convey dimension-related information in array definitions by arithmetic, we allow them to be associated with subsets of the dimensions of the computational space of the procedure so that they can be transformed when the computational spacing mapping takes place. To simplify the notations below, we extend the notations $\mathrm{dims}(\cdot)$ and $\mathrm{size}_k(\cdot)$ to cover any object with associated dimensions and/or dimensions sizes such as procedures and arrays.

Now we are ready to define the desired mechanism which enables application of lower-dimensional algorithms in higher-dimensional contexts. Suppose that a procedure P invokes another procedure P' with the input and output arguments $\langle I_1, I_2, \ldots, I_{|\mathrm{in}(P)|}\rangle$ and $\langle O_1, O_2, \ldots, O_{|\mathrm{out}(P)|}\rangle$. Let the corresponding input and output parameters of P' be

$$\mathrm{in}(P') = \langle I'_1, I'_2, \ldots, I'_{|\mathrm{in}(P')|}\rangle\,, \tag{5.18}$$

$$\mathrm{out}(P') = \langle O'_1, O'_2, \ldots, O'_{|\mathrm{out}(P')|}\rangle\,, \tag{5.19}$$

respectively. When we do not distinguish their purposes, we use

$$\langle A_1, A_2, \ldots, A_{|\mathrm{in}(P')|+|\mathrm{out}(P')|}\rangle$$

and

$$\langle A'_1, A'_2, \ldots, A'_{|\mathrm{in}(P')|+|\mathrm{out}(P')|}\rangle$$

to refer to all the arguments and parameters, respectively. To map the computational space of P' onto that of P, an injective dimension map

$$f\colon\ \mathrm{dims}(P') \to \mathrm{dims}(P)$$

is specified, which implicitly requires that $|\text{dims}(P)| \geq |\text{dims}(P')|$. The function f effectively partitions $\text{dims}(P)$ into two parts. The first part, $f(\text{dims}(P'))$, is used for computational space mapping. Each $\delta_k \in f(\text{dims}(P'))$ will be used as the corresponding $\delta'_{k'} = f^{-1}(\delta_k)$ during the procedure invocation. Meanwhile, the second part, $\text{dims}(P) \setminus f(\text{dims}(P'))$, is used to introduce repeated invocation.

We impose the following constraints on the arguments and parameters to ensure that the procedure invocation has well-defined semantics:

- For each pair $\langle A_m, A'_m \rangle$,
    - $f^{-1}(\text{dims}(A_m)) = \text{dims}(A'_m)$;
    - for all $\delta'_{k'} \in \text{dims}(A'_m)$ and $\delta_k = f(\delta'_{k'})$, $\text{size}_k(A_m) = \text{size}_{k'}(A'_m)$ assuming that n_k and $n'_{k'}$, the respective sizes of the corresponding grid partitions in δ_k and $\delta'_{k'}$, are equal.
- For each pair $\langle I_m, O_l \rangle$, if O'_l directly or indirectly depends on I'_m in P', then $\text{dims}(O_l) \setminus f(\text{dims}(P')) \supseteq \text{dims}(I_m) \setminus f(\text{dims}(P'))$.
- For each $\delta_k \in \text{dims}(P) \setminus f(\text{dims}(P'))$, $\text{size}_k(A_m)$ is the same for all A_m such that $\delta_k \in \text{dims}(A_m)$.

Informally, the first constraint ensures that when each A_m is used as the corresponding A'_m during the procedure invocation, the sizes of its dimensions in $f(\text{dims}(P'))$ match those of A'_m in P'. The second constraint ensures that each output parameter has the necessary extra dimensions to accommodate all different values that will be generated by the repeated invocation. The last constraint ensures that for each dimension not participating in computational space mapping, all the arguments are either scalars or have a consistent size in that dimension.

Taking advantage of the above constraints, a Cartesian index range R can be established such that

$$\text{dims}(R) = \bigcup_m \text{dims}(A_m) \setminus f(\text{dims}(P')). \tag{5.20}$$

The index range R is bounded in each $\delta_k \in \text{dims}(R)$ by $\text{size}_k(A_m)$ for any A_m such that $\delta_k \in \text{dims}(A_m)$. Each index in R identifies, with the inapplicable components ignored, a lower-dimensional slice lying in $f(\text{dims}(P'))$ of each A_m. The lower-dimensional slices of all the arguments identified by the same index can then be used in an invocation of P' subject to computational space mapping. Such invocations are repeated for all the indices in R. Each output parameter O_l such that $\text{dims}(O_l) \subset \text{dims}(R)$ will be defined multiple times due to repetition over the dimensions in $\text{dims}(R) \setminus \text{dims}(O_l)$. However, due to the functional nature of the programming model, the aforementioned constraints guarantee that these definitions are identical duplicates and can be trivially consolidated into a single definition.

5.4.2 Extension of procedure iteration

Since procedure iteration is an augmented version of procedure invocation, it can receive the same extension as the latter. Compared to procedure invocation, procedure iteration has an stopping criterion additionally. After the semantics of repetition is introduced, the stopping criterion can evaluate to different values in the same iteration depending on which lower-dimensional slices of the arguments are used in the embedded procedure invocation. Differing values of the stopping criterion indicate that some instances of repetition can exit the iterative process while the rest must continue. There are several possible strategies to handle such discrepancy. One may choose to terminate each individual instance of repetition as soon as it satisfies the stopping criterion. This minimizes the total amount of computation but requires a mechanism to consolidate the multiple definitions of a single output parameter resulting from different numbers of iterations (e.g., keeping all of them or selecting just one according to a certain rule). Alternatively, one may also choose to let all instances of repetition continue until they all satisfy the stopping criterion. This imposes an additional assumption on the iterative process that the iteration is convergent, i.e., once the stopping criterion is satisfied, it will remain satisfied even if the iteration continues. Consolidation of duplicate array definitions is trivial in this case because all instances of repetition execute the same number of iterations. We adopt the latter approach because it offers regularity in the iterated procedure invocation, which can potentially enable more aggressive optimizations during code generation.

5.4.3 Examples

Listing 5.3 illustrates an application of the procedure `Thomas` in Listing 5.1 in a three-dimensional computational space. The number “`3`” following the procedure name “`Thomas`” indicates that the only dimension of the computational space of procedure `Thomas` is mapped the third dimension of that of procedure `ThomasIn3D`. Since the array `x` and `y` are not scalars in the first two dimensions, the Thomas algorithm is repeated on $n_1 n_2$ right-hand side vectors stored along the third dimension of `y` to compute the corresponding solution vectors in `x`. Listing 5.4 is a similar example for the first-order central difference scheme implemented in Listing 5.2.

5.5 Code generation and optimization

A consequence of our array programming model being functional is that it fully specifies what needs to be computed but leaves out most details about how the computation is to be performed. An implementation of the programming model is free to schedule the operations specified by procedures and choose the data structures for storing the operation results. The code generation and optimization strategies described in this section exploit the features and assumptions of the programming model to automatically carry out code transformations which a programmer equipped with architectural knowledge about the target computing platform would otherwise manually perform.

Listing 5.3. Application of the Thomas algorithm specified in the functional array programming model in a three-dimensional computational space

```
PROCEDURE ThomasIn3D
  INPUT { y[n1, n2, n3], l[*, *, n3],
          d[*, *, n3], u[*, *, n3] : real }
  OUTPUT { x[n1, n2, n3] : real }

  {x} = Thomas[3](y, l, d, u)
```

Listing 5.4. Application of the first-order central difference scheme specified in the functional array programming model in a three-dimensional computational space

```
PROCEDURE PeriodicCentralDifferenceIn3D
  INPUT { x[n1, n2, n3] : real }
  OUTPUT { y[n1, n2, n3] : real }

  {y} = PeriodicCentralDifference[3](x)
```

5.5.1 Dimensional procedure rewriting

Section 5.4 extends procedure invocation and procedure iteration the our functional array programming model with the semantics of repeated invocation. A naïve strategy strategy for implementing the repetition is to literally iterate over the Cartesian index range defined in section 5.4.1 and invoke the procedure once per iteration. While very simple, its implications on program efficiency, in terms of both computation and communication, can be disastrous. Consider the examples in Listings 5.3 and 5.4 assuming that the array dimensions are in a column-major order as they appear in the listings. For the Thomas algorithm, a skilled programmer would first avoid the unnecessary repeated invocation of the procedure `ThomasFactorize` in Listing 5.1. He/she would then devise a three-level loop nest which batches the right-hand side vectors by the first dimensions of the arrays in the innermost loops to enable SIMD parallelism. He/she would also counterintuitively iterate over the second dimensions of the arrays in the outermost loop in order to exploit the data locality of the forward and backward sweeps of the algorithm. For the first-order central difference scheme, the programmer would aggregate the two circular shifts in Listing 5.2 to reduce the total communication initiation cost. Naïve repetition is unable to take advantage of any of these optimization opportunities.

Dimensional procedure rewriting is a process which creates the possibility to apply all of the above code optimizations by factoring the semantics of repeated invocation into the specifications of the invoked procedures. Suppose that a procedure P invokes another procedure P' with a dimension map f. The rewriting process first lets P' adopt the computational space of P by renaming the members of $\mathrm{dims}(P')$ according to f and adding the dimensions in $\mathrm{dims}(P) \setminus f(\mathrm{dims}(P'))$, after which $\mathrm{dims}(P')$ becomes equivalent to $\mathrm{dims}(P)$, and f can be replaced by an identify map and is ignored hereafter. Accordingly, each parameter A'_m of P' adopts the dimensions and sizes of the corresponding argument A_m in P. For each local array L of P' and each dimension δ_k newly added to $\mathrm{dims}(P')$, if L directly or indirectly depends on an input parameter

I'_m such that $\delta_k \in \text{dims}(I'_m)$, δ_k is also added to L with $\text{size}_k(L) = \text{size}_k(I'_m)$. After all the arrays in P' have been transformed, their definitions are modified accordingly to reflect the semantics of repeated invocation.

Listings 5.5 and 5.6 show the results of rewriting the examples in Listings 5.1, 5.2, 5.3 and 5.4. Notice that the rewriting process already avoids repeated invocation of procedure `ThomasFactorize` and aggregates the two circular shifts in procedure `PeriodicCentralDifference`.

5.5.2 Computation–communication interleaving

After a procedure is rewritten, the semantics of repeated invocation introduced by the extended procedure invocation and procedure iteration becomes explicit in the body of the procedure. The operations specified within are then ready to be scheduled. Scheduling occurs in two stages. The first stage is computation–communication interleaving, which creates opportunities to overlap the two types of operations.

When we consider computation–communication interleaving, we regard individual arrays as the smallest scheduling unit. There are also groups of arrays which must each be considered as an atomic whole. These include arrays defined by arithmetic which circularly reference one another in their definitions and the output parameters which appear in the same procedure invocation or procedure iteration. The inseparability of the latter is obvious, while that of the former is due to the fact that the elements of those arrays need to be evaluated in an intertwined order. We represent arrays which can be scheduled separately by singleton groups and use the term *array group* as the unified terminology for the scheduling units in computation–communication interleaving.

The dependence relation between array groups of a procedure defines a directed dependence graph $G(V, E)$. Because there cannot be circular references between array groups, G is acyclic. The goal of computation–communication interleaving is to determine a topological ordering of V optimized for computation–communication

Listing 5.5. Rewritten specification of the Thomas algorithm in the functional array programming model

```
PROCEDURE ThomasFactorize
  INPUT { l[*, *, n3], d[*, *, n3], u[*, *, n3] : real }
  OUTPUT { l_[*, *, 1:n3:0], d_[*, *, 1:n3:0],
           u_[*, *, 0:n3:1] : real }

  l_[i3] = l[i3] / d_[i3-1]
  d_[1] = d[1]
  d_[i3] = d[i3] - l_[i3] * u_[i3-1]
  u_[i3] = u[i3]


PROCEDURE ThomasSolve
  INPUT { y[n1, n2, n3], l[*, *, n3], d[*, *, n3],
          u[*, *, n3] : real }
  LOCAL { z[n1, n2, 1:n3:0] : real }
  OUTPUT { x[n1, n2, 0:n3:1] : real }

  z[i1, i2, 1] = y[i1, i2, 1]
  z[i1, i2, i3] = y[i1, i2, i3] - l[i3] * z[i1, i2, i3-1]
  x[i1, i2, i3] = (z[i1, i2, i3] - u[i3] * x[i1, i2, i3+1]) / d[i3]
  x[i1, i2, n3] = z[i1, i2, n3] / d[n3]


PROCEDURE Thomas
  INPUT { y[n1, n2, n3], l[*, *, n3],
          d[*, *, n3], u[*, *, n3] : real }
  LOCAL { l_[*, *, n3], d_[*, *, n3], u_[*, *, n3] : real }
  OUTPUT { x[n1, n2, n3] : real }

  {l_, d_, u_} = ThomasFactorize(l, d, u)
  {x} = ThomasSolve(y, l_, d_, u_)


PROCEDURE ThomasIn3D
  INPUT { y[n1, n2, n3], l[*, *, n3],
          d[*, *, n3], u[*, *, n3] : real }
  OUTPUT { x[n1, n2, n3] : real }

  {x} = Thomas(y, l, d, u)
```

Listing 5.6. Rewritten specification of the first-order central difference scheme in the functional array programming model

```
PROCEDURE PeriodicCentralDifference
  INPUT { x[n1, n2, n3], h[*, *, *] : real }
  LOCAL { x1[n1, n2, *], xn[n1, n2, *],
          x0[n1, n2, *], xn1[n1, n2, *] : real }
  OUTPUT { y[n1, n2, 1:n3:1] : real }

  x1[i1, i2] = x[i1, i2, 1]
  xn[i1, i2] = x[i1, i2, n3]
  x0 = cshift[*, *, +1](xn)
  xn1 = cshift[*, *, -1](x1)
  y[i1, i2, 1] = (x[i1, i2, 2] - x0[i1, i2]) / (2 * h)
  y[i1, i2, i3] = (x[i1, i2, i3+1] - x[i1, i2, i3-1]) / (2 * h)
  y[i1, i2, n3] = (xn1[i1, i2] - x[i1, i2, n3-1]) / (2 * h)


PROCEDURE PeriodicCentralDifferenceIn3D
  INPUT { x[n1, n2, n3] : real }
  OUTPUT { y[n1, n2, n3] : real }

  {y} = PeriodicCentralDifference(x)
```

overlapping. A greedy strategy is to initiate each communication operation as early as possible and complete the operation as late as possible. This maximizes the time between the start and the end of the operation and allows more computation to be performed in parallel during that time. However, this strategy works against itself when a communication operation depends on another one, because delaying completion of the latter also delays initiation of the former. Therefore, instead of adopting this simplistic strategy, we solve the computation–communication interleaving problem as a formal optimization problem.

In preparation for formulating the optimization problem, we model the target computing platform in an idealized fashion where computation is strictly sequential, while communication has unlimited parallelism. For estimating the cost of each computation or communication operation, we assume that an implementation of the programming model has built-in knowledge about the performance characteristics of the target computing platform. Application-specific information such as the average dimensions of the grid partitions and the expected numbers of iterations needed by each procedure iteration is requested from the programmer, who in turn may determine it from practical needs and empirical profiling.

We formulate the optimization problem using mixed integer linear programming (MILP). For each array group $\Gamma \in V$, we represent the start time of its computation or communication by a nonnegative continuous variable t_Γ. We use a continuous variable t to represent the end time of execution of the procedure. The objective of the MILP problem is to minimize t. The problem contains the following constraints:

Dependence: For each $\langle \Gamma_1, \Gamma_2 \rangle \in E$,

$$t_{\Gamma_2} \geq t_{\Gamma_1} + c_{\Gamma_1} \tag{5.21}$$

where c_{Γ_1} is the computation or communication cost of Γ_1.

Computational sequentiality: For each $\{\Gamma_1, \Gamma_2\} \subseteq V$ such that Γ_1 and Γ_2 are mutually unreachable in G,

$$t_{\Gamma_1} \geq t_{\Gamma_2} + \mathbb{1}_{\mathrm{comp}}(\Gamma_2) c_{\Gamma_2} \vee t_{\Gamma_2} \geq t_{\Gamma_1} + \mathbb{1}_{\mathrm{comp}}(\Gamma_1) c_{\Gamma_1} \tag{5.22}$$

where $\mathbb{1}_{\mathrm{comp}}(\cdot)$ is a binary indicator function of whether an array group is defined by computation as opposed to communication. To conform to the restrictions of MILP, this disjunctive form is expressed as two separate constraints:

$$t_{\Gamma_1} \geq t_{\Gamma_2} + \mathbb{1}_{\mathrm{comp}}(\Gamma_2) c_{\Gamma_2} + b_{\Gamma_1,\Gamma_2} M, \tag{5.23}$$

$$t_{\Gamma_2} \geq t_{\Gamma_1} + \mathbb{1}_{\mathrm{comp}}(\Gamma_1) c_{\Gamma_1} + \bar{b}_{\Gamma_1,\Gamma_2} M \tag{5.24}$$

where b_{Γ_1,Γ_2} is a binary variable, $\bar{b}_{\Gamma_1,\Gamma_2} = 1 - b_{\Gamma_1,\Gamma_2}$, and M is a sufficiently large constant such as

$$M = \sum_{\Gamma \in V} c_{\Gamma}. \tag{5.25}$$

End of procedure execution: For each $\Gamma \in V$,

$$t \geq t_{\Gamma} + c_{\Gamma}. \tag{5.26}$$

We sort the array groups $\Gamma \in V$ in increasing order of the values of t_Γ in the solution to the MILP problem, breaking ties by prioritizing communication over computation. The result is a rough sketch of the code to be generated for the procedure. Only at this point do we apply the aforementioned greedy strategy and a communication completion operation before the first use of each array defined by communication.

5.5.3 Computation scheduling

The second stage of computation–communication scheduling is computation scheduling. In the code sketch produced by computation–communication interleaving, the only missing details are in which order element values of arrays defined by arithmetic are evaluated. Deriving evaluation schemes for those arrays is the goal of computation scheduling.

In computation scheduling, we identify each consecutive chunk of array groups defined by arithmetic in the code sketch and consider each such chunk separately. For the arrays in a chunk, we can derive a dependence relation between their individual elements. The formulation of array definitions by arithmetic guarantees that the dependence relation is affine. Generating code from affine dependence relations is prior art [6, 13, 30, 33]. Therefore, we avoid reinventing the wheel here and assume that an implementation of the programming model will reuse existing methodologies and software.

After computation scheduling, the code sketch is functionally complete and can be mechanically translated into forms accepted by normal compilation workflows such as source code written in a high-level programming language.

5.5.4 Array dimension elimination

During dimensional procedure rewriting, new dimensions are added to the local arrays of procedures to accommodate all different values generated during repeated invocation. These dimensions are added using a conservative strategy which considers only the dependence between arrays. However, depending on the exact contents of the code sketches produced by computation–communication scheduling, some of these dimensions may in fact be redundant. For instance, consider the Fortran code in Listing 5.7 for the procedure `ThomasSolve` in Listing 5.5. It is obvious that the second dimension of `z` is redundant as the values distinguished by that dimension do not have overlapping lifetimes. It is not possible for dimensional procedure rewriting to omit the redundant dimensions in foresight because it lacks the information to justify such omission, which is not available until after computation scheduling has completed.

Listing 5.7. Fortran code for the procedure `ThomasSolve` in Listing 5.5

```
SUBROUTINE ThomasSolve(y, l, d, u, x)

  REAL, INTENT(IN)  :: y(n1, n2, n3), l(n3), d(n3), u(n3)
  REAL, INTENT(OUT) :: x(n1, n2, n3)

  REAL              :: z(n1, n2, n3)
  INTEGER           :: i1, i2, i3

  DO i2 = 1, n2
    DO i1 = 1, n1
      z(i1, i2, 1) = y(i1, i2, 1)
    END DO
    DO i3 = 2, n3
      DO i1 = 1, n1
        z(i1, i2, i3) = y(i1, i2, i3) - l(i3) * z(i1, i2, i3-1)
      END DO
    END DO
    DO i1 = 1, n1
      x(i1, i2, n3) = z(i1, i2, n3) / d(n3)
    END DO
    DO i3 = n3 - 1, 1, -1
      DO i1 = 1, n1
        x(i1, i2, i3) = &
          (z(i1, i2, i3) - u(i3) * x(i1, i2, i3+1)) / d(i3)
      END DO
    END DO
  END DO

END SUBROUTINE
```

Array dimension elimination locates and removes the aforementioned redundant dimensions from the local arrays of a procedure. The process is based on the live range analysis of individual array elements using the polyhedral model. It first labels all the statements in the code sketch of the procedure with unique symbolic timestamps. Listing 5.8 and Figure 5.1 illustrate the labeling method. After the code sketch is converted into an abstract syntax tree (AST), each node of the AST which is not the root is labeled with its position among its siblings. If a node represents a loop, its label has a second component which is the loop variable. The timestamp of a statement is the concatenation of the labels of the corresponding node in the AST and its nonroot ancestors in top-down order, zero-padded so that the timestamps of all statements have the same length. For Listing 5.8, the timestamps of the statements are

$$\begin{aligned} S_1 &\colon \ \langle 0,0,0,0,0\rangle\,, \\ S_2 &\colon \ \langle 1,i_2,0,i_1,0\rangle\,, \\ S_3 &\colon \ \langle 1,i_2,1,0,0\rangle\,, \end{aligned}$$

respectively, where $1 \leq i_1 \leq n_1$, and $1 \leq i_2 \leq n_2$. Assuming without loss of generality that the loop step sizes are positive, the lexicographical order of the timestamps is exactly the same as the dynamic execution order of the statements. Thus, these timestamps can be used to depict precisely the definition and use histories of individual array elements. Let the space of timestamps be T. For any array A, we denote its index range by $\mathrm{idx}(A)$ and represent its elementwise definition and use histories using two relations $\mathrm{def}(A), \mathrm{use}(A) \subseteq \mathrm{idx}(A) \times T$.

For each subset $\Delta \subseteq \mathrm{dims}(A)$, we want to determine whether its members are redundant as a whole. Consider two indices $j^{(1)}, j^{(2)} \in \mathrm{idx}(A)$ which differ in at least one component which corresponds to a dimension in Δ but are equal in all components that correspond to dimensions not in Δ. If the dimensions in Δ are removed from A, the elements $A[j^{(1)}]$ and $A[j^{(2)}]$ will be aliased to each other. Hence, the removal is legal iff their live ranges do not overlap. Due to the functional nature of our array

Listing 5.8. A code sketch represented using the Fortran syntax

```
S1
DO i2 = 1, n2
  DO i1 = 1, n1
    S2
  END DO
  S3
END DO
```

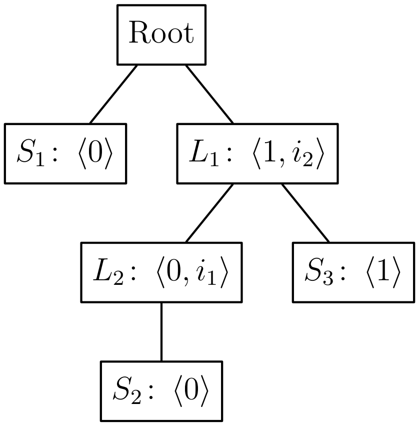

Root
$S_1 : \langle 0 \rangle$
$L_1 : \langle 1, i_2 \rangle$
$L_2 : \langle 0, i_1 \rangle$
$S_3 : \langle 1 \rangle$
$S_2 : \langle 0 \rangle$

Figure 5.1. Abstract syntax tree of the code sketch in Listing 5.8

programming model, each array element can have only one definition, and all its uses depend on that definition. Therefore, overlapping live ranges can be detected by determining the existence of three timestamps $t_1 \leq t_2 \leq t_3$ in lexicographical order such that

$$\langle j^{(1)}, t_1 \rangle, \langle j^{(2)}, t_2 \rangle \in \mathrm{def}(A), \tag{5.28}$$

$$\langle j^{(1)}, t_3 \rangle \in \mathrm{use}(A). \tag{5.29}$$

Since $\mathrm{dims}(A)$ has a finite number of members, the maximal Δ can be via enumeration, and we denote that by $\mathrm{elim}(A)$. For input and output arrays, we let $\mathrm{elim}(A) = \emptyset$.

Listing 5.9 shows the result of applying array dimension elimination to the Fortran code for procedure `ThomasSolve` in Listing 5.7. Notice that the second dimension of array `z` has been removed. It is also the only dimension whose removal is legal.

5.5.5 Array coalescing

Consider the subroutine `ThomasSolve` in Listing 5.7 again. Array dimension elimination is able to remove the second dimension of array `z` as a programmer would do manually. However, the resulting code is still suboptimal. There is no need to use three separate arrays `x`, `y` and `z` because they can be coalesced into a single array, which makes the implementation in-place and shrinks its memory footprint by a factor of three. The need for such coalescing is even more pressing when the effective semantics of a procedure modifies an array only partially as in cases such as the boundary conditions in numerical methods. Out-of-place implementations will the unchanged elements to be copied and end up incurring unnecessary but significant overhead. Array coalescing seeks to mitigate situations like these.

Detection of coalescible arrays uses an array element live range analysis similar to that in array dimension elimination. Two arrays A_1 and A_2 with the same dimensions and sizes can be coalesced iff for each index $j \in \mathrm{idx}(A_1) = \mathrm{idx}(A_2)$, there do not exist

Listing 5.9. Fortran code for the procedure `ThomasSolve` in Listing 5.5 after array dimension elimination

```fortran
SUBROUTINE ThomasSolve(y, l, d, u, x)

  REAL, INTENT(IN)  :: y(n1, n2, n3), l(n3), d(n3), u(n3)
  REAL, INTENT(OUT) :: x(n1, n2, n3)

  REAL              :: z(n1, n3)
  INTEGER           :: i1, i2, i3

  DO i2 = 1, n2
    DO i1 = 1, n1
      z(i1, 1) = y(i1, i2, 1)
    END DO
    DO i3 = 2, n3
      DO i1 = 1, n1
        z(i1, i3) = y(i1, i2, i3) - l(i3) * z(i1, i3-1)
      END DO
    END DO
    DO i1 = 1, n1
      x(i1, i2, n3) = z(i1, n3) / d(n3)
    END DO
    DO i3 = n3 - 1, 1, -1
      DO i1 = 1, n1
        x(i1, i2, i3) = &
          (z(i1, i3) - u(i3) * x(i1, i2, i3+1)) / d(i3)
      END DO
    END DO
  END DO

END SUBROUTINE
```

three timestamps $t_1 \leq t_2 \leq t_3$ in lexicographical order such that the live ranges of the elements $A_1[j]$ and $A_2[j]$ overlap, or, put mathematically,

$$\langle j, t_1 \rangle \in \mathrm{def}(A_1), \tag{5.30}$$

$$\langle j, t_2 \rangle \in \mathrm{def}(A_2), \tag{5.31}$$

$$\langle j, t_3 \rangle \in \mathrm{use}(A_1). \tag{5.32}$$

However, applying this criterion to Listing 5.9 will miss the opportunities to coalesce array `z` with arrays `x` and `y` because array dimension elimination has altered the shape of `z`. Therefore, we also need to consider partial elimination of the redundant dimensions from the local arrays. For this purpose, we define the concept of a *slot* to represent the potential storage space for one or more arrays. Arrays sharing the same slot are coalesced. For each array A, we create a slot S for every subset $\Delta \subseteq \mathrm{elim}(A)$ such that $\mathrm{dims}(S) = \mathrm{dims}(A) \setminus \Delta$, and $\mathrm{size}_k(S) = \mathrm{size}_k(A)$ for all $\delta_k \in \mathrm{dims}(S)$. An array A can be assigned to a slot S if there exists a viable dimension-eliminated version of A which has the dimensions and sizes as S. But we prevent assignment of arrays reducible to scalar variables to nonscalar slots because scalar variables are better handled by traditional compiler optimizations. Assuming that application-specific information has been obtained from the programmer to calculate the sizes of the slots, our goal is to minimize the total size of the occupied slots.

As with computation–communication interleaving, we formulate the minimization problem using MILP. For each slot S, we create denote the candidate arrays which can be assigned to it by $\mathrm{cand}(S)$. For each array–slot pair $\langle A, S \rangle$, we represent whether A is assigned to S by a binary variable $b_{A,S}$. For each slot S, we represented whether it is occupied by a continuous variable t_S. The objective of the MILP problem is to minimize

$$\sum_S c_S t_S$$

where

$$c_S = \prod_{\delta_k \in \mathrm{dims}(S)} \mathrm{size}_k(S) \tag{5.33}$$

is the size of a slot S. The problem contains the following constraints:

Array–slot assignment: For each array A,

$$\sum_S b_{A,S} = 1, \tag{5.34}$$

and for each slot S such that $A \notin \mathrm{cand}(S)$,

$$b_{A,S} = 0. \tag{5.35}$$

Coalescing restriction: For each array pair $\langle A_1, A_2 \rangle$ which cannot be assigned to a slot S simultaneously,

$$b_{A_1,S} + b_{A_2,S} \leq 1. \tag{5.36}$$

For each input array I and each local array L such that $I, L \in \mathrm{cand}(S)$ for a slot S,

$$b_{I,S} + b_{L,S} = 2 \rightarrow \exists O\colon b_{O,S} = 1 \tag{5.37}$$

where O is an output array. This prevents coalescing between coalescing between an input array and a local array unless the former is already coalesced with one or more output arrays. This constraint guarantees that the involved input arrays can be safely overwritten. In an MILP-conformant form, this constraint is expressed as

$$b_{I,S} + b_{L,S} \leq 2 \sum_O b_{O,S} + 1. \tag{5.38}$$

Slot occupancy: For each array–slot pair $\langle A, S \rangle$,

$$t_S \geq b_{A,S}. \tag{5.39}$$

Coalescing an input array with an output array converts an out-of-place procedure into an in-place procedure. A practical issue arises concerning the actual benefit of such conversion in the interaction between the invoker and invokee procedures. Suppose that a procedure P invokes another procedure P' with input and output arguments I and O whose counterparts in P' are I' and O'. Array coalescing is beneficial only when both pairs of input–output arrays are coalescible because when only one pair is coalesced, the uncoalesced pair will force an array copy. Such a situation warrants considering the array coalescing of P and P' jointly. But this risks inflating the size of the MILP problem, which has no known polynomial-time algorithms. Therefore, we use a two-pass approximation. First, we compute a tentative array coalescing of P assuming that I' and O' can be coalesced in P'. If I and O are not tentatively coalesced, we disallow coalescing of I' and O' in P'. Next, we compute the array coalescing of P' recursively. Last, we compute the definitive array coalescing of P, disallowing coalescing of I and O if I' and O' are not coalesced in P'.

After the array coalescing of a procedure has been computed, the procedure is modified to include the necessary changes. In particular, assignments which have become in-place copies are removed.

Listing 5.10 shows the Fortran code for the procedure `ThomasSolve` in Listing 5.5 after both array dimension elimination and array coalescing have been applied. It has the smallest possible memory footprint for the algorithm and is also computationally efficient.

5.6 Empirical evaluation

5.6.1 Prototype implementation

We evaluate our functional array programming model and the accompanying code generation and optimization strategies using a prototype programming tool written in Python. The input syntax is similar to that of the listings in sections 5.3, 5.4 and 5.5. The output language is Fortran, and MPI is used for communication. For

Listing 5.10. Fortran code for the procedure `ThomasSolve` in Listing 5.5 after array dimension elimination and array coalescing

```fortran
SUBROUTINE ThomasSolve(x, l, d, u)

  REAL, INTENT(INOUT) :: x(n1, n2, n3)
  REAL, INTENT(IN)    :: l(n3), d(n3), u(n3)

  INTEGER             :: i1, i2, i3

  DO i2 = 1, n2
    DO i3 = 2, n3
      DO i1 = 1, n1
        x(i1, i2, i3) = x(i1, i2, i3) - l(i3) * x(i1, i2, i3-1)
      END DO
    END DO
    DO i1 = 1, n1
      x(i1, i2, n3) = x(i1, i2, n3) / d(n3)
    END DO
    DO i3 = n3 - 1, 1, -1
      DO i1 = 1, n1
        x(i1, i2, i3) = &
          (x(i1, i2, i3) - u(i3) * x(i1, i2, i3+1)) / d(i3)
      END DO
    END DO
  END DO

END SUBROUTINE
```

solving the MILP problems of computation–communication interleaving and array coalescing, we use the GNU Linear Programming Kit (GLPK) [22] and COIN-OR Branch-and-Cut (CBC) [16] solvers, respectively. We use two different solvers because we found cases where CBC produces invalid solutions for computation–communication interleaving, and GLPK is much too inefficient for array coalescing. To represent the affine dependence relations between array elements and their definition and use histories and detect overlapping lifetimes of element values in computation scheduling, array dimension elimination and array coalescing, we use the Integer Set Library (ISL) [80]. Code generation from affine dependence relations in computation scheduling is accomplished by combining ISL and CLooG [7].

As an implementation detail in computation scheduling, we use ISL to derive the scattering functions required by CLooG. To derive the scattering functions, ISL uses not only an affine dependence relation but also a proximity relation. It attempts to minimize the dependence distance over the latter while respecting the former. We specify the proximity relation as the union of the affine dependence relation and the linear element order of the first two dimensions of all involved arrays. Experiments show that restricting the element order component of the proximity relation to just the first two dimensions of the arrays leads to more efficient code than other choices such as expanding the coverage to all dimensions.

5.6.2 Benchmarks for experimental evaluation

We use the following five benchmarks to demonstrate the effect of the code generation and optimization strategies presented in section 5.5:

conpq: Pointwise primitive-to-conservative conversion of the flow variables in three-dimensional LES of jet engine noise.

deriv: Application of the sixth-order compact spatial partial differentiation scheme along all three directions of a three-dimensional Cartesian grid. The truncated SPIKE algorithm is chosen as the embedded tridiagonal linear system solvers.

matmul: Matrix multiplication expressed as an dimension-expanded version of the vector dot product.

thomas: Application of the Thomas algorithm along all three directions of a three-dimensional Cartesian grid.

interlv: A synthetic benchmark designed for demonstrating the effect of computation–communication interleaving.

We run the benchmarks on Carter, an InfiniBand-interconnected dual-Intel Sandy Bridge cluster hosted at the Rosen Center for Advanced Computing (RCAC) at Purdue University. Among them, conpq, matmul and thomas are run sequentially, whereas deriv and interlv are each run on four nodes using 64 processor cores and MPI processes. For each benchmark, we measure the running times using ten evenly spaced grid partition sizes. The grid partitions are always cube-shaped. The grid partition sizes, which we refer to as n below, range from 24 to 240 for conpq, deriv and thomas, from 100 to 1,000 for matmul and from 10 to 100 for interlv.

5.6.3 Effect of dimensional procedure rewriting

In Figures 5.2, 5.3, 5.4 and 5.5, we plot the running times of the benchmarks conpq, deriv, matmul and thomas with dimensional procedure rewriting disabled. The data are normalized to those of running the benchmarks with dimensional procedure rewriting enabled. Disabling the rewriting processes is equivalent to the situation where the programmer expresses the semantics of his/her application in the most concise possible form using programming tools without dedicated support for regular grid-based numerical applications.

Except for conpq, our programming tool achieves significant speedup. Existing programming tools are sufficient for conpq because its minimum implementation essentially consists of straight-line scalar code, which is easy for the Fortran compiler to inline and transform. Meanwhile, the other three benchmarks contain loops

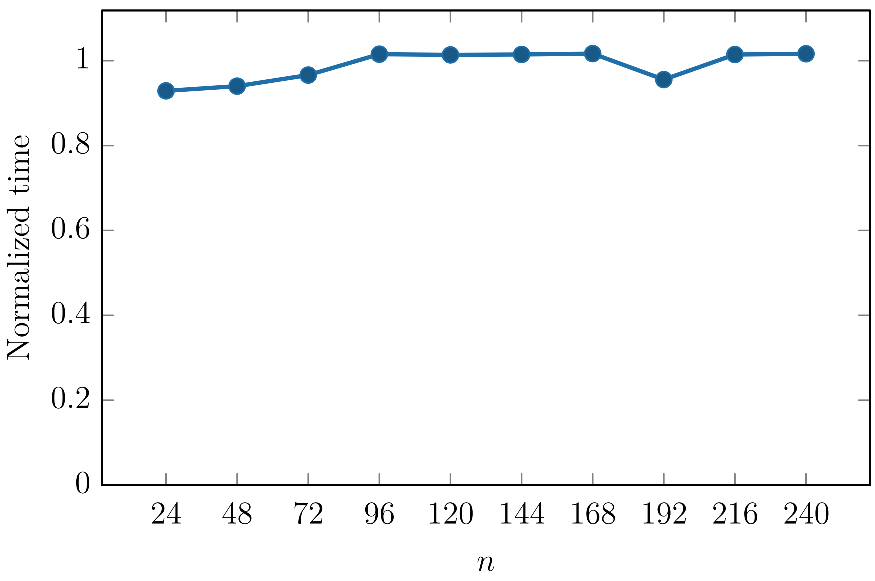


Figure 5.2. Normalized running times of benchmark `conpq` with dimensional procedure rewriting disabled

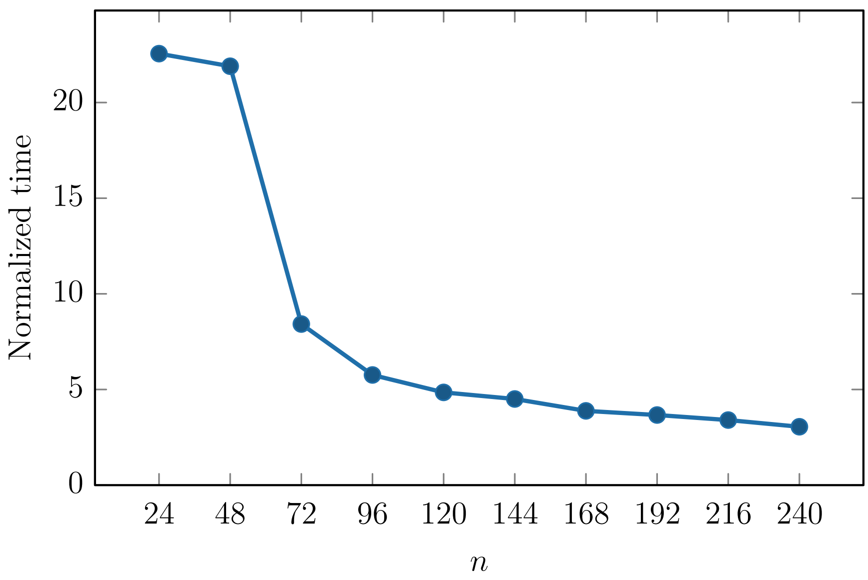


Figure 5.3. Normalized running times of benchmark `deriv` with dimensional procedure rewriting disabled

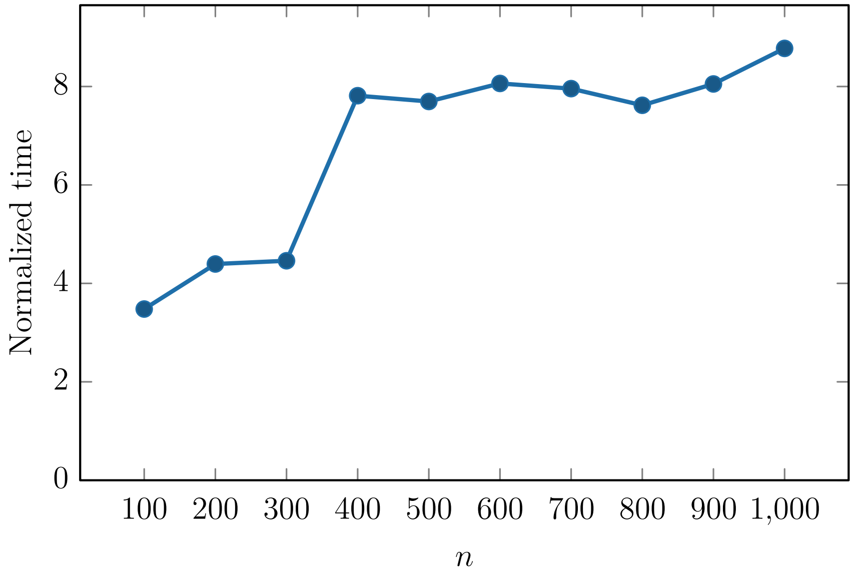


Figure 5.4. Normalized running times of benchmark **matmul** with dimensional procedure rewriting disabled

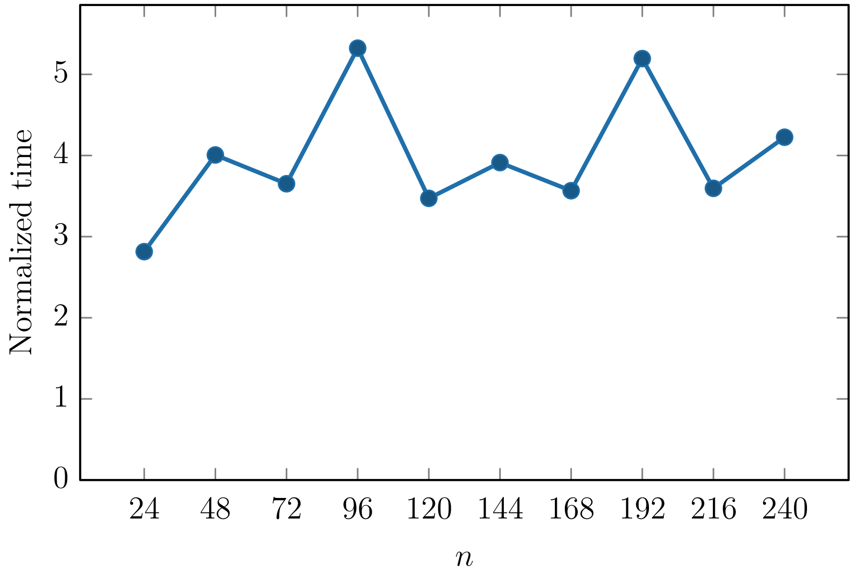


Figure 5.5. Normalized running times of benchmark **thomas** with dimensional procedure rewriting disabled

and procedure invocations, which tend to prevent aggressive compiler optimizations. Furthermore, Figures 5.2, 5.3, 5.4 and 5.5 also show how different types of numerical applications react to dimensional procedure rewriting being disabled. Benchmark deriv is communication-bound for small n and thus suffers heavy penalties when there is no communication aggregation; matmul is computation-bound but requires loop tiling to hide the memory access cost; thomas is memory bandwidth-bound, whose relatively constant performance loss results from lack of vectorization. Explicitly rewritten procedures are necessary to enable all these optimizations. Our programming tool automatically performs the rewriting and keeps the programmer's programming burden to the minimum.

5.6.4 Effect of computation–communication interleaving

In section 5.5.2, we mentioned that a greedy strategy for interleaving computation and communication is to initiate the communication operations as early as possible and complete them as late as possible. We use the benchmark interlv to demonstrate the advantage of our MILP-based method over this greedy strategy. In interlv, there are two separate dependence chains

$$\begin{aligned} &X_0 \to X_1 \to X_2 \to X_3 \to X_4 \to X_5 \to X_6, \\ &Y_0 \to Y_1 \to Y_2 \to Y_3 \to Y_4 \to Y_5 \to Y_6 \end{aligned}$$

where each X_k depends on X_{k-1} by $\mathcal{O}(n^3)$ computation, and each Y_k depends on Y_{k-1} by $\mathcal{O}(n^2)$ communication. The greedy strategy first starts the communication operation of Y_1, then finishes the computation of all X_k before completing the communication operations of all Y_k. In contrast, our MILP-based method perfectly interleaves the two dependence chains.

Figure 5.6 shows the running times of the greedy strategy normalized to those of our MILP-based method. The zigzag pattern is due to the fact that our method has greater fluctuation in performance than the greedy strategy. Our method is slower by

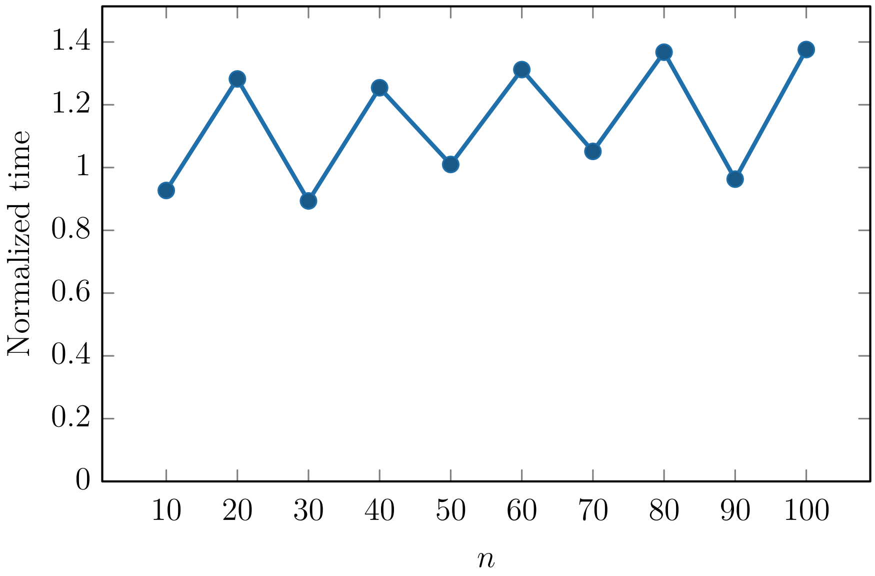


Figure 5.6. Normalized running times of benchmark interlv with greedy computation–communication interleaving

more than 5 % in only two cases and up to 38 % faster in the best case. Overall, our method shows an average speedup of 1.13 over the greedy strategy.

5.6.5 Effect of array dimension elimination and array coalescing

We demonstrate the necessity of applying both array dimension elimination and array coalescing to the code sketches produced by dimensional procedure rewriting and computation–communication scheduling using the benchmarks `conpq`, `deriv`, `matmul` and `thomas`. We consider all four cases wher array dimension elimination and array coalescing are turned on and off independently. We also include optimized handwritten implementations of the benchmarks in the comparison.

We plot the running times of the four benchmarks under different conditions in Figure 5.7. The data are normalized to those of running the benchmarks with both array dimension elimination and array coalescing enabled. When both optimizations are disabled, the code sketches are translated into Fortran literally, preserving the immutable nature of functional programming and all array dimensions conservatively introduced by dimensional procedure rewriting. In particular, the resulting code physically stores all intermediate computation results as arrays. For programs involving asymptotically more intermediate values than input and output values such as `conpq` and `matmul`, this leads to very significant slowdown. Also, notice that not all benchmarks need both optimizations—array dimension elimination has no significant effect on `deriv` and is undone by array coalescing in `thomas`, while array coalescing is inapplicable in `matmul`. However, in order to cover all possible situations, both of them must be enabled.

Compared to handwritten code, our programming tool achieves essentially the same level of performance in `conpq`, `matmul` and `thomas`. For `deriv`, the average speedup is a somewhat surprising 1.13. Since the algorithms and data structures are identical, and employ essentially the same optimization techniques as our programming tool in the handwritten code, we suspect that it is differences in the implementation

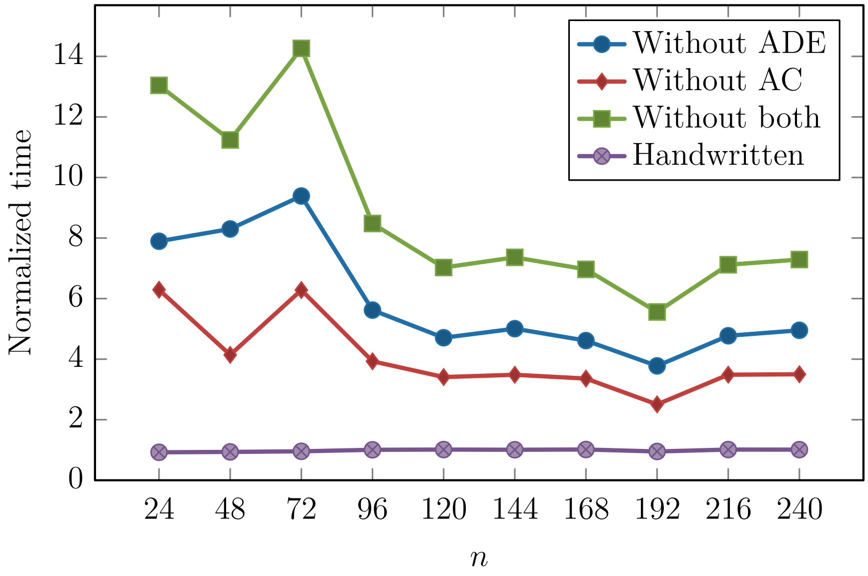


Figure 5.7. Normalized running times of benchmark `conpq` under different conditions

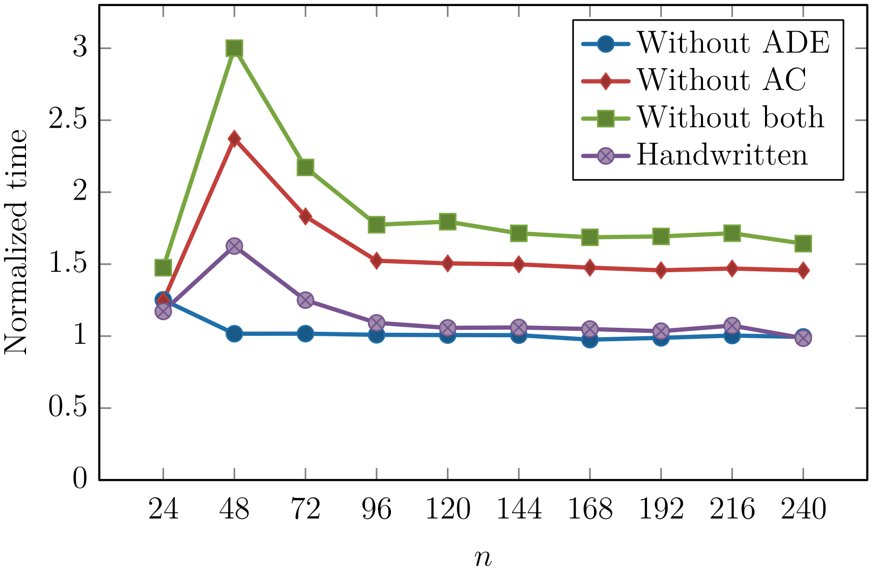


Figure 5.8. Normalized running times of benchmark `deriv` under different conditions

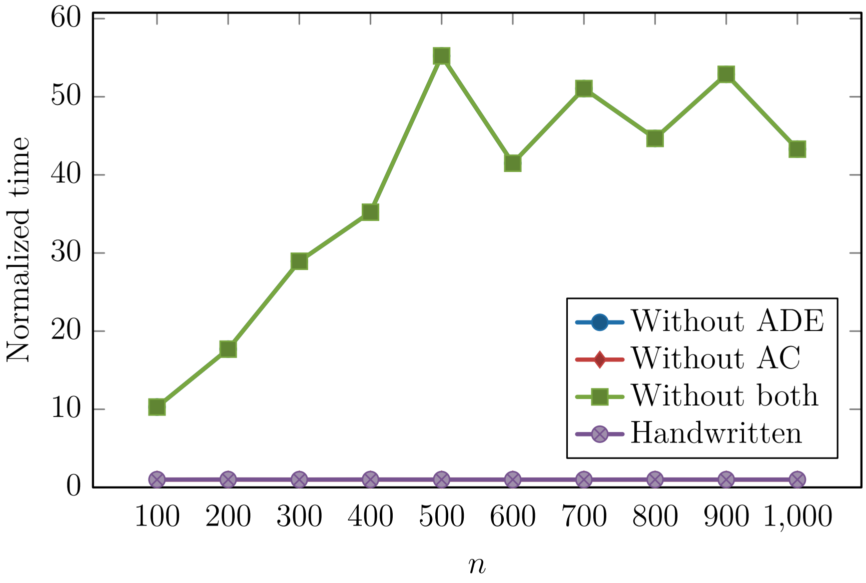


Figure 5.9. Normalized running times of benchmark `matmul` under different conditions

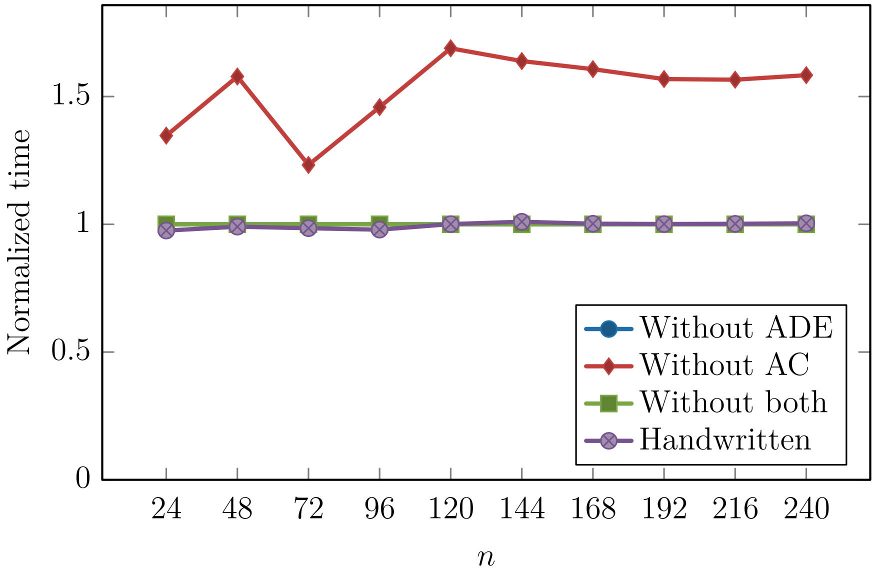


Figure 5.10. Normalized running times of benchmark `thomas` under different conditions

details that caused the Fortran compiler to generate code with different efficiency. But because `deriv` is communication- and memory bandwidth-bound, the headroom for further optimization is limited. Therefore, we can claim that our programming tool is competitive in terms of program efficiency compared to manual programming.

5.6.6 Applicability in finite difference-based three-dimensional large eddy simulation of jet engine noise

Table 5.1 summarizes the major components of the three-dimensional LES-based jet engine noise prediction application in [48]. The application consists of a mix of elemental and vector algorithms. The physics-related components have elemental formulations, but their definitions depend on spatial partial differentiation of the flow variables, which is defined on vectors. The spatial filtering scheme is also vector-based in nature.

Our programming tool can be used to implement all the major components of the jet engine noise prediction application except for a transposition step in the centerline treatment which temporarily reslices and redistributes the grid partitions among the processors, a computation pattern beyond the expressiveness of our functional array programming model. When it comes to evaluating the programming effort, due to the syntactic differences between the input language of our programming tool and Fortran, a direct quantitative comparison based on counts of lines of code has limited value. Therefore, we use a qualitative comparison instead. For elemental algorithms, our programming tool does not enable much reduction in the programming effort because their implementations in Fortran is also every simple. For vector algorithms, our programming tool significantly reduces the programming effort because it needs only one implementation which operates on a single vector for each algorithm. In comparison, when implemented in Fortran, in order to deliver the maximum performance, each algorithm requires two two-dimensional computation kernels, namely a columnwise version and a rowwise version; more boilerplate code is needed to

Table 5.1.
Major algorithmic components of large eddy simulation-based jet engine noise prediction

Component	Type of algorithm
Time integration	Elemental
Spatial partial differentiation	Vector
Navier–Stokes equations	Elemental
Boundary conditions	Elemental
Outflow damping	Elemental
Spatial filtering	Vector
Acoustic statistics	Elemental

repeatedly invoke those computation kernels on three- and four-dimensional arrays and handle communication of arrays with different numbers of dimensions.

6 CONCLUSIONS AND SUGGESTED DIRECTIONS FOR FUTURE EXPLORATION

6.1 Conclusions

In this study, we examined the numerical methods and programming practices of finite difference-based three-dimensional LES of jet engine noise. Two specific issues pose challenges to researchers and engineers in the development of high-fidelity LES-based jet engine noise prediction applications. First, due to the stringent requirements on numerical accuracy of CAA, finite difference methods in LES of jet engine noise must rely on an implicitly formulated compact scheme for spatial partial differentiation. A performance-critical component of the compact spatial partial differentiation scheme is the embedded tridiagonal linear system solver. Previously, researchers and engineers have utilized methods including the transposition method, the multiblock method and the Schur complement method to solve the tridiagonal linear systems. Each of these methods makes different trade-off between numerical accuracy and empirical efficiency, both of which are the desired features of an ideal solver. Second, the aforementioned compact spatial partial differentiation scheme needs to be applied to the computational grid along the three coordinate directions of the computational space. So does a compact spatial filtering scheme applied to the stabilize the otherwise divergent numerical method of finite difference-based LES. Without programming tools which provide dedicated support for regular grid-based numerical application, the programming tasks of implementing these numerical schemes are tediously repetitive and error-prone, especially for non-CS researchers and engineers.

To tackle the challenges posed by these two issues, we first described an efficient parallel tridiagonal linear system solver based on the truncated SPIKE algorithm. Our algorithm avoids the unscalable grid reslicing and redistribution present in the transposition method and is methodologically closer to the Schur complement method in the sense that our algorithm also reduces the original system to a smaller reduced system. Taking advantage of the diagonal dominance of the tridiagonal linear systems which arise from the compact spatial partial differentiation and spatial filtering schemes, it uses block Jacobi iteration instead of parallel cyclic reduction or more complex iterative methods such as the Krylov subspace methods to solve the reduced system and consequently requires only communication between neighboring processors in the logical grid of processors. The resulting solver solves the tridiagonal linear systems accurately as the transposition method and the Schur complement method do and offers theoretically provable and empirically confirmed optimal scalability in weak scaling scenarios as the multiblock method does. In addition to presenting the theory of our tridiagonal linear system solver, we also described strategies for implementing the compact spatial partial differentiation and spatial filtering schemes efficiently in a practical LES-based jet engine noise prediction application. Experimental performance measurements show that our new implementation of the application achieves significant speedup over an implementation based on the transposition method, especially when the number of processor cores participating in the computation is large.

Next, we presented two programming models and the associated code optimization and generation methods which enable simple expression of application of lower-dimensional algorithms in higher-dimensional contexts, a pattern of computation frequently found in regular grid-based numerical applications. The first programming model is imperative and is based on generalization of Fortran elemental subroutines. An ordinary Fortran elemental subroutine repeatedly applies the same sequence scalar operations on combinations of individual elements taken from one or more arrays. In what we refer to as a generalized elemental subroutine, we allow the elemental data objects to be, in addition to individual array elements, array slices of arbitrary dimensions and define an appropriate semantics of repetition accordingly. Through loop nest generation, local variable transformation and subroutine invocation aggregation, we demonstrated that generalized elemental subroutines can enable a level of performance in the generated code close to that of handwritten code while significantly reducing the amount of programming effort. However, the code optimization methods designed for generalized elemental subroutines are relatively conservative due to the fact that the programming model does not make any assumptions about the elemental semantics of the subroutines. This motivated us to consider a second programming model. The latter programming model is functional and restricts the expressible semantics to several patterns that are sufficiently expressive to specify the computation needed by finite difference-based LES but compact enough that the semantics can be analyzed precisely using the polyhedral model. This design enabled us to employ the more advanced polyhedral model-based computation scheduling methods compared to the heuristic-driven loop nest generation method of generalized elemental subroutines and additionally consider computation–communication interleaving and array coalescing in the code optimization and generation process. As a result, we were able to match the performance level of handwritten code with automatically generated code in empirical experiments.

6.2 Suggested directions for future exploration

6.2.1 Efficient numerical methods and implementation of finite difference-based large eddy simulation of jet engine noise using implicit time integration

As we have mentioned in section 1.1, the CFL condition dictates that the time integration step size in LES must scale proportionally to the unit grid spacing in order to maintain numerical stability. This implies that the finer the grid spacing, the more time steps are needed to complete the time integration. Consequently, the computational cost of LES for the same jet simulation problem increases superlinearly with respect to the total size of the computational grid. For high-fidelity LES, such superlinear scaling can render the computation prohibitively expensive. One possible mitigation to this limitation in the time integration step size is use an implicit time integration method as opposed to the explicit Runge–Kutta methods. The feasibility of implicit time integration is rooted in the fact that while the CFL condition links the integration step size to the unit grid spacing, it does so only qualitatively and does not prescribe a concrete quantitative relation between the two quantities. Therefore, although the CFL condition is fundamentally uncircumventable, it is possibly to exploit the extra numerical stability provided by implicit time integration methods over the explicit methods to increase the time integration step size.

The formulation of LES with implicit time integration in [48], which is an extension of the method proposed in [8] to three-dimensional simulation problems, uses a linear multistep method (LMM) based on the second-order backward difference formula (BDF). Being an alternating-direction implicit (ADI) method, it avoids the need to solve linear systems which couple the three coordinate directions of the computational space and instead relies on linear systems defined on grid lines as in the case of explicit time integration. Notable characteristics of the linear systems include:

- Each linear system involves all five flow variables, and its coefficient matrix is comprised of 5×5 blocks capturing the interactions between the flow variables.
- In order to achieve fourth-order and sixth-order spatial accuracy, the coefficient matrices need to be block tridiagonal and block pentadiagonal, respectively.
- The coefficient matrices are not diagonally dominant. Quite the contrary, the main diagonals are likely the lightest-weighted in terms of element magnitudes among all diagonals.
- The coefficient matrices are time- and space-dependent, i.e., they vary from one time integration step to another and from one grid line to another.

These characteristics necessarily challenge the feasibility of straightforward generalization of our specialized version of the truncated SPIKE algorithm to the block tridiagonal and block pentagonal linear systems. If a new linear system solver is to be applied, it must be numerically stable and sufficiently efficient so that the net performance gain from the increased time integration step sizes minus the added computational cost due to the more complex mathematical formulation is not negated.

6.2.2 Extension of the semantic model and code optimization strategies of the functional array programming model

While the functional array programming model described in Chapter 5 are sufficient for expressing the major patterns of computation in finite difference-based LES of jet engine noise and capable of generating code whose efficiency with performance on par with handwritten code, the following extensions to its semantic model and code optimization strategies which expands its coverage of regular grid-based numerical applications are worth considering:

- The syntactic rules of array definitions by arithmetic can be relaxed. The syntactic restrictions in section 5.3.1 only cater to the needs of finite difference-based LES, but the complexity of the permissible syntaxes for array definitions by arithmetic is really only constrained by the operation scheduling method used by computation scheduling and the definition–use chain analysis method used by array dimension elimination and array coalescing. If these methods are readily able or can be extended to handle the more complex syntaxes, there is no need to stay confined to the current design.
- Flexible array dimension sizes and recursive procedure invocation can be introduced. As an importance class of regular grid-based numerical applications, multigrid (MG) methods cannot be expressed by our functional array programming model. In order to represent the grid hierarchy, especially in the restriction and prolongation steps, array dimension sizes must not be limited to compile-time constants and the symbolic run-time constants n_k. Recursion in procedure invocation is also necessary for full multigrid (FMG) methods, which additionally implies the need for predicated versions of array definitions.
- Array dimension elimination and array coalescing can take loop tiling into consideration. For array dimension elimination, this means to detect the smallest size that a dimension of an array can be reduced to if that dimension cannot

be completely eliminated. When loop tiling is present, array elements accessed in different loop tiles may not have overlapping lifetimes even if those accessed in the same loop tiles do. This creates opportunities to reduce the sizes of the involved array dimensions to the tile sizes. For array coalescing, this means to use loop tiling combined with proper computation scheduling to work around overlapping lifetimes between arrays in stencil-like computation, which can potentially enable in-place implementations.

REFERENCES

REFERENCES

[1] K. M. Aikens. *High-fidelity large eddy simulation for supersonic jet noise prediction.* PhD dissertation, School of Aeronautics and Astronautics, Purdue University, 2014.

[2] K. M. Aikens, N. S. Dhamankar, C. S. Martha, Y. Situ, G. A. Blaisdell, A. S. Lyrintzis, and Z. Li. Equilibrium wall model for large eddy simulations of jets for aeroacoustics. AIAA Paper 2014-0180, AIAA, 2014.

[3] K. M. Aikens, G. A. Blaisdell, A. S. Lyrintzis, and Z. Li. Analysis of converging–diverging beveled nozzle jets using large eddy simulation with a wall model. Accepted AIAA Paper, 2015.

[4] P. R. Amestoy, I. S. Duff, J. L'Excellent, and J. Koster. MUMPS: A general purpose distributed memory sparse solver. In *The Fifth International Workshop on Applied Parallel Computing*, volume 1947 of *Lecture Notes in Computer Science*, pages 121–130, 2001.

[5] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 2–11, 1990.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004.

[7] C. Bastoul. CLooG, 2013. `http://cloog.org/`.

[8] R. M. Beam and R. F. Warming. An implicit factored scheme for the compressible Navier–Stokes equations. *AIAA Journal*, 16(4):393–402, 1978.

[9] C. Bogey and C. Bailly. Three-dimensional non-reflective boundary conditions for acoustic simulations: Far field formulation and validation test cases. *Acta Acustica*, 88(4):463–471, 2002.

[10] C. Bogey, C. Bailly, and D. Juve. Noise investigation of a high subsonic, moderate Reynolds number jet using a compressible LES. *Theoretical and Computational Fluid Dynamics*, 16(4):273–297, 2003.

[11] C. Bogey, O. Marsden, and C. Bailly. Effects of moderate Reynolds numbers on subsonic round jets with highly disturbed nozzle-exit boundary layers. *Physics of Fluids*, 24(10):105–107, 2012.

[12] C. Bogey, O. Marsden, and C. Bailly. Influence of initial turbulence level on the flow and sound fields of a subsonic jet at a diameter-based Reynolds number of 10^5. *Journal of Fluid Mechanics*, 701:352–385, 2012.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[14] G. A. Brès, J. W. Nichols, S. K. Lele, F. E. Ham, R. H. Schlinker, R. A. Reba, and J. C. Simonich. Unstructured large eddy simulation of a hot supersonic over-expanded jet with chevrons. AIAA Paper 2012-2213, AIAA, 2012.

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

[16] COIN-OR Foundation. COIN-OR Branch-and-Cut, 2014. `https://projects.coin-or.org/Cbc`.

[17] Cray Inc. *Using Cray Performance Analysis Tools*, 2010.

[18] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, and X. S. Li. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[19] N. S. Dhamankar. Boundary conditions towards realistic simulation of jet engine noise. Master's thesis, School of Aeronautics and Astronautics, Purdue University, 2013.

[20] J. J. Dongarra and A. H. Sameh. On some parallel banded system solvers. *Parallel Computing*, 1(3–4):223–235, 1984.

[21] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical Python. *Computers in Physics*, 10(3):262–267, 1996.

[22] Free Software Foundation, Inc. GNU Linear Programming Kit, 2014. `http://www.gnu.org/software/glpk/`.

[23] R. W. Freund and N. M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60(1):315–339, 1991.

[24] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[25] E. Garnier, P. Sagaut, and M. Deville. A class of explicit ENO filters with applications to unsteady flows. *Journal of Computational Physics*, 170(1):184–204, 2001.

[26] E. Garnier, N. Adams, and P. Sagaut. *Large Eddy Simulation for Compressible Flows*. Springer, 2009.

[27] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(3):12–21, 1993.

[28] C. Grelck. Single Assignment C (SAC) – High productivity meets high performance. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer Berlin Heidelberg, 2012.

[29] C. Grelck and S. Scholz. Axis control in SAC. In *Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 182–198. Springer Berlin Heidelberg, 2003.

[30] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.

[31] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12(1):95–113, 1965.

[32] H. Jasak, A. Jemcov, and Ž. Tuković. OpenFOAM: A C++ library for complex physics simulations. In *Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics*, pages 47–66, 2007.

[33] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[34] Y. Khalighi, J. W. Nichols, S. K. Lele, F. E. Ham, and P. Moin. Unstructured large eddy simulations for prediction of noise issued from turbulent jets in various configurations. AIAA Paper 2011-2886, AIAA, 2011.

[35] J. W. Kim and D. J. Lee. Generalized characteristic boundary conditions for computational aeroacoustics. *AIAA Journal*, 38(11):2040–2049, 2000.

[36] J. W. Kim and D. J. Lee. Generalized characteristic boundary conditions for computational aeroacoustics, part 2. *AIAA Journal*, 42(1):47–55, 2004.

[37] M. Klein, A. Sadiki, and J. Janicka. A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations. *Journal of Computational Physics*, 186(2):652–665, 2003.

[38] E. K. Koutsavdis, G. A. Blaisdell, and A. S. Lyrintzis. Compact schemes with spatial filtering in computational aeroacoustics. *AIAA Journal*, 38(4):713–715, 2000.

[39] S. K. Lele. Compact finite difference schemes with spectral-like resolution. *Journal of Computational Physics*, 103(1):16–42, 1992.

[40] P.-T. Lew. *A study of sound generation from turbulent heated round jets using 3-D large eddy simulation.* PhD dissertation, School of Aeronautics and Astronautics, Purdue University, 2009.

[41] M. J. Lighthill. On sound generated aerodynamically. I. General theory. *Proceedings of the Royal Society A*, 211(1107):564–587, 1952.

[42] M. J. Lighthill. On sound generated aerodynamically. II. Turbulence as a source of sound. *Proceedings of the Royal Society A*, 222(1148):1–32, 1953.

[43] C. Lin and L. Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 96–114. Springer Berlin Heidelerg, 1994.

[44] J. Liu, C. Kaplan, and E. Oran. A brief note on implementing boundary conditions at a solid wall using the FCT algorithm. Memorandum Report NRL/MR/6410-06-8943, Naval Research Laboratory, 2006.

[45] S.-C. Lo, K. M. Aikens, G. A. Blaisdell, and A. S. Lyrintzis. Numerical investigation of 3-D supersonic jet flows using large-eddy simulation. *International Journal of Aeroacoustics*, 11(7–8):783–812, 2012.

[46] D. B. Loveman. High Performance Fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[47] A. S. Lyrintzis. Surface integral methods in computational aeroacoustics—from the (CFD) near-field to the (acoustic) far-field. *International Journal of Aeroacoustics*, 2(2):95–128, 2003.

[48] C. S. Martha. *Toward high-fidelity subsonic jet noise prediction using petascale supercomputers*. PhD dissertation, School of Aeronautics and Astronautics, Purdue University, 2013.

[49] C. S. Martha, Y. Situ, M. E. Louis, G. A. Blaisdell, A. S. Lyrintzis, and Z. Li. Development and application of an efficient, multiblock 3-D large eddy simulation tool for jet noise. AIAA Paper 2012-0833, AIAA, 2012.

[50] S. Mendez, M. Shoeybi, A. Sharma, F. E. Ham, S. K. Lele, and P. Moin. Large-eddy simulations of perfectly expanded supersonic jets using an unstructured solver. *AIAA Journal*, 50(5):1103–1118, 2012.

[51] C. C. K. Mikkelsen and M. Manguoglu. Analysis of the truncated SPIKE algorithm. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1500–1519, 2008.

[52] J. Mohd-Yosuf, D. Livescu, and T. Kelley. Adapting the CFDNS compressible Navier–Stokes solver to the Roadrunner hybrid supercomputer. In *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 95–109. DEStech Publications, Inc., 2010.

[53] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[54] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. D., Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 224–235, 2011.

[55] J. W. Nichols, S. K. Lele, P. Moin, F. E. Ham, and J. E. Bridges. Large-eddy simulation for supersonic rectangular jet noise prediction: Effects of chevrons. AIAA Paper 2012-2212, AIAA, 2012.

[56] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2013.

[57] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[58] M. Pharr and W. R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*, pages 1–13, 2012.

[59] E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: The SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006.

[60] E. Polizzi and A. H. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007.

[61] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[62] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[63] P. Sagaut. *Large Eddy Simulation for Incompressible Flows*. Springer, 3rd edition, 2006.

[64] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics*, 40(1):158–176, 2000.

[65] Y. Situ and Z. Li. A practical approach towards composing regular grid-based numerical applications using functional array programming, 2014. Submitted to the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[66] Y. Situ, L. Liu, C. S. Martha, M. E. Louis, Z. Li, A. H. Sameh, G. A. Blaisdell, and A. S. Lyrintzis. Reducing communication overhead in large eddy simulation of jet engine noise. In *2010 IEEE International Conference on Cluster Computing*, pages 255–264, 2010.

[67] Y. Situ, L. Liu, C. S. Martha, M. E. Louis, Z. Li, A. H. Sameh, G. A. Blaisdell, and A. S. Lyrintzis. A communication-efficient linear system solver for large eddy simulation of jet engine noise. *Cluster Computing*, 16(1):157–170, 2013.

[68] Y. Situ, Y. Wang, and Z. Li. Automated rapid prototyping of regular grid-based numerical applications using generalized elemental subroutines. In *2013 IEEE International Parallel and Distributed Processing Symposium*, pages 284–294, 2013.

[69] Y. Situ, C. S. Martha, M. E. Louis, Z. Li, A. H. Sameh, G. A. Blaisdell, and A. S. Lyrintzis. Petascale large eddy simulation of jet engine noise based on the truncated SPIKE algorithm. *Parallel Computing*, 40(9):496–511, 2014.

[70] C. K. W. Tam. Advances in numerical boundary conditions for computational aeroacoustics. AIAA Paper 1997-1774, AIAA, 1997.

[71] H. K. Tanna, P. D. Dean, and R. H. Burrin. The Generation and Radiation of Supersonic Jet Noise, Volume III: Turbulent Mixing Noise Data. Technical Report AFAPL-TR-76-65, Air Force Aero-Propulsion Laboratory, 1976.

[72] Top500.org. Top500 List – June 2014, 2014. `http://www.top500.org/list/2014/06/`.

[73] E. Touber and N. D. Sandham. Large-eddy simulation of low-frequency unsteadiness in a turbulent shock-induced separation bubble. *Theoretical and Computational Fluid Dynamics*, 23(2):79–107, 2009.

[74] A. Uzun. *3-D Large Eddy Simulation for Jet Aeroacoustics.* PhD dissertation, School of Aeronautics and Astronautics, Purdue University, 2003.

[75] A. Uzun and M. Y. Hussaini. Simulation of noise generation in near-nozzle region of a chevron nozzle jet. *AIAA Journal*, 47(8):1793–1810, 2009.

[76] A. Uzun and M. Y. Hussaini. High-fidelity numerical simulations of a round nozzle jet flow. AIAA Paper 2010-4016, AIAA, 2010.

[77] A. Uzun and M. Y. Hussaini. On some issues in large-eddy simulations for chevron nozzle jet flows. AIAA Paper 2011-0019, AIAA, 2011.

[78] A. Uzun and M. Y. Hussaini. Some issues in large-eddy simulations for chevron nozzle jet flows. *Journal of Propulsion and Power*, 28(2):246–258, 2012.

[79] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[80] S. Verdoolaege. Integer Set Library, 2014. `http://isl.gforge.inria.fr/`.

[81] M. R. Visbal and D. V. Gaitonde. Very high-order spatially implicit schemes for computational acoustics on curvilinear meshes. *Journal of Computational Acoustics*, 9(4):1259–1286, 2001.

[82] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[83] D. K. Wilde. The ALPHA language. Research Report 2295, INRIA, 1994.

[84] D. K. Wilde. *From ALPHA to imperative code: A transformational compiler for an array based functional language.* PhD dissertation, Oregon State University, 1995.

[85] Z.-T. Xie and I. P. Castro. Efficient generation of inflow conditions for large eddy simulation of street-scale flows. *Flow, Turbulence and Combustion*, 81(3):449–470, 2008.

[86] H. C. Yee, N. D. Sandham, and M. J. Djomehri. Low-dissipative high-order shock-capturing methods using characteristic-based filters. *Journal of Computational Physics*, 150(1):199–238, 1999.

[87] X. Zhang, G. A. Blaisdell, and A. S. Lyrintzis. High-order compact schemes with filters on multi-block domains. *Journal of Scientific Computing*, 21(3):321–339, 2004.

VITA

VITA

Yingchong Situ was born and raised in Guangzhou, China. Upon graduation from high school in 2004, he enrolled in Peking University and obtained his Bachelor of Science degree in Computer Science and Technology in 2008. He then attended graduate school at Purdue University in the Department of Computer Science and completed his Master of Science and Doctor of Philosophy degrees in Computer Science in 2014. His PhD research focused on improving the scalability of finite difference-based large eddy simulation of jet engine noise prediction and developing programming tools which simplify the programming tasks of the aforementioned application. After graduation from Purdue University, he joined Google Inc. as a software engineer.