Dynamic Load Balancing for a hp-adaptive Discontinuous Galerkin Wave Equation Solver via Spacing-Filling Curve and Advanced Data Structure

Shiqi He

Thesis submitted to the University of Ottawa
in partial Fulfillment of the requirements for the

Master of Applied Science

Department of Mechanical Engineering
Faculty of Engineering
University of Ottawa

Dynamic Load Balancing for a hp-adaptive Discontinuous Galerkin Wave Equation Solver via Spacing-Filling Curve and Advanced Data Structure

by

Shiqi He

Abstract

Scientific or engineering simulations of complex fluid flows, *e.g.*, turbulent flow, often resort to high-order methods to obtain high resolutions. However, a large-scale, time-dependent, long-time simulation can be extremely computationally expensive. To achieve high resolution while maximizing the computational savings, we combine a high-order method – the *discontinuous Galerkin spectral element method (DG-SEM)*, with parallel *adaptive mesh refinement and coarsening (AMR)* techniques and apply it to a two-dimensional wave equation solver. DG-SEM discretizes solution functions as weighted polynomial series. In this work, *Legendre polynomials* are chosen. With higher-order method and AMR techniques, a computational cost-efficient solver is formed.

Two types of refinements are invoked: *h-refinement*, where elements are split or merged, and *p-refinement*, where polynomial orders can be increased or decreased. To retain the spectral convergence on the *non-conforming* element interfaces caused by the hp-refinement, the *mortar element method* is adopted.

A hash table AMR technique is developed for the AMR data encoding and storing. It can be built independently on distributed memory systems. It provides high control of the mesh resolution and efficient data management operations. Results using this approach on the adaptive wave equation solver confirm that its memory usage remains low on up to $16,384$ processors for a problem with over a million elements and 150 million degrees of freedom.

Dynamic load imbalance is incurred by the adaptivity of the program, which degrades the performance of the supercomputers. A *space-filling curve (SFC) based* repartitioning algorithm is implemented in this work. The algorithm is designed to execute quickly in parallel. Its low memory overhead is beneficial to distributed memory systems. Additionally, the high-quality repartitioning results successfully reduce the dynamic workload imbalance among the

computational processors. The scalability of the load balancing algorithm is demonstrated on two different high-performance computing systems with up to $16,384$ processors. The maximum memory scaling is up to 4,096 processors and no considerable memory growth is observed beyond the maximum scaling. High load balancing quality is demonstrated.

Using the hash table AMR and SFC-based repartitioning algorithm, we obtain speedup factors up to 8.46 over a range of processor numbers from 32 to 2048, which, in the long run, can reduce week-long computational wait-times to a matter of days.

Keywords: Spectral element method, adaptive mesh refinement, space-filling curves, Hilbert curve, dynamic load balancing, large-scale scientific computing.

Acknowledgement

I would like to give my sincere gratitude to my supervisor Prof. Catherine Mavriplis. She gave me the first lecture when I came to university of Ottawa and lately became my supervisor. As an International student, I encountered some obstacles at the beginning of my master's study. However, Prof. Mavriplis offered me enormous patience and never spared her compliments to me, even to a meager progress I made, which strongly encouraged me and supported me throughout my research life. Her meticulous and persistence attitudes towards research impressed me and enlightened me. Every time I felt frustrated or impatient to my research I would think about her. For me, she represents peace and resilience. I feel extremely luck to have her as my supervisor and I am very grateful to the guidance she gave me, both in study and in life.

Secondly, I would like to give thank to the two members of our research team: Mayank Vadsola and François Lepage. We are three master's students that started in the same semester. Mayank and François offered me tremendous help. They are always so patient to me and willing to give me help or guidance towards my study or research. Without them, my thesis cannot be finished in such pace.

Thirdly, I am grateful to every professor that lectured me. The knowledge I learned form them laid the foundation of my thesis. Moreover, their friendly personality smooths out my tiredness and made the whole studying process so enjoyable.

Next, I would like to thank my family: my mum and my dad. It is them who supported me to pursuit my degree abroad. Especially my mother, who never pushed me or questioned me about my progress. On the contrary, she encouraged me to enjoy my university life as much as possible. She is always so supportive and respective to the decisions I made. Again, I can say nothing but thank you to my mother and father.

Moreover, I would like to thank the University of Ottawa. University of Ottawa composed my major life here. Two years passed so swift, and when I looked back, the school provided me excellent study facilities and conditions. It brings so many amazing people united. I feel lucky and grateful to be a student here. University of Ottawa deems to be a very pleasant experience in my memory.

Last but not least, I would like to give my gratitude to the institutions that support my the-

sis. This research was supported by the Natural Sciences and Engineering Research Council of Canada through Discovery Grant (RGPIN-2017-05320) and a Grant from the Collaborative Research and Training Experience (CREATE- 481695-2016) program in Simulation-Based Engineering Science. The scholarship it provides largely support my research. I also want to thank all the staff members that helped me in Compute Canada. They assisted my research with respect to the interaction with HPCs.

Contents

Title Page ii

Abstract ii

Acknowledgement v

Contents xi

List of Figures xii

List of Tables xx

List of Algorithms xxiii

List of Symbols xxiv

List of Greek Symbols xxvii

Other Symbols xxviii

Superscripts xxix

Subscripts xxx

Other Notations xxxi

List of Acronyms xxxii

1 Introduction 1

2 Discontinuous Galerkin Spectral Element Method **10**

2.1 Finite Element Approximation . . . 10

2.1.1 Evaluate the Weak Form of the Problem . . . 10

2.1.2 Weighted Residual Approximation . . . 12

2.2 Spectral Approximation . . . 13

2.2.1 Functions of Approximation . . . 13

2.2.2 Polynomial Basis Functions . . . 14

Legendre Polynomials . . . 15

2.2.3 Gauss Quadrature . . . 15

2.2.4 Polynomial Interpolation in Lagrange Form . . . 16

2.3 Approximation of Wave Equations . . . 17

2.3.1 System of Equations . . . 18

2.3.2 Discontinuous Galerkin Approximation . . . 19

Apply DG-SEM to One Element . . . 19

Riemann Problem for Conservation Laws . . . 24

Boundary Fluxes . . . 26

Time Integration . . . 30

General Runge-Kutta Method . . . 30

Third-order Low-storage Runge-Kutta . . . 32

Stability Analysis . . . 33

Runge-Kutta Stability Analysis . . . 33

CFL Condition . . . 38

Fourier Analysis of Hyperbolic Equation . . . 42

Apply DG-SEM to Multiple Elements . . . 48

Implementation of Solver . . . 50

2.3.3 Benchmark Solution: Plane Wave Propagation . . . 51

Convergence Performance . . . 52

3 Adaptive Mesh Refinement **54**

3.1 Structured Grid AMR . . . 55

3.2 hp-adaptivity . . . 57

3.3 Error Estimator . . . 60
3.3.1 Truncation Error . . . 61
3.3.2 Quadrature Errors . . . 63
3.3.3 Summary . . . 64
3.4 Refinement Criteria . . . 65
3.5 Adaptivity Performance . . . 66
3.5.1 Solution Resolution Improvement Demonstration . . . 66
3.5.2 Experiment One: Resolve Plane Wave Starting from a Coarse Mesh . . . 69
3.5.3 Experiment Two: Reflecting Waves . . . 71

4 Non-conforming Mortar Element Discretization 75
4.1 Element Interfaces . . . 75
4.2 Mortar Element Method . . . 76
4.2.1 From Element to Mortar . . . 77
4.2.2 From Mortar to Element . . . 79
4.2.3 Examples . . . 81

5 Data Structure 83
5.1 Data Structures for AMR . . . 83
5.2 Proposed AMR Data Structure: Hash Table AMR . . . 85
5.2.1 Hash Table Construction . . . 85
5.2.2 Pairing Functions . . . 87
Cantor Pairing Function . . . 87
Szudzik Pairing Function . . . 88
5.2.3 Element Storage . . . 89
Full Element Storage Symbols . . . 91
5.3 Comparison Between Hash Table and Tree Structure AMR . . . 91

6 Interprocessor boundaries update methods 95
6.1 Element Interface Storage . . . 96
6.2 MPI Boundary Table . . . 97
6.2.1 MPI Table Example . . . 98

6.2.2 MPI Table Implementation . . . 100
6.3 MPI Derived Data Types . . . 110
6.3.1 Implementation . . . 110
Using Extents . . . 111
Using Addresses . . . 114
6.4 Update Interprocessor Interfaces . . . 115

7 Dynamic Load Balancing **121**
7.1 Repartitioning Methods . . . 122
7.1.1 Graph-based Repartitioning Algorithms . . . 122
7.1.2 Space-Filling Curves based Repartitioning Algorithms . . . 123
7.2 Space-Filling Curves . . . 124
7.2.1 Introduction to Space-Filling Curves . . . 124
7.2.2 Morton Curve . . . 125
7.2.3 Hilbert Curve . . . 126
7.3 Fast-generated Hilbert Curve Algorithm . . . 126
7.3.1 Classifications . . . 127
7.3.2 A Table-driven Algorithm . . . 127
7.4 Workload Redistribution Algorithms . . . 130
7.4.1 Chains-on-chains partition . . . 130
Partitioning Strategy . . . 130
7.4.2 Repartitioning Algorithm . . . 132
All-to-all Collective Method . . . 133
Iterative Method . . . 133
Centralized Method . . . 133
Distributed Method . . . 133
Step-by-step Explanation with a Four Processors Example . . . 134
Whole Algorithm . . . 138
7.5 Interprocessor Boundaries Update Method . . . 145
7.6 Reallocate Elements . . . 149
7.6.1 Derived Data Types . . . 150

7.6.2 Element Transfer Algorithms . . . 151
7.7 Whole Load Balancing Algorithm Summary . . . 163
7.8 Dynamic Load Balancing Performance . . . 165
7.8.1 Load Balancing Performance of Experiment One . . . 166
7.8.2 Load Balancing Performance of Experiment Two . . . 168

8 Results **172**

8.1 Introduction to scalability . . . 172
8.2 TAU Profiler . . . 174
8.3 Experiment Platforms . . . 175
8.3.1 Graham . . . 175
8.3.2 Cedar . . . 175
8.4 Execution Time . . . 175
8.4.1 Experiment Three . . . 175
Strong Scaling . . . 176
Weak Scaling . . . 177
8.4.2 Experiment Four . . . 178
Strong Scaling . . . 179
8.5 Memory Usage . . . 179
8.5.1 Experiment Three . . . 180
Strong Scaling . . . 180
Weak Scaling . . . 181
8.5.2 Experiment Four . . . 181
Strong Scaling . . . 182
8.6 Load Balancing Quality . . . 182
8.6.1 Experiment Five . . . 183
8.6.2 Experiment Six . . . 184
8.7 Application Performance . . . 185
8.7.1 Experiment Seven: High Load Imbalance . . . 185
8.7.2 Experiment Eight: Low Load Imbalance . . . 187
8.7.3 Experiment Nine: Application Scalability with Low Refinement Level . . . 188

8.7.4 Experiment Ten: Application Scalability with High Refinement Level . . 189

9 Conclusions **192**

9.1 Conclusions 192

9.2 Future Work 194

Bibliography **196**

List of Figures

1.1 AMR example: isobars and mesh cut on a business jet configuration computed with AMR approach (reprinted from [22]). 3

1.2 AMR example: shock reflection off with an oblique wedge with AMR approach (reprinted form [9]). Regions of finer mesh resolution are shown as the rectangle blocks in the figure. 4

1.3 Conforming and non-conforming element interfaces illustration. Elements are in colour. The intersections of the black lines represent the collocation points within an element. In this example the yellow element has $4th$ order polynomials (*i.e.*, 5 points in each direction). 5

1.4 Two AMR approaches. Colour schemes represent different hierarchies of mesh. . 6

2.1 Legendre polynomials $P_n(x)$ of orders n ranging from 0 to 5. 15

2.2 2D reference domain: $[-1,1]\times[-1,1]$. f_i $(i = 1, ..., 4)$ are the four element boundaries and $\mathbf{n_i}$ are the unit vectors pointing outwards from the element boundaries. 20

2.3 Solution nodes and boundary flux nodes. Black round nodes are the Gauss-Legendre collocation nodes where solutions are approximated. The blue square nodes are boundary flux nodes where element fluxes are approximated. They are the Gauss-Legendre collation nodes placed on the boundaries. 22

2.4 Illustration of the initial state of the Riemann problem. $\mathbf{u}_L$ and $\mathbf{u}_R$ are the left and right states of the Riemann problem. At time 0 two states are separated at position $x = x_0$. 25

2.5 Element boundary flux on the right element interface. $\widehat{n}$ is the unit vector. $\mathbf{Q}^L$ and $\mathbf{Q}^R$ are the interior and external solutions respectively. w^+ and w^- are used to indicate the right-going and left-going waves. 27

2.6 Stable region of the model function $y = e^{\lambda t}$. $z = h\lambda$, where h is the step size. The stable region is shaded in grey. 34

2.7 The configuration of stability function $R(z)$ of the 3rd-order low-storage RK method. $Re(z)$ is the real axis, $Im(z)$ is the imaginary axis. The absolute value of $R(z)$ is plotted in the vertical axis. The contour plot of $|R(z)| = 1$ is shown as a grey line in (b). 37

2.8 Absolute stability region of the 3rd-order low-storage RK method. Point p_1 is the intersection point on the real axis. Points p_2 is the intersection point on the imaginary axis. 37

2.9 Domain of dependence. 39

2.10 Domain of dependence example: Euler method in time and upwind spatial scheme to solve the advection equation. The numerical domain of dependence is coloured in yellow. The analytical solution is a straight line with slope $\frac{1}{c}$, which is indicated by the red line. To fulfill the CFL condition, the slope of the numerical domain of dependence (green line) should be less than the slope of the analytical domain of dependence (red line). 39

2.11 Numerical domain of dependence for 3rd-order RK method using central difference scheme. k_i $(i = 1, 2, 3)$ are the intermediate stages of RK. Solution at point x_j^{n+1} depends on solutions at points range from x_{j-3}^n to x_{j+3}^n. The numerical domain of dependence is coloured in yellow. 41

2.12 (a) The distribution of the eigenvalues of the Legendre first order derivative of order $N = 32$. (b) The relationship between the growth of the maximum eigenvalue with respect to the polynomial order. (Reprinted from [40]) 47

2.13 Solution discontinuity between elements. The solutions Q^k of kth element (Ω^k) are defined on the Gauss points inside the element region. The solutions (Q^L on the left and Q^R on the right) on the element interface x_k between element Ω^k and Ω^{k+1} are not required to be equal. 48

2.14 Propagation of Gaussian plane wave at three time snapshots. Colour scheme: the magnitude of the pressure. Domain size: $[0, 1] \times [0, 1]$. The wave propagates from the lower left corner to the upper right corner. The DG-SEM numerical solution is obtained by using $K = 16 \times 16 = 256$ elements, polynomials of order $p = 6$ in each direction and a time step of $\Delta t = 1 \times 10^{-5}$. 52

2.15 Exponential error convergence is obtained for both pressure p and velocity u. The green line represents the $L2$ error in the pressure. The orange line is for velocity. The two lines both obey the exponential convergence rate. 52

3.1 Large-scale problem example: simulation of turbulent flow around a NACA4412 wing section with a uniform C-mesh. 3.2 billion grid points are used to get a well-resolved result. Even with 16,384 cores, 35 million CPU hours are needed to get the convergence of turbulence. Reprinted from [78]. 54

3.2 The data structures of two AMR approaches. Colour schemes represent different hierarchies of mesh. (a) Blocked-structured AMR. (b) Three-structured AMR . . 56

3.3 hp-refinement illustration (2D). 57

3.4 Three types of element non-conforming interfaces. Type one: geometrically non-conforming. Type two: functionally non-conforming. Type three: geometrically and functionally non-conforming. 58

3.5 Vortex shedding simulation using (a) non-conforming and (b) conforming discretization. Shown are instantaneous vorticity contours in the near-wake. (Reprinted from [38].) 59

3.6 Convergence comparison between h- and p-type refinement for an elliptic Helmholtz problem. Plot of the $H1$ norm error as a function of the total number of degrees of freedom for coarse (a) and fine (b) meshes. (Reprinted from [38].) 60

3.7 Spectrum decay trend: when the solution is smooth, the spectrum a_n has an exponential decay, and the magnitude of the slope of the least squares best fit is larger than 1 (green dashed line). When the solution is poor, the magnitude of the slope will be less than 1 (blue dashed line). 66

3.8 Wave illustration. 67

3.9 Computed and exact pressure are compared on the cross-section $x = 0.55$ at time $t = 0.5$. 67

3.10 Computed error in solution at time $t = 0.5$. The magnitude of the $L2$ norm of the error is presented by the colour scheme. Note that the range of the colour bars are different between the two figures. 68

3.11 Adaptive mesh solution: computed and exact pressure at time $t = 0.5$, $K = 8$ elements are used on the cross-section. The elements on the left side of the red dashed line have polynomial order $p = 8$ and elements on the right have polynomial order $p = 6$. 69

3.12 AMR Experiment One Scene One: resolve a Gaussian plane wave starting from a coarse mesh. The configuration of the pressure is shown in purple. The polynomial order p of each element is indicated by the degree of greyness in the figure, ranging from 6 to 12. K is the number of elements. 70

3.13 AMR Experiment One Scene Two: resolve a Gaussian plane wave from a coarse mesh. The configuration of the pressure is shown in colour purple. The polynomial order p of each element is indicated by the degree of greyness in the figure, ranging from 6 to 12. K is the number of elements. 71

3.14 Wave 1 (in red) is propagating towards the upper right corner of the real domain. Its reflection on the right boundary can be seen as a mirror image wave (in green), moving with the same vertical velocity (with respect to the reflection boundary) in magnitude but in the opposite direction. 72

3.15 AMR Experiment Two Scene One: compare the refinement strategies applied to a well-resolved wave (red) and a mal-resolved (reflected) wave (blue). The magnitude of the velocities u are shown in red and blue. The polynomial order p of each element is indicated by the degree of greyness in the figure. K is the number of elements. 73

3.16 AMR Experiment Two Scene Two: compare the refinement strategies applied to a well-resolved wave (red) and a mal-resolved (reflected) wave (blue). The magnitude of the velocities u are shown in red and blue. The polynomial order p of each element is indicated by the degree of greyness in the figure. K is the number of elements. 74

4.1 Element interfaces: an element with polynomial order N in x direction and M in y direction is represented as $P_{N,M}$. Element 1 and 3 share conforming interfaces. Element 2 and 5 are geometrically non-conforming. Element 2 and 4 are functionally non-conforming. Element 4 and 5 are geometrically and functionally non-conforming. 75

4.2 Elements use mortars to communicate between interfaces. Ω^k represents kth element and the grey blocks are the mortars between elements. 76

4.3 Mortar configuration. Ω refers to the element and Ξ represents mortar. 77

4.4 Mortar element method demonstration: grids for the convergence using conforming interfaces, functionally non-conforming interfaces, geometrically non-conforming interfaces. 81

4.5 A comparison of conforming and non-conforming interface errors as a function of approximation polynomial order. The green line shows the convergence of the conforming interface. The orange line represents geometrically non-conforming interface error convergence. The yellow line is the error performance of functionally non-conforming interface. All of them exhibit exponential convergence, demonstrating the validity of the mortar element method. 82

5.1 (Quad-)tree data structure for 2D meshes. Each node in the tree represents an element in the mesh. In the quadtree data structure, one node is subdivided into 4 children nodes. 84

5.2 Hash table data structure. A value inside a hash table is accessed with its unique key. 85

5.3 Element coordinates 86
5.4 Cantor pairing function graphical illustration. 87
5.5 Szudzik pairing function graphical illustration. 88
5.6 Element storage in hash table. Each element stores its coordinate, numerical solutions, neighbour's keys (in $facen$), pointer to the next element ($next$), and other problem related variables. 89
5.7 Example: left element refines twice and get the right mesh. 90
5.8 A comparison of two data structures for the same mesh refinement (of Fig. 5.7) . 90
5.9 2:1 balance restriction: the element edge length cannot not exceed 2 times its adjacent elements' length in the 2D lid-driven cavity meshes. (Reprinted from [58].) 93
5.10 h-refinement without geometrical restriction: no geometrical restriction is imposed on the adjacent elements' non-conforming interfaces in the 2D wave problem meshes. 94

6.1 Element interface storage configuration. 96
6.2 Four directions of the element faces: south, north, west and east. 97
6.3 mpi_table data structure. 97
6.4 MPI table construction example: mesh distribution. The MPI boundary between Rank 0 and Rank 1 is denoted by the red dashed line. The elements coloured in pink reside on Rank 0 and the green elements on Rank 1. The element keys begin with "k", and the element coordinates are in the brackets. 98
6.5 MPI table construction example: element $k31$'s north interface possibilities. . . 101
6.6 MPI table construction example: element $k31$ finds its new neighbours. 102
6.7 Relationship between element boundary length and element length exposed to the MPI boundary. MPI boundary is indicated by the red dashed line. Processors are discriminated by the colour scheme. Face 1 (north face) of element $E1$ is partially exposed to the MPI boundary, therefore, element boundary length $\leq$ element length on the MPI boundary. 109
6.8 32 bit machine memory bus. A single bank is one byte width. A machine with 4 banks needs one read cycle to fetch a 4 byte integer. 113
6.9 Data structure padding example. A character occupies 1 byte of memory and an integer occupies 4 bytes of memory. In order to place integer b on a whole row of memory bus, 3 bytes are kept vacant after character a. The same logic is applied to the padding after character c. 113

7.1 Two popular SFCs: Hilbert curve and Morton curve 124

7.2 Morton curves for different grid sizes/levels. The Morton curve can be generated fast but "jumps" are inevitable in its pattern, for example, the jump from the yellow cell to the green cell. 125

7.3 Hilbert curves for different grid sizes/levels. 126

7.4 Construction of the Hilbert curve. Each cell has its own status s ($s \in \{H, A, B, R\}$). The status of the parent cell determines the geometrical arrangement of the children cells. 127

7.5 Four types of geometrical arrangements of the children cells. With the status $S(v)$ of a parent cell v, its children's status sequence can be derived by using Table 7.1. Then the children's geometrical arrangement can be found in the four cases shown. The colour scheme in this figure matches with the colour applied to the children sequence in Table 7.1. 128

7.6 The position numbers of four sibling cells. The number indicates the geometrical position of a cell among four sibling cells. 129

7.7 The Hilbert curve traverses the mesh with mixing levels. 130

7.8 Four processors example: initial element partition. The colour scheme in the figure represents different processors. 134

7.9 Four processors example: exclusive prefix sum. 135

7.10 Four processors example: adding the exclusive prefix sum value to the local prefix sum, the global prefix sum is obtained. 135

7.11 Four processors example: compute the processor mapping values. The value of *pmapping* is the element's future rank number. Note that Processor 2 and 3 will need to send their last elements to the next processor (indicated by the arrows). 136

7.12 Four processors example: each processor sends their last element's global ID with the *pmapping* value to the next processor, if any. 136

7.13 Four processors example: local *proc_mapping_table*. 137

7.14 Example: global *proc_mapping_table* 137

7.15 Four processors example: load balancing result. 138

7.16 Receiver decision-making for rank 0. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours. 152

7.17 Receiver decision-making for the last rank. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours. 152

7.18 Receiver decision-making for ranks in between. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours. 153

7.19 Load balancing Experiment One Scene One: Start the simulation with 4 elements ($K = 4$). Time $t = 0$. (a) Configuration of the wave in purple and the polynomial order (p) of each element is indicated by the degree of greyness. (b) Colour scheme represents different processors. Elements in the same colour reside on the same processor. 166

7.20 Load balancing Experiment One Scene Two: hp-refinements progresses and the workloads are balanced among processors. Time $t = 6.0 \times 10^{-5}$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.19. 167

7.21 Load balancing Experiment One Scene Three: hp-refinement progresses and the workloads are balanced among processors. Time $t = 8.0 \times 10^{-5}$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.19. 168

7.22 Load balancing Experiment Two Scene One: mesh and the workloads among processors at time $t = 0$. (a) Configuration of the velocities in the x direction in red and blue, and the polynomial order p of each element is indicated by the degree of greyness. (b) The colour scheme represents different processors. Elements in the same colour reside on the same process. 169

7.23 Load balancing Experiment Two Scene Two: hp-refinement progresses and the workloads are balanced among processors. Time $t = 0.038$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.22. 170

7.24 Load balancing Experiment Two Scene Three: hp-refinement progresses and the workloads are balanced among processors. Time $t = 0.5$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.22. 170

8.1 Example: theoretical speedup according to Amdahl's law. The upper limit of the parallel speedup is determined by the serial portion of the program. The colour schemes correspond to different parallel portions (parallel portion = 1 − serial portion) of a program. 173

8.2 Example: theoretical speedup according to Gustafson's law. The speedup has no upper limit, but its slope is affected by the serial portion of the program. The colour schemes correspond to the different serial portions. 174

8.3 Execution time strong scaling of the load balancing algorithm on Graham with $K = 16,384$ elements, initially. 176

8.4 Main routines' execution time strong scaling profile. Except for MPI_Exsan (rose bars) and $MPI\_Barrier$ (yellow bars), other routines scale up well, which indicates that MPI_Exscan is the bottleneck of the load balancing algorithm. . 177

8.5 Execution time weak scaling on Cedar with $K = 128$ elements per processor (P). 178

8.6 Execution time strong scaling on Graham with $K = 1.048576 \times 10^6$ elements, initially. 179

8.7 Memory consumption strong scaling on Graham with $K = 16,384$ elements, initially. 180

8.8 Memory usage weak scaling on Cedar with 128 elements per processor. (Note the small range of the memory consumption axis.) 181

8.9 Memory consumption strong scaling on Graham with $K = 1.048576 \times 10^6$ elements, initially. 182

8.10 Load balancing efficiency on Graham with $K = 16,384$ elements, initially. Green curve represents the results with load balancing (LB) and the yellow curve stands for results without LB. 183

8.11 Load balancing efficiency on Graham with $K = 1.048576 \times 10^6$ elements, initially. Green curve represent the results with load balancing (LB) and the yellow curve stands for results without. 184

8.12 Performance comparison: time consumption of each simulation time step with load balancing (LB) (green line) and without LB (red line). A significant speedup can be observed for LB. (On Cedar) 186

8.13 Performance comparison: speedup factors (Cedar) 186

8.14 Performance comparison: time consumption of each simulation time step with load balancing (LB) (green line) and without LB (red line). After applying LB, a significant speedup can be observed. (On Graham) 187

8.15 Performance comparison: speedup factors (Graham). 188

8.16 Strong scalability indication (maximum h-refinement level = 2). 189

8.17 Strong scalability experiment: resolve a wave with h-refinement level of 5. . . . 190

8.18 Strong scalability indication (maximum h-refinement level = 5). 191

List of Tables

2.1 Butcher tableau of Runge-Kutta method. 31

2.2 Butcher tableau of an s-stage explicit Runge-Kutta method. 32

2.3 Coefficients of the third-order low storage Runge-Kutta method. 33

2.4 Butcher tableau of explicit 3rd-order low-storage Runge-Kutta method. 36

2.5 Butcher tableau of Runge-Kutta 3rd-order method. 41

3.1 The values of the $L2$ norm error on the domain at time $t = 0.5$ for uniform mesh and adaptive mesh. 67

5.1 Symbols of terms of element storage 91

5.2 Tree-structured AMR and hash table AMR features comparison. 92

6.1 MPI table construction example: MPI north table of Rank 0. 99

6.2 MPI table construction example: MPI south table of Rank 1. 99

6.3 MPI table construction example: record the keys of the elements on the MPI boundary. 100

6.4 MPI table construction example: deduce the possible neighbours of the elements on the MPI boundary. 101

6.5 MPI table construction example: calculate the element length that is exposed to the MPI boundary. 102

7.1 Children cell statuses table: children status values $S^C(s, j)$ for 2D Hilbert-curve. s refers to the status of the parent element, j refers to the jth child. The colour scheme in the table corresponds to that of the geometrical arrangements in Fig. 7.5. 128

7.2 Children cell positions table: children position values $P^C(s, j)$ for 2D Hilbert-curve. s refers to the status of the parent cell, j refers to the jth child. 129

7.3 Summary of the important symbols in the load balancing algorithm. 131

7.4 Symbols of variables in element reallocation 149

8.1 Weak scaling problem set up with $K = 128$ elements per core. 178

List of Algorithms

2.1 **Riemann_solver_x:** Riemann solver in x direction. 29
2.2 **Riemann_solver_y:** Riemann solver in y direction. 30
2.3 **DG_step_by_RK3**: time integration by using third-order low-storage Runge-Kutta method. 33
2.4 **Driver_for_DG_approximation:** driver for DG-SEM approximation. 51
2.5 **DG_time_der:** compute the time derivatives. 51
5.1 **Szudzik**: Szudzik pairing function. 89
5.2 **Get_key_fun**: inputs element coordinates, outputs the key of the element. . . . 89
6.1 **Elem_length**: return the element boundary length. 98
6.2 **Put_in_mpi_table**: insert current MPI boundary element to the MPI table. . . 103
6.3 **Possible_neighbours**: form the possible neighbour array based on the face number. 104
6.4 **Neighbour_array_x**: form possible neighbour array in h-refinement level descending order (x direction). 105
6.5 **Neighbour_array_y**: form possible neighbour array in h-refinement level descending order (y direction). 106
6.6 **Total_neighbours**: compute the total number of possible neighbours. 107
6.7 **Construct_mpi_table**: construct MPI table in one direction. 108
6.8 **Record_length**: Record the element length that is exposed to a certain neighbour processor. 109
6.9 **Sender_recver**: processors communicate to update MPI boundaries. 116
6.10 **Update_hash**: update variable $facen$ (neighbour information) in element hash table. 118
6.11 **Erase_old_face**: erase the old MPI boundary information on target face: $facei$. 119
6.12 **Update_mpi_boundaries**: algorithm to update MPI boundaries in one direction. 119
6.13 **Adapt**: apply hp-refinement to local element and update the interprocessor boundaries. 119
7.1 **Build_mapping_table**: build the processor mapping table. Processors transfer elements based on such table. 140

7.2 **Elem_load**: element computation workload. 142
7.3 **Update_neighbours**: update the element's local neighbours information in the $facen$ variable before element transfer. 143
7.4 **Neighbour_change**: update the corresponding neighbour $facen$. Note that the neighbour element will remain local. 144
7.5 **Opposite_dir**: input the current face number, return the opposite face number. 145
7.6 **Ownership_one_dir**: fill the ownership of the elements in the MPI table in one direction. 146
7.7 **Send_recv_ownership**: exchange MPI boundary tables to updates the MPI boundaries. 147
7.8 **Update_mpib**: update the MPI boundaries based on the element ownership received. 148
7.9 **Change_face**: compare the neighbour's new rank with the old one and update the $facen$ accordingly. 148
7.10 **Update_mpi_boundaries_LB**: update the MPI boundaries before repartitioning. 149
7.11 **Reallocate_elem**: redistribute the elements. 153
7.12 **Send_pack**: pack element character information. 157
7.13 **Solution_pack**: pack up element solutions. 157
7.14 **Face_pack**: store all the element neighbours in a vector. 158
7.15 **Erase_elem_old**: erase the elements in the sending list from the local element hash table. 159
7.16 **Recv_elem**: receive element information from the sender. 159
7.17 **Recv_solu**: receive element solution from the sender. 160
7.18 **Recv_face**: receive element's neighbour information from the sender. 160
7.19 **Enlarge_hash**: after receiving element information, we create new units in hash table and store them. 161
7.20 **Fill_facen**: put the neighbour information inside the element hash table. 163
7.21 **Load_balancing:** complete load balancing algorithm. 163
7.22 **Clear_mapping_tables:** clears up the processor mapping tables and other variables that are related to the repartitioning algorithms 164
7.23 **MPI_Table_rebuild:** rebuild the MPI tables in a and y directions. 164
7.24 **Clear_tables:** clear up all the MPI tables and possible neighbours. 164
7.25 **Load_balancing_recursive**: recursive load balancing algorithm. Partition the workload until the user defined load efficiency is reached. 165

List of Symbols

A_x, A_y	constant matrix
C	a constant
$D_{i,k}$	derivative matrix
E	edge set, load efficiency
E_{opt}	optimal load efficiency
F	first time derivative
G	intermediate variable in time integration, graph
J	maximum polynomial order between left and right elements
K	number of element
L, L_k	Legendre polynomial
L_s	local total workload on partition s
L_{max}	maximum local total workload/bottleneck of partition
M	polynomial order in y direction
N	polynomial order in x direction, node number in a tree data structure, processor number, interval number
P	projection matrix, total processor number
$R(x)$	residual function
$R(z)$	stability function on complex plane
S	surface, speedup
U	solution on the element side, approximated solution
V**,** V_i	volume, vertex set, a partition
$W(i)$	global prefix sum of the ith element
W_{global_sum}	total workload
X_h, X_m	finite function space
Δt	time step
$\Delta x_k, \Delta y_k$	kth element size in x and y direction respectively
$\mathbf{A}$	Jacobian matrix, coefficient matrix
$\mathbf{F}$	numerical fluxes
$\mathbf{F}, \mathbf{F}_{i,j}, \mathbf{G}_{i,j}$	approximated fluxes
$\mathbf{K}$	matrix of right eigenvectors of coefficient matrix $\mathbf{A}$

$\mathbf{Q}, \mathbf{Q}_{i,j}$	approximated vector of solutions
$\mathbf{W}$	characteristic variables of Riemann problem
$\mathbf{k}$	wave vector
$\mathbf{q}$	the vector of three targeted components
$\mathfrak{F}$	flux flux
$\widehat{D}_{i,k}$	weighted derivative matrix
$\widehat{x}, \widehat{y}, \widehat{n}$	unit vector
$\widetilde{a}_n$	Legendre approximation coefficient computed by extrapolation
$\widetilde{u}_i$	nodal value of basis function
$\widetilde{u}_i$	coefficient of the basis function expansion
a_k	Legendre polynomial coefficient
a_m, b_m, g_m	coefficients in 3rd-order Runge-Kutta method
a_n	function coefficient, Legendre approximation coefficient
a_{ij}, b_j, c_i	general Runge-Kutta coefficients
b_n	exact Legendre polynomial coefficient
c	sound speed
e	natural number, edge
f	solution function, flux in x direction
f_i	element face number
g	flux in y direction
h	time step size
i	imaginary unit ($i^2 = -1$)
k	wave number
k_x, k_y	wavevector component
l, l_j	Lagrange interpolating polynomial
p, p_N	solution function, pressure, parallel portion of the program
$prefix(i)$	local prefix sum of the ith element
s	serial portion of the program
$u, u_n, \widehat{u}_n$	solution function, velocity component in x direction, vertex
u_h	approximated solution
u_{quad}	quadrature error
u_{trunc}	truncated discretized series
v	velocity component in y direction, vertex
$v(x)$	an arbitrary function
w	polynomial weight
w^+	left-going wave
w^+	right-going wave
w_{ideal}	ideal workload per partition
x, y, r, s	coordinate axis
x_k, y_k	element right face coordinate

x_{k-1}, y_{k-1}	element left face coordinate
z	coordinate axis, complex number

List of Greek Symbols

Γ	adjacent vertices set, domain boundary
Λ	diagonal matrix of coefficient matrix $\mathbf{A}$
Ξ	mortar element
Φ	numerical flux on mortar
Ψ	solution projection on the mortar
Ω	element representation, computational domain
Ω_e	domain of element e
α	the offset of linear least squares best fit
β	the slope of linear least squares best fit
$\delta_{i,j}$	Kronecker delta
ϵ_{est}	estimated error
ϵ	discretization tolerance, balance ratio
η	reference space coordinate in y direction
λ, λ_i	eigenvalue
ξ	reference space coordinate in x direction
σ	error decay rate
τ	truncation error
ϕ_i, ϕ_n	basis function, orthogonal basis functions
φ, φ_j	test function
ψ	intermediate term in barycentric Lagrange interpolation

Other Symbols

$\bigcup$	union operation over a collection of sets
$\bigtriangledown$	gradient operator
$\bigtriangledown\cdot$	divergence operator
$\int$	integral operator
$\mathbb{N}$	natural number set
∂	partial derivative
$\pi, \pi(\cdot,\cdot), \pi(\cdot,\cdot,\cdot)$	pairing function
$\prod$	product operator
$\sum$	summation operator
d	total derivative

Superscripts

$'$	function first derivative
(i)	column i of a matrix
$*$	flux variable
$+$	right-going (wave)
$-$	left-going (wave)
-1	matrix inverse
0	stationary (wave)
C	child cell
L	left interface
R	right interface
T	matrix transpose
k	kth element
n	solution at time step n, nth element

Subscripts

0	initial state
L	left state
R	right state
c	child element
h	approximated
i	solution at point i
k, n, m	element right interface
$k-1, n-1, m-1$	element left interface
max	maximum value
min	minimum value
n	time step n
p	parent element
t	time derivative
x, y	spatial first derivative with respect to x/y direction
xx, yy	spatial second derivative with respect to x/y direction

Other Notations

$C(v, j)$	jth child of cell v
$F(\cdot)$	function representation
$O(\cdot)$	big O notation: order of
$Re(\cdot)$	the real component of a complex number
$S(v)$	the status of cell v
$\cdot$	time derivative
$\eta(\cdot)$	the weight of a vertex
∞	infinity
$\|\cdot\|$	absolute value operator
bold	discrete vector
$\overline{a}$	average value of a
$\theta(\cdot)$	the weight of an edge
$\widehat{\ }$	unit vector
$int(\cdot)$	take the integer part of
$max(\cdot, \cdot)$	get the maximum of two values

List of Acronyms

2D	two-dimensional
3D	three-dimensional
AMR	adaptive mesh refinement and coarsening
ARC	advanced research computing
CCP	chains-on-chains partitioning
CFD	computational fluid dynamics
CFL	Courant Friedrichs Lewy
CG-SEM	continuous Galerkin spectral element method
DG-SEM	discontinuous Galerkin spectral element method
DNS	direct numerical simulation
FEM	finite element method
FTT	fully threaded tree
GL	Gauss-Legendre
HPC	high performance computer
IC	initial condition
IMEX	implicit explicit
IVP	initial value problem
LB	load balancing
MPI	message passing interface
ODE	ordinary differential equation
PDE	partial differential equation
RK	Runge-Kutta
SFC	space-filling curve
TAU	tuning and analysis utilities

Chapter 1

Introduction

The development of digital computers in the $20th$ century has had a huge impact on the way the sciences of fluid mechanics are applied to the design and development of industrial equipment and vehicles. Traditionally, experimental and theoretical methods were used in engineering practice. Now, a third methodology is gaining prominence alongside the rapid development of high performance computers. The third methodology, known as computational fluid dynamics (CFD), is a numerical approach. This numerical approach is applied to solve the partial differential equations (PDEs) that govern a process of interest.

Although experimental methods are still irreplaceable, the trend to a heavier reliance on computer-based predictions in design is clear [63]. Chapman shows that the cost of performing a given calculation has been decreased by approximately a factor of 10 every 8 years [21]. The cost of experiments can be avoided by using CFD methods for some cases, for instance, the extensive flight tests of an airplane, full-scale tests of a gas turbine, or destructive testing of expensive components [19]. For other cases, even if CFD methods cannot fully supplant experimental approaches, they can be used to reduce the number of experimental tests. A more cost-efficient approach is to combine experimental measurement with CFD simulations. CFD simulations can be applied to improve our understanding of the fluid phenomenon in the design, while the experimental methods can be used to justify the CFD results. This strategy meets the requirements of today's development: improving the performance of the design while being more environmental friendly.

However, even though the CFD approaches contribute to a large saving, solving a three-dimensional, time-dependent complex flow, for example, turbulent flow, is prohibitively expensive [19]. To get an accurate solution by using CFD methods is still impractical for many engineering problems. According to NASA's CFD Vision 2030 study [2], the use of CFD in aerospace design is limited by the inadequate accuracy and unreliable prediction of turbulent flow separations. To obtain accurate details of turbulent flow, not only do we need better computer hardware, but also higher resolution numerical methods. The spectral method is one of

them.

Unlike conventional numerical methods like the finite difference method or the finite element method, the spectral method is a high-order method with rapid convergence. It uses a simple representation of a function $u(x)$ throughout the domain via a truncated series expansion:

$$u(x) \approx u_N(x) = \sum_{n=0}^{N} \widehat{u}_n \phi_n(x), \tag{1.1}$$

where $\phi_n(x)$ are the *basis functions*. Inserting the discretization into differential or integral equations, the unknowns $\widehat{u}_n$ are computed [38]. The basis functions are usually chosen as *Legendre polynomials* or *Chebyshev polynomials*. More details about the usage of spectral methods in fluid mechanics can be found in the book of Canuto *et al.* [18].

Spectral methods are usually classified into two categories [38]: (1) collocation methods and (2) Galerkin methods. The second category solves equations in integral form over the domain which is broken into non-overlapping elements. In this work, we combine spectral approximation with the Galerkin approach, in the so-called *discontinuous Galerkin spectral element method (DG-SEM)*, to solve the acoustic wave equations. DG-SEM uses a flux formulation across element boundaries. Despite the discontinuities at element boundaries, DG-SEM retains the *spectral (exponential) convergence* (for smooth solutions) of the spectral method. This high-order method exhibits small diffusion and dispersion errors, and the small data volume-to-surface ratio is beneficial for parallel computing [38]. Moreover, high-order methods play a very important role in time-dependent complex flow studies, for instance, turbulent flow. For stationary CFD problems, a quadratic convergence rate is usually sufficient for a typical engineering accuracy of 10%. However, this is not true for a time-dependent flow over a long time simulation. Kreiss *et al.* in their experiments [43] indicate that the required resolution depends on the number of time periods M, the lower the order of the scheme the stronger the dependence is. According to Kreiss, the computational cost, W, for a prescribed accuracy (same phase error), for a short period of time ($M \propto O(1)$), low-order methods might be favoured, but for long-time integration ($M \propto O(100)$), a sixth-order scheme's total operation count is six times less than for a second-order scheme. In the end, it is cheaper to use high order method. Therefore, in the limit of high accuracy and long-time integration, high order methods are more computationally efficient than low-order methods [38, 43]. For spectral-based methods, the exponential convergence is even more efficient for calculations that require higher accuracy, *e.g*, DNS and instability analysis.

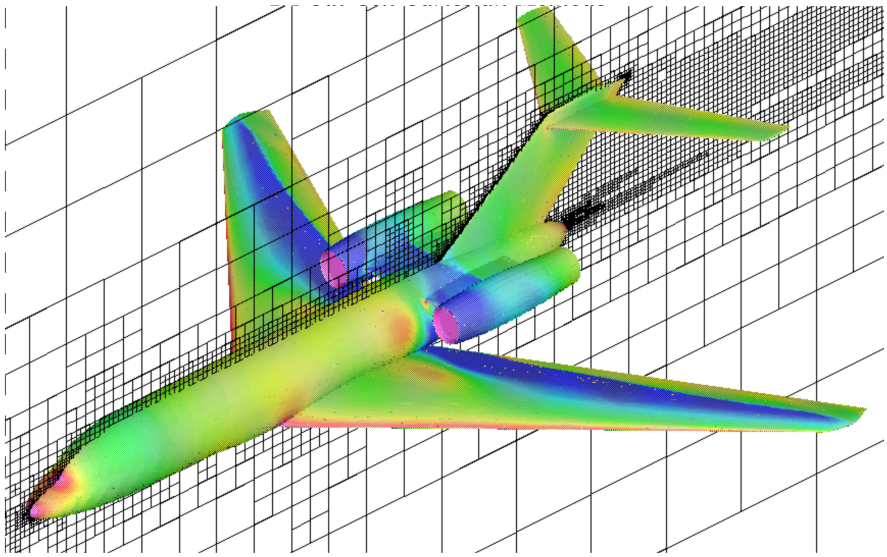


Figure 1.1: AMR example: isobars and mesh cut on a business jet configuration computed with AMR approach (reprinted from [22]).

Although today's advanced high performance computers (HPCs) support scientists and engineers to investigate their problems in a fast turnaround time using parallel processing, grand challenge problems can require the computational domain to be discretized into millions of elements in order to get the desired accuracy, which makes it impractical to solve with uniform fine meshes even with the help of a petascale supercomputer. Additionally, uniform meshes sometimes do not provide sufficient resolution to computationally hard regions. For example, if we want to trace the details of a shock wave, we need a high resolution around the discontinuous region. In order to efficiently use resources for resolution, *adaptive refinement and coarsening (AMR)* methods have been developed to tackle the previous problems. With the help of AMR, we can start with a coarse mesh and have the algorithm identify the regions that need a finer mesh. In this way, spatial and temporal high-resolutions are triggered only at needed locations and times (*e.g.* in Fig. 1.1 finer meshes are constructed around the configuration of the airplane and in Fig. 1.2, finer meshes are placed on the difficult regions of a shock wave.). Berger *et al.* [9] showed that AMR could save a factor of 8 or more CPU time over an equivalent uniform grid in their 2D shock wave reflection problem. Moreover, AMR is beneficial to the memory consumption when compared with the equivalent resolution on a uniform mesh: the uniform mesh required 2.2 times more memory storage than AMR to obtain the same level of resolution in their numerical experiments. *PARAMESH* [51] also reports that the degree of time improvement increases with higher level of refinement. In our work, we combine the DG-SEM with AMR so that the program can tackle sophisticated fluid phenomena while utilizing the available computational resources efficiently.

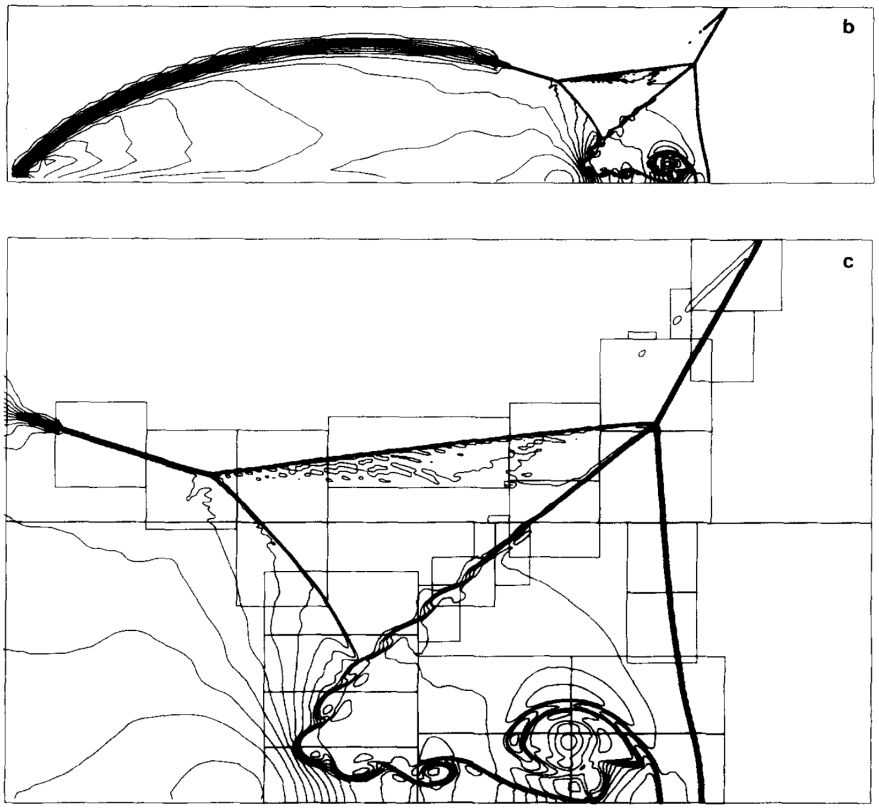
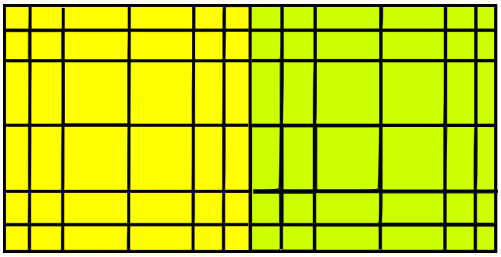


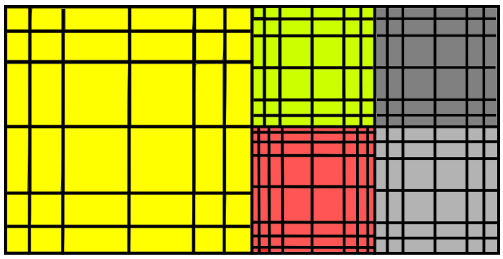
Figure 1.2: AMR example: shock reflection off with an oblique wedge with AMR approach (reprinted form [9]). Regions of finer mesh resolution are shown as the rectangle blocks in the figure.

Two types of mesh refinements are implemented in this work. The first one subdivides elements into smaller elements to improve the resolution. We call this type of mesh enrichment *h-refinement*. The second type of refinement is provided exclusively by high-order methods: *p-refinement*. This type of refinement raises the polynomial orders of the discretization inside specific elements, contributing to an increment of the *collocation points* inside the elements. If the discretized solution is sufficiently smooth inside the elements, further p-refinement can deliver exponential convergence.

In addition to concerns for high accuracy, NASA's CFD Vision 2030 Study points out that mesh generation and adaptivity are still significant bottlenecks in the CFD workflow [2]. This work seeks to improve parallel algorithms for mesh adaptivity for high order methods. Potent hp-refinement criteria are inherited from Mavriplis [53]. An *a posteriori error estimator* is used to indicate the smoothness of the current solutions. Then p-refinement is prescribed if the solution is smooth, otherwise h-refinement is applied. More details about the error estimator are disclosed in Section 3.3. The motivation to implement hp-adaptive AMR to our solver is to effectively diminish the computational error and fully exploit the resources of the HPCs.



(a) Conforming element interfaces: the element sizes and the polynomial orders are equal.



(b) Non-conforming element interfaces: the yellow element and the green element are geometrically non-conforming. The red one and the green one are functionally non-conforming. The yellow element and the red element are geometrically and functionally non-conforming.

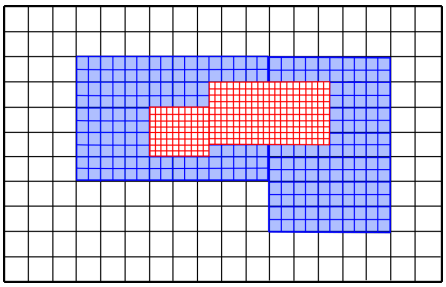
Figure 1.3: Conforming and non-conforming element interfaces illustration. Elements are in colour. The intersections of the black lines represent the collocation points within an element. In this example the yellow element has $4th$ order polynomials (*i.e.*, 5 points in each direction).

However, parallel AMR poses several significant challenges. First of all, the *hp-adaptivity* of the solver incurs a non-uniform mesh: adjacent elements could have different sizes and/or different polynomial orders. Such element interfaces are defined as *non-conforming interfaces* (Fig. 1.3b). To be more specific, the non-conforming interfaces could be *geometrically non-conforming*, *functionally non-conforming* or both. Since the exponential convergence of spectral discretizations relies on smooth solutions, the discontinuities introduced by non-conforming element interfaces can potentially disrupt the spectral convergence. To maintain the fast convergence, the *mortar element* method [52, 41] has been developed for both hyperbolic and elliptic problems. We have adopted the mortar element method from Ref. [42] to ensure the additional errors introduced by the discontinuities due to non-conforming interfaces remain as low as the spectral approximation errors, thereby retaining the exponential convergence. The derivation of the mortar element method can be found in Chapter 4.

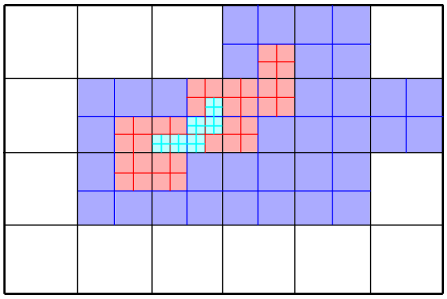
Secondly, grids with different resolution have different refinement levels. We define the grids with the same refinement level as having the same grid hierarchy. How to encode and store the hierarchical mesh is another tricky problem, as simple data structures cannot fulfill the requirements. Two popular AMR techniques have been studied and developed to tackle the hierarchical grid: *block-structured* AMR [10, 24, 51] and *tree-structured* AMR [16, 39].

Block-structured AMR uses finer uniform mesh "patches" on top of the coarser mesh to increase the resolution (Fig. 1.4a). Different patches of different mesh size represent different grid hierarchies. The same integration routines can be performed on different grid hierarchies. However, extra work needs to be done on different levels of patch interfaces, which complicates the AMR implementation. Additionally, the error estimator recursively accesses all (local) meshes from the coarsest to the finest to make the future refinement decision [9]. This recursive evaluation

process wastes time. Moreover, since adjustment to the mesh is performed with the minimum unit as a patch, it means we have less flexibility and control over the mesh resolution. This could result in patching unneeded areas.



(a) Block-structured AMR. Finer uniform meshes are patched on coarser meshes.



(b) Tree-structured AMR. Subdividing elements into smaller elements to improve resolutions.

Figure 1.4: Two AMR approaches. Colour schemes represent different hierarchies of mesh.

Unlike block-structured AMR, which uses an area of uniform fine mesh to increase the resolution, in tree-structured AMR, every element/cell in the mesh is a node in a tree-structure. We subdivide a cell into children cells to improve the resolution (Fig. 1.4b), which avoids the problem of overlapping cells in block-structured AMR. Therefore, the tree-structured AMR has higher flexibility in mesh refinement and coarsening than block-structured AMR. Cell subdivision is applied only at needed places, which makes efficient use of the computational storage. However, tree-structured AMR is a much more complicated data structure to implement and maintain. Standard grid-based solvers cannot be applied directly on a tree. The tree structure is built based on pointers between tree nodes, which consumes a considerable amount of computer memory [7]. Moreover, maintaining the tree structure in parallel, or performing tree structure operations like traversal, insertion and deletion presents challenges. For finite-volume related CFD applications which need frequent access to the element neighbour cells, fundamental tree structures are clumsy. Tree traversal needs to be performed to extract an element's (or a cell's) neighbour information. The tree traversal is hard to vectorize and parallelize. Therefore, with a dynamic mesh distributed among a large number of processors, the element neighbour query and tree structure update can be cumbersome processes. Nevertheless, some researchers have implemented the necessary operations needed to make this work. The finite element library *LibMesh* [71] replicates the global mesh on all processors to assist the tree operations. However, this strategy does not favour scalability of the program. Khokhlov developed a *fully threaded tree (FTT)* [39] which is a tree threaded in all possible directions. FTT offers a fast way to access neighbour, children and parent cells. The AMR library *p4est* [16] provides its own scalable algorithm to maintain and store the trees in parallel.

Instead of using these available approaches, we propose our own AMR approach: *a hash table*

AMR method. We utilize a hash table data structure to maintain and manage our AMR data. Hash table AMR has resolution flexibility similar to that of tree-structured AMR. The data is stored in an array manner but with a dynamic size. The hash table supports constant time item access, which is much more efficient than the common tree structure approach. Since we do not need to preserve the grid hierarchies by using pointers and the parent cells' data, we save memory over the tree-structured approach. Additionally, hash table AMR can be executed independently on distributed memory machines. Eliminating the task of maintaining mesh hierarchies, we are freed from the process of managing interprocessor tree operations. This feature contributes to the simplicity of our algorithm. We explore the idea of hash table AMR and its implementation in Section 5.2.

Another challenge of parallel AMR lies in the ever-changing element interfaces due to the adaptivity. The element interfaces include the interfaces among local (to the processor) elements as well as those shared with elements on remote processors across the *message passing interface (MPI)* boundaries. In order to perform the required solver computations, elements need to exchange data with their neighbour elements. This can be achieved relatively easily on a local process, but valid cross-processor communication presents a significant challenge. For an ordinary tree, an intuitive way is to use pointers that point to the parent, children and neighbour cells of an element and those pointers need to be updated every time the mesh changes. This is a tiresome process and the pointers consume considerable memory storage. FTT successfully cuts down the memory consumption by storing the four/eight children (for 2D/3D problems) generated after each subdivision in a contiguous memory slot, so that the data from the sibling neighbours can be extracted by memory address calculation. Thus, pointers between siblings are avoided, and siblings share pointers to their neighbour parent cells. Non-sibling neighbours can be found efficiently by traversing from their parents. FTT supports parallel modifications at any given tree level. Since neighbour cells are accessed through parent cells at a different level of the tree, any refinement or coarsening operation only affects the pointers related to the current sibling cells, *i.e.*, there is no change required in neighbouring cells located at the same level. *p4est* addresses the problem by building interconnection tables for element faces, edges and corners. It utilizes a unique way to number elements, and develops efficient algorithms to determine the owner process of a tree or node. To improve the efficiency of neighbour finding algorithms, p4est restricts a *2:1 balance* [72] to the neighbour size relation. In our work, we also introduce a unique way of element numbering. We use the unique numbers known as "*keys*" to access elements in the hash table. The nature of the hash table empowers us to access element neighbours in constant time complexity with the explicitly stored neighbour keys. Following the element numbering mechanics we propose, one element can deduce its possible neighbour keys in one operation. This simplifies the interprocessor boundaries update problem. We explain the whole algorithm in detail in Chapter 6.

Last but not least, the adaptivity of the solver introduces load imbalance to the solver. Load imbalance is one of the major impedances to high-level parallelism. For applications that have

dynamic workload during runtime, the load imbalance among the HPCs' cores degrades overall performance. To accelerate the simulation and improve the level of parallelism, *repartitioning* algorithms that redistribute the workload evenly among computational units are desired. The workload partition problem is an optimization problem that requires the workload to be distributed evenly among the processors, while minimizing the cross-processors boundaries between partitions. The partition problem is NP-hard [14]. *Graph-based* repartitioning techniques have been long-studied and tackle this problem pretty well. They treat every element in the mesh as a node, and the connections among elements as edges. Therefore, the whole mesh can be treated as a graph and the partition problem becomes one of cutting the graph into the desired number of portions that ideally contain equal numbers of nodes with minimum edge cuts. Many libraries are developed based on graph-based theories, for example, *ParMetis* [71], *Scotch* [60] and *Party* [56]. These libraries do a good job, but have a limited scalability. Since graph-based algorithms require global information of the whole graph, the memory consumption grows linearly with the problem size. Thus, their scalabilities are restrained by the memory size of the distributed memory systems.

Another approach that has garnered the attention of researchers is that of *space-filling curves (SFCs) based* repartitioning algorithms. In this type of partition algorithms a curve is woven through all the elements in the mesh thereby reducing the multidimensional partition problem into a one-dimensional one, which largely simplifies the problem. The compactness, or the so-called good *locality*, of the SFCs ensures a "good" partition interface. In this work, a SFC-based repartitioning algorithm is developed to alleviate the load imbalance during the solver's execution and improve the performance of the solver. The SFC we used is called the *Hilbert curve*. It has the best locality in comparison to other space-filling curves, which helps to minimize the interprocessor communications. We present algorithms to generate a Hilbert curve in parallel in Section 7.3. In Section 7.4, a distributed load balancing algorithm is then presented. In this algorithm, the repartitioning strategies can be devised independently on each processor, which is an important feature for a scalable algorithm. The load balancing algorithm is designed with the following objectives: (1) fast execution time, (2) high-quality repartitioning results and (3) low memory overhead.

We present the scalability of our load balancing algorithm and the overall performance of our application in Chapter 8. The experiments are performed on supercomputers *Graham* and *Cedar* offered by *Compute Canada*. Scalability tests are done with millions of elements on up to $16,384$ processors. The quality of the load balancing results is evaluated.

In summary, we combine a state-of-the-art high-order method with effective adaptive mesh refinement techniques to sharpen CFD simulation capabilities, while using dynamic load balancing to push the most powerful computers currently available to their capacity limits. This project represents a significant step in the development and popularization of *direct numerical simulation (DNS)*, which even goes beyond the scope of NASA's CFD Vision 2030: coupling DNS with AMR while making full use of the valuable computational resources.

This thesis is organized as follows. In Chapter 2, we present the derivation of DG-SEM and set up our application problem. In the next chapter (Chapter 3), AMR approaches are introduced and our refinement techniques are explained. How to tackle non-conforming element interfaces in the spectral element method is presented in Chapter 4. Our proposed hash table AMR implementation is described in Chapter 5. How to update interprocessor boundaries is explained in detail in Chapter 6. Next, the load balancing algorithms are discussed and presented in Chapter 7. Performance results from two different supercomputer platforms are showcased in Chapter 8. Finally, we draw our conclusions and discuss future work in Chapter 9.

Chapter 2

Discontinuous Galerkin Spectral Element Method

In this chapter, we detail the formulation of the discontinuous Galerkin spectral element method (DG-SEM). To do so, we first introduce finite element methods in Section 2.1 and spectral methods in Section 2.2, before proceeding to the DG-SEM formulation in Section 2.3. DG-SEM is applied on an acoustic wave equation on a two-dimensional domain. A benchmark solution is presented at the end of this chapter.

2.1 Finite Element Approximation

The spectral element method is based on the same principles as the *finite element method (FEM)*. To better explain the spectral element method, we first introduce some essential concepts in FEM.

2.1.1 Evaluate the Weak Form of the Problem

In this section, we use the Poisson equation as an example to explain the mechanisms behind the FEM. In a fixed Cartesian coordinate system, we denote the coordinates x, y, z in index form as x_1, x_2, x_3. Similarly, the displacement will be written as u, v, w or u_1, u_2, u_3. In the following work, we write all the quantities' coordinates and displacements as x_l and u_l, where the index l is 1, 2, 3 for three-dimensional applications, or 1, 2 for two-dimensional applications.

We take the Poisson equation as:

$$\sum_{l=1,2,3} \frac{\partial^2 u}{\partial x_l^2} + f = 0, \tag{2.1}$$

where u is the solution function of x_l $(l = 1, 2, 3)$. x are the coordinates (in the following text we drop the subscript for simplicity) which define the domain Ω, on which the problem is posed. f is a specified function.

With appropriate boundary conditions, for example, Dirichlet boundary conditions :

$$u = u_D, \quad on\ \Gamma, \tag{2.2}$$

where Γ is the physical boundary of the domain Ω, the solution is defined uniquely. Equation 2.1 and Equation 2.2 are known as the *strong form* of the problem.

The finite element method approximates the solution function by another function in a finite dimensional function space that we choose. We want to approximate the solution function $u(x)$ by $u_h(x)$, which is defined in a finite-dimension function space X_h, *i.e.*, $u_h(x) \in X_h$. Function space X_h has a basis: $\{\phi_1(x), \phi_2(x), \cdots, \phi_N(x)\}$, where the $\phi_i(x)$ $(i = 1, \cdots, N)$ are called the *basis functions*. Any function $g(x)$ in space X_h can be expressed as a linear combination of the basis functions:

$$g(x) = \sum_{i=1}^{N} a_i \phi_i(x), \quad g(x) \in X_h, \tag{2.3}$$

where a_i are scalars.

Thus, our goal is to find a solution $u_h(x) \in X_h$, such that, $\frac{\partial^2 u_h(x)}{\partial^2 x} + f(x)$ is as close to 0 as possible.

To directly deal with the strong form of the problem requires us to compute the second derivative of the solution function. This requirement can be weakened by considering an integral expression:

$$\int_\Omega \left(\frac{\partial^2 u_h(x)}{\partial^2 x} + f(x) \right) v(x) d\Omega = 0, \tag{2.4}$$

where Ω represents the domain and v is an arbitrary function.

Since v is an arbitrary function, by requiring Equation 2.4 to hold for any v, we are forcing the $\left(\frac{\partial^2 u_h(x)}{\partial^2 x} + f(x) \right)$ term to be zero. Then we integrate the second derivative term by parts in Equation 2.4 and obtain:

$$\int_\Gamma \frac{\partial u}{\partial x}\, v\, \overrightarrow{n} \cdot d\overrightarrow{\Gamma} - \int_\Omega \frac{\partial u}{\partial x} \frac{\partial v}{\partial x}\, d\Omega + \int_\Omega f v\, d\Omega = 0, \tag{2.5}$$

where $\vec{n}$ denotes the unit vector pointing outwards from the boundary Γ.

Equation 2.5 is the so-called *weak form* of the problem. In the weak form, only first-order derivatives are necessary in constructing a solution. The weak form is the basis for the finite element approximation [83].

2.1.2 Weighted Residual Approximation

Next, we approximate the solution in a weighted residual manner. Recall u_h is in function space X_h, which can be represented as:

$$u \approx u_h = \sum_{i=1}^{N} \widetilde{u}_i \phi_i, \tag{2.6}$$

where ϕ_i are the basis functions and $\widetilde{u}_i$ are the unknown coefficients.

Similarly, we can express the arbitrary function v as:

$$v \approx v_h = \sum_{j=1}^{M} \widetilde{v}_j \varphi_j, \tag{2.7}$$

where v_h is the approximation of v in finite function space X_m with the basis functions φ_j ($j = 1, \cdots, M$). In Equation 2.4, we call the function set φ_j the *test functions*. $\widetilde{v}_j$ are the arbitrary parameters.

In the finite element method, the test and basis functions are defined in a local manner. One can consider each of the test and basis functions to be defined in partitions Ω_e of the total domain Ω. Each partition Ω_e is known as an *element*. The sum of the elements forms the total domain:

$$\Omega \approx \Omega_h = \bigcup \Omega_e. \tag{2.8}$$

The elements are lines in 1D, triangles or rectangles in 2D and tetrahedra or hexahedra in 3D. In the finite element approximation, we divide the modal body into equivalent system of many smaller elements interconnected at points (nodal points) common to two or more elements. The basis functions are usually linear functions in each element and the unknown coefficients are the *nodal values* of ϕ.

The finite element method approximates the solution by using weighted residuals. By substituting the approximated solution u_h inside the governing equation, we obtain the residual R as

$$R = \sum_{i=1}^{N} \widetilde{u}_i \frac{\partial^2 \phi_i}{\partial x^2} + f, \tag{2.9}$$

which should be zero for the exact solution.

Then we seek for the best nodal values $\widetilde{u}_i$ that satisfy:

$$\int_{\Omega} \varphi_j R \, d\Omega = 0, \quad j = 1, \cdots, M, \tag{2.10}$$

where φ_j are the weighting functions.

We again apply integration by parts to reduce the high-order derivatives to first-order derivatives. In the present case, the weighted residual approximation becomes:

$$\int_{\Gamma} \left(\sum_{i=1}^{N} \widetilde{u}_i \frac{\partial \phi_i}{\partial x} \right) \varphi_j \, \vec{n} \cdot d\overrightarrow{\Gamma} - \int_{\Omega} \frac{\partial \varphi_j}{\partial x} \left(\sum_{i=1}^{N} \widetilde{u}_i \frac{\partial \phi_i}{\partial x} \right) d\Omega + \int_{\Omega} f \varphi_j \, d\Omega = 0, \quad j = 1, \cdots, M. \tag{2.11}$$

If the arbitrary function v is defined in the same function space X_h as the approximated solution u_h, *i.e.*, they share the same basis functions and hence $\phi_i = \varphi_i$, then this type of approximation is called the *Galerkin method*. Therefore, in the Galerkin approach, Equation 2.11 becomes:

$$\int_\Gamma \left(\sum_{i=1}^{N} \widetilde{u}_i \frac{\partial \phi_i}{\partial x} \right) \phi_j \, \overrightarrow{n} \cdot d\overrightarrow{\Gamma} - \int_\Omega \frac{\partial \phi_j}{\partial x} \left(\sum_{i=1}^{N} \widetilde{u}_i \frac{\partial \phi_i}{\partial x_l} \right) d\Omega + \int_\Omega f \phi_j \, d\Omega = 0, \quad j = 1, \cdots, N. \tag{2.12}$$

Equation 2.12 forms a linear system, where we can solve for n unknowns $\widetilde{u}_i$ with n equations.

2.2 Spectral Approximation

In this section, we proceed to the spectral element method. As we did in FEM, we approximate the solution by a linear combination of basis functions in a finite function space. However, for the spectral element approximation, instead of using piece-wise linear basis functions, we choose high-order polynomials to increase the order of accuracy of the approximation. The idea of spectral approximation is to represent the solution of a differential equation as a finite weighted sum of orthogonal functions, such as Legendre polynomials or Chebyshev polynomials. The spectral method is usually implemented with either the collocation or the Galerkin approach, the latter of which uses the polynomials as test functions as well. The Galerkin spectral element method combines the advantages of spectral methods, i.e. high accuracy and fast convergence, with those of Galerkin finite element methods, i.e. geometrical flexibility, to form a powerful method for resolution of partial differential equations modelling complex phenomena.

2.2.1 Functions of Approximation

The spectral method is used to approximate the solution of a differential equation in the form of a finite sum of weighted orthogonal basis functions. For a function $f(x)$, we discretize it in finite basis function form as

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty} a_n \phi_n(x) = \sum_{n=0}^{N} a_n \phi_n(x) + \sum_{n=N+1}^{\infty} a_n \phi_n(x) \\ &= \sum_{n=0}^{N} a_n \phi_n(x) + \tau \\ &\approx \sum_{n=0}^{N} a_n \phi_n(x). \end{aligned} \tag{2.13}$$

We extend the function $f(x)$ into an infinite sum of orthogonal basis functions $\phi_n(x)$ multiplied by coefficients a_n. To approximate the solution, we only calculate the infinite sum up to $(N+1)$ terms, thereby truncating the infinite series. Thus, we call the τ in Equation 2.13 the *truncation error*. The size of the error depends on how fast the coefficients a_n decay to zero. Therefore, we want the coefficients to decay as fast as possible. Mathematical proof has been

given: for certain basis polynomial series, for instance, Fourier series, Legendre polynomials and Chebyshev polynomials, the coefficients decay exponentially fast, which is the so-called *spectral accuracy* [40]. High-order methods such as these, have been used almost exclusively in the direct numerical simulation of turbulent flow in the last two decades. With the high-order discretization of the solution function and the fast convergence, the spectral method is able to fully resolve the details of turbulent flows in simple geometries.

2.2.2 Polynomial Basis Functions

The spectral approximation is based on a series of orthogonal polynomials to be used as basis functions. On the domain interval $[a, b]$, the basis functions are orthogonal to each other, so that:

$$\int_a^b \phi_i(x)\phi_j(x)dx = C_i\delta_{ij}, \tag{2.14}$$

where C_i is a constant. δ_{ij} is the Kronecker delta function and has the property:

$$\delta_{ij} = \begin{cases} 1, & if\ \ i = j, \\ 0, & if\ \ i \neq j. \end{cases} \tag{2.15}$$

Equation 2.14 can be written in inner product form. Let V be a vector space over $\mathbb{R}$, the inner product of continuous functions in $[a, b]$, denoted as $V = L^2(a, b)$, is defined as: given two arbitrary vectors $f(x)$ and $g(x)$, the inner product between them is:

$$(f, g) = \int_a^b f(x)g(x)dx. \tag{2.16}$$

Thus, we rewrite Equation 2.14 as

$$(\phi_i, \phi_j) = \int_a^b \phi_i(x)\phi_j(x)dx = C_i\delta_{ij}. \tag{2.17}$$

In spectral approximation, we choose high-order orthogonal polynomials as the basis functions. The polynomials form an orthogonal basis for square integrable functions $L^2_w(a, b)$, where w is the corresponding weight of the function. For orthogonal polynomials, we have:

$$(\phi_i, \phi_j) = \int_a^b \phi_i(x)\phi_j(x)w(x)dx = D_{ij}\delta_{ij}, \tag{2.18}$$

where D_{ij} is a constant. The polynomials are orthogonal to each other with respect to a weight function $w(x)$.

For non-periodic problems, two popular polynomial sets that fulfill such requirements are *Legendre polynomials* and *Chebyshev polynomials*. Legendre polynomials have the feature that the weight function is a constant: $w(x) = 1$. Therefore, the constant weight makes integrals easier to compute analytically. In this work, we adopt Legendre polynomials as our basis functions.

Legendre Polynomials

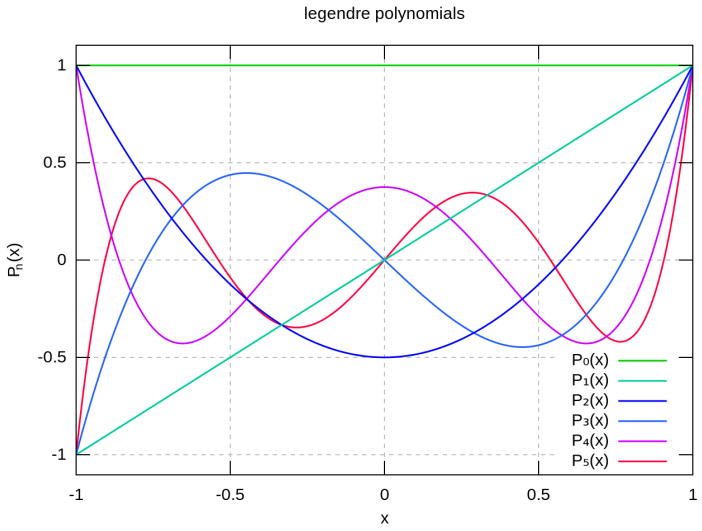


Figure 2.1: Legendre polynomials $P_n(x)$ of orders n ranging from 0 to 5.

Legendre polynomials were discovered in 1782 by Adrien-Marie Legendre. They form a system of complete and orthogonal polynomials. With a weight function of $w(x) = 1$, over the interval $[-1, 1]$, $L_k(x)$ is the Legendre polynomial of degree k. It has the property:

$$\int_{-1}^{1} L_{k_1}(x) L_{k_2}(x) dx = 0, \quad if \;\; k_1 \neq k_2. \tag{2.19}$$

It can be generated by using the three-term recurrence relation

$$L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x). \tag{2.20}$$

We show the first six Legendre polynomials in Fig. 2.1. With the knowledge of $L_0(x) = 1$ and $L_1(x) = x$, larger order Legendre polynomials can be derived by using the recursive equation 2.20.

The derivatives of Legendre polynomials satisfy

$$(2k+1) L_k(x) = L'_{k+1}(x) - L'_{k-1}(x). \tag{2.21}$$

2.2.3 Gauss Quadrature

In order to evaluate an integral numerically, Gauss quadrature is invoked. A quadrature rule is an approximation of the defined integral of a function, and Gauss quadrature offers high precision. The most common interval is taken as $[-1, 1]$ and the quadrature rule is

$$\int_{-1}^{1} f(x) dx \approx \sum_{i=0}^{N} w_i f(x_i). \tag{2.22}$$

By choosing appropriate points x_i in the interval $[-1, 1]$, known as the Gauss quadrature points, we can tailor the accuracy of this approximation. Using the roots of the $(N+1)st$ Legendre polynomial for these points, the approximation is exact for polynomials $f(x)$ of degree 2N + 1 or less. This rule is known as the *Gauss-Legendre quadrature rule*. The Gauss-Legendre quadrature does not include the endpoints on the interval $[-1, 1]$. To get the abscissas x_i and the weights w_i, the roots of $(N+1)st$ Legendre polynomial are needed.

$$x_i = roots\ of\ L_{N+1}(x), \quad i = 0, ..., N, \tag{2.23}$$

$$w_i = \frac{2}{(1 - x_i^2)\left[L'_{N+1}(x_i)\right]^2}. \tag{2.24}$$

If one wants to use the endpoints on the interval as the integration points, there is another option called *Gauss-Lobatto quadrature*. However, Gauss-Lobatto quadrature is less accurate than Gauss-Legendre quadrature. It can evaluate the function integral exactly for polynomials of degree $2N - 1$ or less. Similarly, the Gauss-Lobatto points can be calculated as

$$x_i = \pm 1,\ roots\ of\ L'_N(x), \quad j = 0, ..., N, \tag{2.25}$$

$$w_i = \frac{2}{N(N+1)} \frac{1}{[L_N(x_i)]^2}. \tag{2.26}$$

We choose to implement Gauss-Legendre quadrature in this work for better precision.

2.2.4 Polynomial Interpolation in Lagrange Form

In numerical analysis, polynomial interpolation is a technique that interpolates a set of data points using a lowest-degree polynomial that passes through all the data points [65]. A common polynomial basis is *Lagrange interpolating polynomials*:

$$l_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x - x_i}{x_j - x_i}, \tag{2.27}$$

where l_j is the Lagrange polynomial of degree N. For each x_i we have

$$l_j(x_i) = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j, \end{cases} \tag{2.28}$$

where δ_{ij} is the *Kronecker delta*.

With this feature, we can interpolate a function $p_N(x)$ of degree N in discrete form

$$p_N(x) = \sum_{j=0}^{N} p(x_j) l_j(x). \tag{2.29}$$

There is an alternative way to write the Lagrange interpolation, which is called *barycentric form* [40], where we modify the Lagrange interpolation as follows

$$p_N(x) = \psi(x) \sum_{j=0}^{N} p(x_j) \frac{w_j}{x - x_j}, \tag{2.30}$$

$$\psi(x) = \prod_{i=0}^{N} (x - x_i), \tag{2.31}$$

$$w_j = \frac{1}{\prod_{\substack{i=0 \\ i \neq j}}^{N} (x_j - x_i)}. \tag{2.32}$$

The modified Lagrange interpolation still preserves the property

$$\psi(x) \sum_{j=0}^{N} \frac{w_j}{x - x_j} = 1. \tag{2.33}$$

Then we rewrite the Lagrange interpolation in barycentric form

$$p_N(x) = \frac{\sum_{j=0}^{N} p(x_j) \frac{w_j}{x - x_j}}{\sum_{j=0}^{N} \frac{w_j}{x - x_j}}. \tag{2.34}$$

We can pre-compute the weights w_j and store them. Then whenever we need to calculate the interpolant at point x, we can quickly calculate it by using the weights and the set of $p(x_j)$.

2.3 Approximation of Wave Equations

The Galerkin spectral element method starts with the Galerkin formulation, then the integrals of Equation 2.12 are calculated by Gauss quadrature. Two types of Galerkin spectral element methods are used in practise: *continuous Galerkin spectral element method (CG-SEM)* and *discontinuous Galerkin spectral element method (DG-SEM)*. A distinct difference between these two types of method is that in CG-SEM, the solution approximation has to satisfy the inter-element continuity conditions, therefore, the approximated solutions are continuous at adjacent element interfaces, whereas for DG-SEM, the inter-element continuity conditions are weakly enforced on the boundaries, *i.e.*, the boundary conditions do not need to be satisfied exactly, then we obtain a "discontinuity" at adjacent element interfaces.

Taking the advection equation as an example:

$$u_t + u_x = 0,\ x \in (-1, 1), \tag{2.35}$$

$$u(x, 0) = u_0(x),\ x \in [-1, 1], \tag{2.36}$$

$$u(-1, t) = g(x),\ t > 0. \tag{2.37}$$

We write the Equation 2.35 in its weak form:

$$(u_t, v) + (u_x, v) = 0, \tag{2.38}$$

where v are the test functions. For CG-SEM, the test functions are required to satisfy the inter-element continuity conditions.

The solution can be approximated as the weighted sum of polynomials:

$$u(x,t) \approx u_h(x,t) = \sum_{j=0}^{N} a_j(t) L_j(x) = \sum_{j=0}^{N} \widetilde{u}_j(t) l_j(x), \tag{2.39}$$

where $L_j(x)$ and a_j are the Legendre polynomials and their coefficients. Then we express the approximation in Lagrange form ($l_j(x)$). The $\widetilde{u}_j$ are the coefficients of the Lagrange polynomials. Since the boundary condition needs to be satisfied exactly for CG-SEM, we set $\sum_{j=0}^{N} \widetilde{u}_j(t) l_j(-1) = g(x)$. Whereas for DG-SEM, when we apply integration by parts to the second term in Equation 2.38 we get:

$$\begin{aligned} (u_x, v) &= [uv]_{-1}^{1} - (u, v_x) \\ &= [u(1,t)v(1) - g(x)v(-1)] - (u, v_x)\,, \end{aligned} \tag{2.40}$$

where we can naturally enforce the boundary conditions.

The DG-SEM is particularly well-suited for wave propagation problems. In this work, we apply the DG-SEM method to the solution of a two-dimensional acoustic wave equation. We constrain our domain to a square. In this context, we will explain and derive the DG-SEM approximation and present its performance with our benchmark solution.

2.3.1 System of Equations

We solve an acoustic wave equation in the form of

$$\frac{\partial^2 p}{\partial t^2} - c^2(p_{xx} + p_{yy}) = 0, \tag{2.41}$$

$$u_t = -p_x, \tag{2.42}$$

$$v_t = -p_y. \tag{2.43}$$

In the equations above, p represents the pressure, u and v are the velocity components in two directions. c is the sound speed. We can combine the three equations above into one:

$$\frac{\partial^2 p}{\partial t^2} + c^2\left((u_x)_t + (v_y)_t\right) = 0. \tag{2.44}$$

We integrate Equation 2.44 with respect to time t and get

$$p_t + c^2\left(u_x + v_y\right) = 0. \tag{2.45}$$

We then rewrite the system of equations in matrix-vector form:

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_x + \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_y = 0, \tag{2.46}$$

which can also be written in vector form, as

$$\mathbf{q}_t + A_x\mathbf{q}_x + A_y\mathbf{q}_y = 0, \tag{2.47}$$

where A_x and A_y are constant matrices shown in Equation 2.46. We use bold font to represent vectors. $\mathbf{q}$ can be comprehended as the vector of the three components we solve for, namely pressure and two velocity components. Since A_x and A_y are constant, we can combine them with the derivatives and obtain

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0, \tag{2.48}$$

where $\mathbf{f}_x = A_x\mathbf{q}_x$ and $\mathbf{g}_y = A_y\mathbf{q}_y$. Then we obtain the wave equation in *conservation law* form

$$\mathbf{q}_t + \bigtriangledown \cdot \mathfrak{F} = 0, \tag{2.49}$$

where $\mathfrak{F} = \mathbf{f}\widehat{x} + \mathbf{g}\widehat{y}$ is the flux vector. $\widehat{x}$ and $\widehat{y}$ are the unit vectors in x and y directions.

Next we apply the divergence theorem to the integral of the conservation law. We obtain

$$\frac{d}{dt}\int_V \mathbf{q}dV = -\int_S \mathfrak{F}\cdot\widehat{n}dS. \tag{2.50}$$

where V is the control volume and S is the volume surface. $\widehat{n}$ is an outward-pointing normal vector.

In the next section, we will take the Galerkin approximation of this integral equation and use Gauss quadrature to evaluate it.

2.3.2 Discontinuous Galerkin Approximation

Apply DG-SEM to One Element

In this section, we present the derivation of the discontinuous Galerkin approximation. We first start with the simplest case: one element on the 2D reference space $[-1, 1] \times [-1, 1]$.

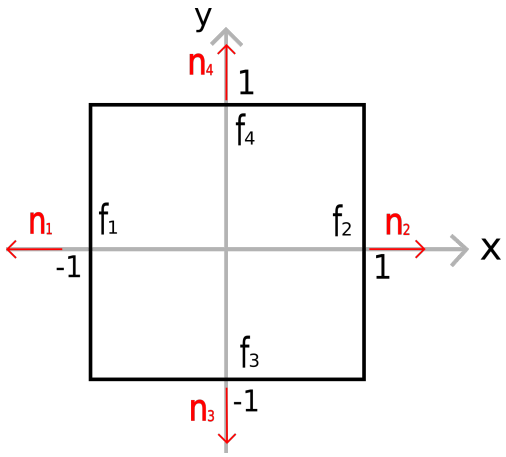


Figure 2.2: 2D reference domain: $[-1,1]\times[-1,1]$. f_i $(i=1,...,4)$ are the four element boundaries and $\mathbf{n_i}$ are the unit vectors pointing outwards from the element boundaries.

The reference domain is shown in Fig. 2.2, with four boundaries f_i $(i=1,...,4)$ and four outward normal vectors $\mathbf{n}_i$.

We discretize our approximated solutions by polynomials of degree N in the x direction and degree M in the y direction, then present them in Lagrange form:

$$\mathbf{q}\approx\mathbf{Q}=\sum_{i=0}^{N}\sum_{j=0}^{M}\mathbf{Q}_{i,j}l_i(x)l_j(y) \tag{2.51}$$

$$\mathfrak{F}\approx\mathbf{F}=\sum_{i=0}^{N}\sum_{j=0}^{M}\left(\mathbf{F}_{i,j}\widehat{x}+\mathbf{G}_{i,j}\widehat{y}\right)l_i(x)l_j(y), \tag{2.52}$$

where $\mathbf{F}_{i,j}\widehat{x}+\mathbf{G}_{i,j}\widehat{y}=A_x\mathbf{Q}_{i,j}\widehat{x}+A_y\mathbf{Q}_{i,j}\widehat{y}$.

The weak form of Equation 2.49 is

$$(\mathbf{q}_t,v)+(\bigtriangledown\cdot\mathfrak{F},v)=0, \tag{2.53}$$

where v are the test functions, and we also present them in Lagrange form:

$$v=\sum_{i=0}^{N}\sum_{j=0}^{M}\widetilde{v}_{i,j}l_i(x)l_j(y). \tag{2.54}$$

Plugging Equation 2.54 into the weak form of conservation law we obtain:

$$\sum_{i=0}^{N}\sum_{j=0}^{M}\left[\int_\Omega\mathbf{q}_t l_i(x)l_j(y)dxdy+\int_\Omega\left(\bigtriangledown\cdot\mathfrak{F}\right)l_i(x)l_j(y)dxdy\right]\widetilde{v}_{i,j}=0, \tag{2.55}$$

where Ω denotes the whole domain.

Since $\widetilde{v}_{i,j}$ are arbitrary, the coefficients before $\widetilde{v}_{i,j}$ must vanish. Thus, we have:

$$\int_\Omega \mathbf{q}_t l_i(x) l_j(y) dxdy + \int_\Omega (\bigtriangledown \cdot \mathfrak{F})\, l_i(x) l_j(y) dxdy = 0, \quad i = 0, ..., N,\ j = 0, ...M \tag{2.56}$$

$$\Rightarrow \int_\Omega \frac{d\mathbf{Q}}{dt} \phi_{i,j} dxdy + \int_\Omega \bigtriangledown \cdot \mathbf{F} \phi_{i,j} dxdy = 0, \quad i = 0, ..., N,\ j = 0, ...M, \tag{2.57}$$

where $\phi_{i,j} = l_i(x) l_j(y)$.

Equation 2.57 can be written in inner product form:

$$(\mathbf{Q}_t, \phi_{i,j}) + (\bigtriangledown \cdot \mathbf{F}, \phi_{i,j}) = 0. \tag{2.58}$$

We then separate the boundary contributions from the interior contributions in the $(\bigtriangledown \cdot \mathbf{F}, \phi_{i,j})$ term by using Green's first identity. Green's first identity is:

$$\int_\Omega \bigtriangledown \cdot (\phi \mathbf{X})\, d\Omega = \int_\Omega \bigtriangledown \phi \cdot \mathbf{X} + \phi \bigtriangledown \cdot \mathbf{X} d\Omega = \oint_S \phi \mathbf{X} \cdot \widehat{n} dS, \tag{2.59}$$

where Ω represents the multidimensional domain and $S = \partial\Omega$ is the boundary of the domain. ϕ is a scalar function and $\mathbf{X}$ is a vector field. $\widehat{n}$ is the boundary unit vector pointing outwards. Applying this theorem to the flux term in Equation 2.58 we obtain:

$$(\bigtriangledown \cdot \mathbf{F}, \phi_{i,j}) = \int_{-1}^{1} \int_{-1}^{1} \phi_{i,j} \bigtriangledown \cdot \mathbf{F} dxdy = \oint_S \phi_{i,j} \mathbf{F} \cdot \widehat{n} dS - \int_{-1}^{1} \int_{-1}^{1} \mathbf{F} \cdot \bigtriangledown \phi_{i,j} dxdy. \tag{2.60}$$

The element boundary conditions are weakly enforced in the discontinuous Galerkin method. We use the boundary fluxes $\mathbf{F}^*$ to impose the boundary conditions. The boundary fluxes will be explained later. We first supplant the boundary conditions by the fluxes in Equation 2.60. Thus, the whole approximation becomes:

$$(\mathbf{Q}_t, \phi_{i,j}) + \oint_S \phi_{i,j} \mathbf{F}^* \cdot \widehat{n} dS - \int_{-1}^{1} \int_{-1}^{1} \mathbf{F} \cdot \bigtriangledown \phi_{i,j} dxdy = 0. \tag{2.61}$$

Then different approaches are used to evaluate the three portions of the integrals in Equation 2.61. We first tackle the first and third integrals since they can be computed similarly by using Gauss quadrature. Then the boundary fluxes in the second term will be explained. We distinguish element interior solutions and boundary fluxes by using different types of nodes in Fig. 2.3.

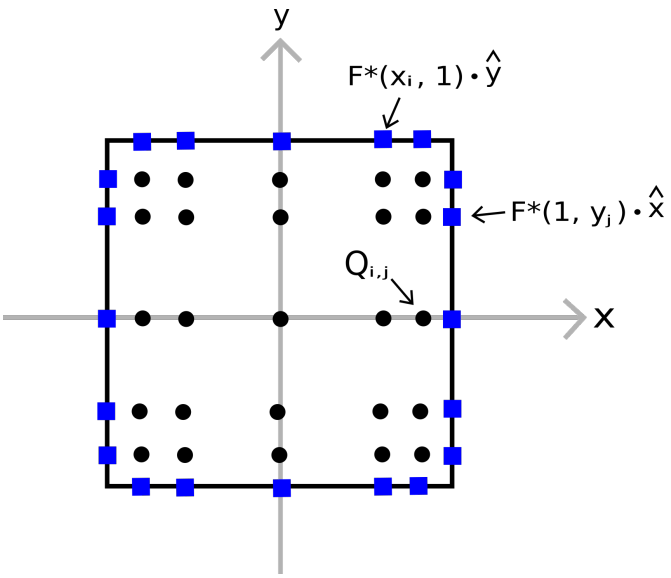


Figure 2.3: Solution nodes and boundary flux nodes. Black round nodes are the Gauss-Legendre collocation nodes where solutions are approximated. The blue square nodes are boundary flux nodes where element fluxes are approximated. They are the Gauss-Legendre collation nodes placed on the boundaries.

The black round nodes in Fig. 2.3 are the Gauss-Legendre collocation nodes which are distributed unevenly inside an element and not on the boundaries. The solutions are approximated on these nodes. The blue square nodes are the boundary flux nodes, where element fluxes are evaluated. Boundary flux nodes are the Gauss-Legendre collation nodes projected on the boundaries.

We first approximate the first term $(\mathbf{Q}_t, \phi_{i,j})$ in Equation 2.61:

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t\phi_{i,j}dxdy &= \int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t l_i(x)l_j(y)dxdy \\ &= \sum_{k=0}^{N}\sum_{l=0}^{M}\frac{d\mathbf{Q}\left(x_k,y_l\right)}{dt}l_i(x_k)l_j(y_l)w_k^{(x)}w_l^{(y)},\end{aligned} \tag{2.62}$$

where we have applied Gauss quadrature to the first term. Note that since the integrand is a polynomial of degree $2N$, the integral can be calculated exactly. Based on the fact that $l_i(x_k) = \delta_{ik}$, the equation above can be simplified to

$$\int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t\phi_{i,j}dxdy = \frac{d\mathbf{Q}\left(x_i,y_j\right)}{dt}w_i^{(x)}w_j^{(y)}. \tag{2.63}$$

Then we evaluate the third term in Equation 2.61 since the process is similar to the first one.

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\mathbf{F}\cdot\bigtriangledown\phi_{i,j}dxdy &= \int_{-1}^{1}\int_{-1}^{1}\mathbf{F}(x,y)l_i'(x)l_j(y)+\mathbf{G}(x,y)l_i(x)l_j'(y)\ dxdy \\ &= \sum_{k=0}^{N}\sum_{l=0}^{M}\left[\mathbf{F}(x_k,y_l)l_i'(x_k)l_j(y_l)+\mathbf{G}(x_k,y_l)l_i(x_k)l_j'(y_l)\right]w_k^{(x)}w_l^{(y)} \\ &= \sum_{k=0}^{N}\mathbf{F}(x_k,y_j)l_i'(x_k)w_k^{(x)}w_j^{(y)}+\sum_{l=0}^{M}\mathbf{G}(x_i,y_l)l_j'(y_l)w_i^{(x)}w_l^{(y)}.\end{aligned} \tag{2.64}$$

Finally, we evaluate the second term in Equation 2.61, which is the boundary integral:

$$\begin{aligned}\oint_S\phi_{i,j}\mathbf{F}^*\cdot\widehat{n}dS = &\int_{-1}^{1}l_i(x)l_j(-1)\mathbf{F}^*(x,-1)\cdot(-\widehat{y})\,dx \\ &+\int_{-1}^{1}l_i(x)l_j(1)\mathbf{F}^*(x,1)\cdot(\widehat{y})\,dx \\ &+\int_{-1}^{1}l_i(-1)l_j(y)\mathbf{F}^*(-1,y)\cdot(-\widehat{x})\,dy \\ &+\int_{-1}^{1}l_i(1)l_j(y)\mathbf{F}^*(1,y)\cdot(\widehat{x})\,dy.\end{aligned} \tag{2.65}$$

We again apply Gauss quadrature to the integrals. The integrands are still polynomials of $2N$, so the integrals can be computed exactly.

$$\begin{aligned}\oint_S\phi_{i,j}\mathbf{F}^*\cdot\widehat{n}dS = &\sum_{k=0}^{N}l_i(x_k)l_j(-1)\mathbf{F}^*(x_k,-1)\cdot(-\widehat{y})\,w_k^{(x)} \\ &+\sum_{k=0}^{N}l_i(x_k)l_j(1)\mathbf{F}^*(x_k,1)\cdot(\widehat{y})\,w_k^{(x)} \\ &+\sum_{l=0}^{M}l_i(-1)l_j(y_l)\mathbf{F}^*(-1,y_l)\cdot(-\widehat{x})\,w_l^{(y)} \\ &+\sum_{l=0}^{M}l_i(1)l_j(y_l)\mathbf{F}^*(1,y_l)\cdot(\widehat{x})\,w_l^{(y)} \\ =&l_j(-1)\mathbf{F}^*(x_i,-1)\cdot(-\widehat{y})\,w_i^{(x)} \\ &+l_j(1)\mathbf{F}^*(x_i,1)\cdot(\widehat{y})\,w_i^{(x)} \\ &+l_i(-1)\mathbf{F}^*(-1,y_j)\cdot(-\widehat{x})\,w_j^{(y)} \\ &+l_i(1)\mathbf{F}^*(1,y_j)\cdot(\widehat{x})\,w_j^{(y)}.\end{aligned} \tag{2.66}$$

We combine the approximation of the three integrals together and divide them by $w_i^{(x)}w_j^{(y)}$ to

obtain

$$\begin{aligned}\frac{d\mathbf{Q}(x_i,y_j)}{dt} &+ \left\{\left[l_i(-1)\mathbf{F}^*(-1,y_j)\cdot(-\widehat{x})\frac{1}{w_i^{(x)}} + l_i(1)\mathbf{F}^*(1,y_j)\cdot(\widehat{x})\frac{1}{w_i^{(x)}}\right] - \sum_{k=0}^{N}\mathbf{F}(x_k,y_j)\frac{l_i'(x_k)w_k^{(x)}}{w_i^{(x)}}\right\} \\ &+ \left\{\left[l_j(-1)\mathbf{F}^*(x_i,-1)\cdot(-\widehat{y})\frac{1}{w_j^{(y)}} + l_j(1)\mathbf{F}^*(x_i,1)\cdot(\widehat{y})\frac{1}{w_j^{(y)}}\right] - \sum_{l=0}^{M}\mathbf{G}(x_i,y_l)\frac{l_j'(y_l)w_l^{(y)}}{w_j^{(y)}}\right\} \\ &= 0, \quad i=0,...,N. \quad j=0,..,M.\end{aligned} \tag{2.67}$$

The $l_i'(x_k)$ is defined as *derivative matrix* $D_{k,i} = l_i'(x_k)$. To simply the derivative terms in Equation 2.67, we take

$$\widehat{D}_{i,k} = -\frac{D_{k,i}w_k}{w_i}. \tag{2.68}$$

Therefore, Equation 2.67 becomes

$$\begin{aligned}\frac{d\mathbf{Q}(x_i,y_j)}{dt} &+ \left\{\left[l_i(-1)\mathbf{F}^*(-1,y_j)\cdot(-\widehat{x})\frac{1}{w_i^{(x)}} + l_i(1)\mathbf{F}^*(1,y_j)\cdot(\widehat{x})\frac{1}{w_i^{(x)}} + \sum_{k=0}^{N}\mathbf{F}(x_k,y_j)\widehat{D}_{i,k}^{(x)}\right]\right\} \\ &+ \left\{\left[l_j(-1)\mathbf{F}^*(x_i,-1)\cdot(-\widehat{y})\frac{1}{w_j^{(y)}} + l_j(1)\mathbf{F}^*(x_i,1)\cdot(\widehat{y})\frac{1}{w_j^{(y)}} + \sum_{l=0}^{M}\mathbf{G}(x_i,y_l)\widehat{D}_{j,l}^{(y)}\right]\right\} \\ &= 0, \quad i=0,...,N. \quad j=0,..,M.\end{aligned} \tag{2.69}$$

To compute the time derivative $\frac{d\mathbf{Q}_{i,j}}{dt}$ in Equation 2.69, we need the Gauss-Legendre (GL) points and weights, the derivative matrices, and the boundary fluxes $\mathbf{F}^*\cdot\widehat{n}$. Algorithms for GL points, weights and derivative matrices are given in [40]. In this work, we only explain how to tackle the boundary fluxes. Note that we have discontinuities on element interfaces. Solving the numerical flux across the discontinuity is the so-called *Riemann problem*. In the next section, we explain how to obtain the boundary fluxes in a Riemann problem.

Riemann Problem for Conservation Laws

The Riemann problem is a typical model for many numerical problems that deal with wave propagation. It is a combination of a partial differential equation (PDE) and piecewise constant initial conditions [29]. The Riemann problem is an initial value problem (IVP). For a hyperbolic system of conservation laws the Riemann problem is written as

$$\begin{aligned}PDE: &\quad \frac{\partial\mathbf{u}}{\partial t} + \frac{\partial\mathbf{F}}{\partial x} = 0 \\ IC: &\quad \mathbf{u}(x,0) = \mathbf{u}_0(x) = \begin{cases}\mathbf{u}_L, & if\ x < x_0 \\ \mathbf{u}_R, & if\ x > x_0,\end{cases}\end{aligned} \tag{2.70}$$

where $\mathbf{u}_L$ and $\mathbf{u}_R$ are the so-called left and right states of the Riemann problem. x_0 is the location of the initial discontinuity. The flux $\mathbf{F}$ is assumed to only depend on $\mathbf{u}$. Fig. 2.4 illustrates the set up of the problem.

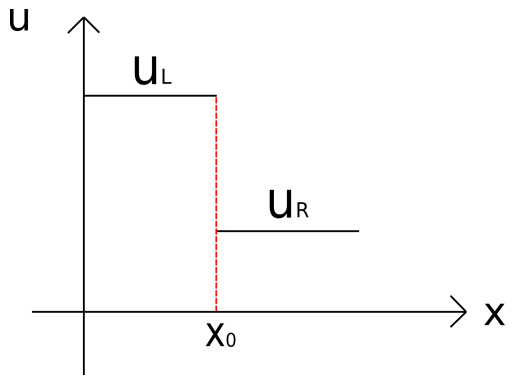


Figure 2.4: Illustration of the initial state of the Riemann problem. $\mathbf{u}_L$ and $\mathbf{u}_R$ are the left and right states of the Riemann problem. At time 0 two states are separated at position $x = x_0$.

Since the flux $\mathbf{F}$ only depends on the vector of observed variables $\mathbf{u}$, *i.e.*, $\mathbf{F} = \mathbf{F}(\mathbf{u})$, we rewrite the conservation law as

$$\mathbf{u}_t + \mathbf{F}_x(\mathbf{u}) = 0. \tag{2.71}$$

Equation 2.71 can be written as:

$$\mathbf{u}_t + \frac{\partial \mathbf{F}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial x} = 0. \tag{2.72}$$

$$\Rightarrow \mathbf{u}_t + \mathbf{A}\mathbf{u}_x = 0, \tag{2.73}$$

where $\mathbf{A}$ is the coefficient matrix. For a hyperbolic system $\mathbf{A}$ is a *Jacobian matrix*:

$$\mathbf{A} = \frac{\partial \mathbf{F}}{\partial \mathbf{u}}. \tag{2.74}$$

$\mathbf{A}$ has real eigenvalues, $\lambda_i(\mathbf{u})$ $(i = 1, ..., n)$, for a system of n conservation laws. The matrix $\mathbf{A}$ is proven to be diagonalizable [74]:

$$\mathbf{A} = \mathbf{K}\Lambda\mathbf{K}^{-1}, \tag{2.75}$$

where Λ is the diagonal matrix, with the eigenvalues λ_i of $\mathbf{A}$ on its diagonal. The columns $\mathbf{K}^{(i)}$ of $\mathbf{K}$ are the right eigenvectors of $\mathbf{A}$ corresponding to the eigenvalues λ_i, that is

$$\Lambda = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix}, \mathbf{K} = \left[\mathbf{K}^{(1)}, \cdots, \mathbf{K}^{(n)}\right], \mathbf{A}\mathbf{K}^{(i)} = \lambda_i \mathbf{K}^{(i)} \tag{2.76}$$

We can define a new set of dependent variables $\mathbf{W} = (w_1, w_2, \cdots, w_n)^T$ based on the inverse matrix $\mathbf{K}^{-1}$

$$\mathbf{W} = \mathbf{K}^{-1}\mathbf{u}. \tag{2.77}$$

So for a linear system, since $\mathbf{A}$ is a constant, then $\mathbf{K}$ is a constant too. We can express the system by using $\mathbf{W}$, and the system is *completely decoupled*. $\mathbf{W}$ is defined as the vector of *characteristic variables*. Next, we express the system by the characteristic variables $\mathbf{W}$. The derivatives of state the vector become:

$$\mathbf{u}_t = \mathbf{K}\mathbf{W}_t, \quad \mathbf{u}_x = \mathbf{K}\mathbf{W}_x. \tag{2.78}$$

We substitute Equation 2.78 in to the conservation laws and get

$$\mathbf{u}_t + \mathbf{F}_x(\mathbf{u}) = \mathbf{u}_t + \mathbf{A}\mathbf{u}_x = \mathbf{K}\mathbf{W}_t + \mathbf{A}\mathbf{K}\mathbf{W}_x = 0. \tag{2.79}$$

Multiplying this equation on the left by $\mathbf{K^{-1}}$ gives

$$\mathbf{W}_t + \mathbf{K^{-1}}\mathbf{A}\mathbf{K}\mathbf{W}_x = \mathbf{W}_t + \Lambda\mathbf{W}_x = 0. \tag{2.80}$$

Equation 2.80 is called the *canonical form* or *characteristic form* of the system.

Boundary Fluxes

In this section, we present the derivation of boundary fluxes $\mathbf{F}^*$ in Equation 2.69. The boundary flux is

$$\mathfrak{F} = \mathbf{f}_x + \mathbf{g}_y = A_x\mathbf{q}_x + A_y\mathbf{q}_y, \tag{2.81}$$

where coefficient matrices A_x and A_y are given by Equation 2.47:

$$A_x = \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, A_y = \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{2.82}$$

The boundary fluxes are also called *numerical fluxes*, which is an approximation of to the physical fluxes $\mathfrak{F}$. For elements on the boundaries of the domain, we impose the boundary conditions on the *upwind* side [74]. The upwind side is decided by the sign of the wave speed: for positive wave speed (with respect to the coordinate), the boundary condition should be set on the left, and as for negative wave speed, the boundary condition should be set on the right. We evaluate the solution on the downwind side from the interpolants.

It is not clear what the upwind side of this system is. So we decouple the wave components that make up the solution vector.

We first deal with the derivative with respect to x. We diagonalize the matrix A_x as in Equation 2.75. The diagonal matrix Λ_x, the vector of right eigenvectors of A_x and its inverse are

$$\Lambda_x = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix}, K_x = \begin{bmatrix} c & -c & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, K_x^{-1} = \begin{bmatrix} \frac{1}{2c} & \frac{1}{2} & 0 \\ -\frac{1}{2c} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.83}$$

The diagonal matrix Λ_x allows us to separate the wave into three directions: right-going, left-going and stationary.

$$\begin{aligned}\Lambda_x &= \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} c & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \Lambda_x^+ + \Lambda_x^- + \Lambda_x^0.\end{aligned} \tag{2.84}$$

Then we are able to split the wave into three components. Each component only contains one direction of wave.

$$A_x = K_x \Lambda_x^+ K_x^{-1} + K_x \Lambda_x^- K_x^{-1} + K_x \Lambda_x^0 K_x^{-1} = A_x^+ + A_x^- + A_x^0. \tag{2.85}$$

Therefore, the boundary flux can be evaluated as

$$\mathbf{F} \cdot \widehat{n} = A^+ \mathbf{q} + A^- \mathbf{q} + A^0 \mathbf{q}. \tag{2.86}$$

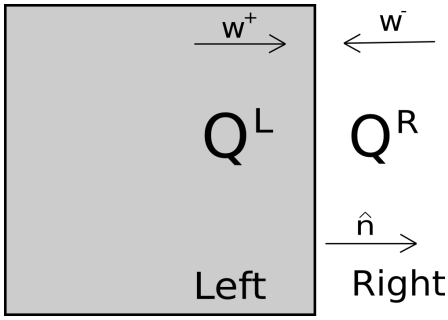


Figure 2.5: Element boundary flux on the right element interface. $\widehat{n}$ is the unit vector. $\mathbf{Q}^L$ and $\mathbf{Q}^R$ are the interior and external solutions respectively. w^+ and w^- are used to indicate the right-going and left-going waves.

We use Fig. 2.5 to explain the element boundary flux. Suppose we are on the right side of the element interface, with the unit vector $\widehat{n}$ pointed outwards from the interface. $\mathbf{Q}^L$ is computed from the element interior solution by interpolation and $\mathbf{Q}^R$ is the external solution. The element's numerical flux is a function of these two solutions on the boundary, $\mathbf{F}^* \left(\mathbf{Q}^L, \mathbf{Q}^R; \widehat{n}\right)$. According to Equation 2.86, the boundary flux should be calculated as

$$\mathbf{F}^* \cdot \widehat{n} = A^+ \mathbf{Q}^L + A^- \mathbf{Q}^R. \tag{2.87}$$

In this way, we ensure both internal and external states are on the upwind side.

Now we are ready to compute the numerical fluxes. We first write out the characteristic variable $\mathbf{W}_x$ in the x direction.

$$\mathbf{W}_x = K_x^{-1} \mathbf{q} = \begin{bmatrix} \frac{1}{2c} & \frac{1}{2} & 0 \\ -\frac{1}{2c} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p+cu}{2c} \\ \frac{-p+cu}{2c} \\ v \end{bmatrix}. \tag{2.88}$$

Next we multiply $\mathbf{W}_x$ with the diagonal matrix Λ_x, which contains the direction of the wave

$$\Lambda_x \mathbf{W}_x = \Lambda_x K_x^{-1} \mathbf{q} = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{p+cu}{2c} \\ \frac{-p+cu}{2c} \\ v \end{bmatrix} = \begin{bmatrix} c \cdot \frac{p^L+cu^L}{2c} \\ -c \cdot \frac{-p^R+cu^R}{2c} \\ 0 \end{bmatrix} = \begin{bmatrix} c \cdot w^+ \\ -c \cdot w^- \\ 0 \end{bmatrix}, \tag{2.89}$$

where the superscripts "L" (left) and "R" (right) indicate which side the solution is from. We use w^+ and w^- to indicate the right-going and left-going waves, respectively.

In the end, the boundary flux in the x direction is

$$\begin{aligned} \mathbf{F}^* \cdot \widehat{x} = A_x \mathbf{q} = K_x \left(\Lambda_x K_x^{-1} \mathbf{q} \right) &= \begin{bmatrix} c & -c & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c \cdot w^+ \\ -c \cdot w^- \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} c^2(w^+ + w^-) \\ c(w^+ - w^-) \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} c\left(\frac{p^L - p^R}{2} + c\frac{u^L + u^R}{2} \right) \\ \frac{p^L + p^R}{2} + c\frac{u^L - u^R}{2} \\ 0 \end{bmatrix}. \end{aligned} \tag{2.90}$$

The boundary fluxes in the y direction share the same derivation. The diagonal matrix Λ_y is the same as in x direction, which also separates the wave into right-going, left-going, and stationary directions. The eigenvector matrix K_y and its inverse are:

$$\Lambda_y = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix}, K_y = \begin{bmatrix} c & -c & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, K_y^{-1} = \begin{bmatrix} \frac{1}{2c} & 0 & \frac{1}{2} \\ -\frac{1}{2c} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \end{bmatrix}. \tag{2.91}$$

Therefore, the characteristic variables $\mathbf{W}_y$ times eigenvalues are

$$\Lambda_y \mathbf{W}_y = \Lambda_y K_y^{-1} \mathbf{q} = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{p+cu}{2c} \\ \frac{-p+cu}{2c} \\ u \end{bmatrix} = \begin{bmatrix} c \cdot \frac{p^L+cu^L}{2c} \\ -c \cdot \frac{-p^R+cu^R}{2c} \\ 0 \end{bmatrix} = \begin{bmatrix} c \cdot w^+ \\ -c \cdot w^- \\ 0 \end{bmatrix}. \tag{2.92}$$

Finally, the numerical flux in the y direction is

$$\begin{aligned}\mathbf{F}^* \cdot \widehat{y} = A_y \mathbf{q} = K_y \left(\Lambda_y K_y^{-1} \mathbf{q}\right) &= \begin{bmatrix} c & -c & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} c \cdot w^+ \\ -c \cdot w^- \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} c^2(w^+ + w^-) \\ 0 \\ c(w^+ - w^-) \end{bmatrix} \\ &= \begin{bmatrix} c\left(\frac{p^L - p^R}{2} + c\frac{v^L + v^R}{2}\right) \\ 0 \\ \frac{p^L + p^R}{2} + c\frac{v^L - v^R}{2} \end{bmatrix}.\end{aligned} \tag{2.93}$$

At this point, we have obtained the full expression of the boundary fluxes in two directions. Here we present two algorithms to compute the boundary fluxes of the Riemann problem.

Algorithm 2.1 *Riemann_solver_x* is based on Equation 2.90. *q_l* and *q_r* are the solutions on the two sides of the element interface. C is the sound speed in this case. *normal* corresponds to the outward normal vector's value with respect to the x axis: it should be 1 or -1. The algorithm outputs the numerical fluxes n_flux.

Algorithm 2.1 Riemann_solver_x: Riemann solver in x direction.

Input: *q_l*, *q_r*, C, *normal*

Output: n_flux

1: $p\_l = q\_l[0]$
2: $u\_l = q\_l[1]$
3: $v\_l = q\_l[2]$
4: $p\_r = q\_r[0]$
5: $u\_r = q\_r[1]$
6: $v\_r = q\_r[2]$
7: $w\_l = (p\_l + C \cdot u\_l)/2.0$
8: $w\_r = (p\_r + C \cdot u\_r)/2.0$
9: $n\_flux[0] = C \cdot (w\_l - w\_r) \cdot normal$
10: $n\_flux[1] = (w\_l + w\_r) \cdot normal$
11: $n\_flux[2] = 0$

Similarly, in the y direction, based on Equation 2.93, we present Algorithm 2.2 *Riemann_y*.

Algorithm 2.2 Riemann_solver_y: Riemann solver in y direction.

Input: q_l, q_r, $index$, C, $normal$

Output: n_flux

1: $p\_l = q\_l[0]$
2: $u\_l = q\_l[1]$
3: $v\_l = q\_l[2]$
4: $p\_r = q\_r[0]$
5: $u\_r = q\_r[1]$
6: $v\_r = q\_r[2]$
7: $w\_l = (p\_l + C \cdot u\_l)/2.0$
8: $w\_r = (p\_r + C \cdot u\_r)/2.0$
9: $n\_flux[0] = C \cdot (w\_l - w\_r) \cdot normal$
10: $n\_flux[1] = 0$
11: $n\_flux[2] = (w\_l + w\_r) \cdot normal$

Time Integration

With the element boundary fluxes and the product of the element fluxes and the derivative matrix D, we are able to obtain the time derivative $\mathbf{Q}_t$ in Equation 2.69. The next step is to integrate the equation in time. We implement a third order low-storage *Runge-Kutta (RK) method* proposed by Williamson [79]. This method has been proven to be memory-friendly for high-order scheme numerical simulation. It requires $2N$ (N is the total number of variables) levels of storage and the truncation error bounds are comparable to the normal Runge-Kutta methods [79].

We first introduce the general form of RK method. Then the algorithm of the 3rd-order low-storage RK method is explained and given.

General Runge-Kutta Method

Runge-Kutta methods are widely used in numerical computation. These methods have a variety forms both implicit and explicit. RK methods are mainly used in approximating solutions of ordinary differential equations (ODEs). In this work, we only consider explicit RK methods.

Let $y(t)$ be the unknown function of time t. Note that y can be a scalar or a vector. We have the initial value problem

$$\frac{dy}{dt} = y' = f(t, y), \quad y(t_0) = y_0, \tag{2.94}$$

where $\frac{dy}{dt} = y'$ is the derivative of y with respect to time t. t_0 is the initial time and y_0 is the corresponding value of the solution at t_0. The function f and initial condition y_0 are provided.

Suppose the solution of current time is y_n, then the solution at the next time step y_{n+1} can be

approximated as:

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{i=1}^{s} b_i k_i \\ &= y_n + h(b_1 k_1 + b_2 k_2 + ... + b_s k_s). \end{aligned} \tag{2.95}$$

The equation above is a general form of an s-stage explicit RK method. h is the size of one time step. k_i is the intermediate stage of one time step. The coefficient b_i is the weight applied to the target stage. Thus, y_{n+1} is approximated by a sum of y_n and s weighted increments. The intermediate stages k are defined as:

$$\begin{aligned} k_1 &= f\left(t_n, y_n\right) \\ k_2 &= f\left(t_n + c_2 h, y_n + h(a_{21} k_1)\right) \\ &\vdots \\ k_s &= hf\left(t_n + c_s h, y_n + h(a_{s1} k_1 + a_{s2} k_2 + ... + a_{s,s-1} k_{s-1})\right), \end{aligned} \tag{2.96}$$

where the a_{ij}, c_i $(1 \leq i, j \leq s)$ are both coefficients. The a_{ij} make up *Runge-Kutta matrix* A. a_{ij} and c_i are connected with the relationship:

$$c_i = \sum_{j=0}^{s} a_{ij}, \quad i = 1, ..., s, \tag{2.97}$$

which means c_i is the sum of the ith row of matrix A.

Coefficients b_i have the constraint:

$$\sum_{i=1}^{s} b_i = 1 \tag{2.98}$$

These coefficients a_{ij}, b_i, c_i are commonly expressed in a *Butcher tableau*[1]:

c_1	a_{11}	a_{12}	$\dots$	a_{1s}
c_2	a_{21}	a_{22}	$\dots$	a_{2s}
$\vdots$	$\vdots$		$\ddots$	$\vdots$
c_s	a_{s1}	a_{s2}	$\dots$	a_{ss}
	b_1	b_2	$\dots$	b_s

Table 2.1: Butcher tableau of Runge-Kutta method.

For an s-stage, explicit RK approach, the RK matrix A is always a lower triangular-matrix with $c_1 = 0$:

[1] named after John C. Butcher

c_1	0			
c_2	a_{21}			
c_3	a_{31}	a_{32}		
$\vdots$	$\vdots$		$\ddots$	
c_s	a_{s1}	$\dots$	$a_{s-1,s}$	0
	b_1	b_2	$\dots$	b_s

Table 2.2: Butcher tableau of an s-stage explicit Runge-Kutta method.

Third-order Low-storage Runge-Kutta

We now explain the third-order low-storage Runge-Kutta method. Suppose we have an ordinary differential equation

$$u_t = F(u, t). \tag{2.99}$$

We use Δt to represent the time step, and $t_n = n\Delta t$ is the current time. The solution vector u has N variables. The solution approximation is $U^n \approx u(t_n)$. Then, the approximation for the next time step t_{n+1} is calculated as follows:

$$\begin{aligned}
&U \leftarrow U^n, \\
&G \leftarrow F(U, t_n), \\
&U \leftarrow U + \frac{1}{3}\Delta t G, \\
&G = -\frac{5}{9}G + F\left(U, t_n + \frac{1}{3}\Delta t\right), \\
&U \leftarrow U + \frac{15}{16}\Delta t G, \\
&G \leftarrow -\frac{153}{128}G + F\left(U, t_n + \frac{3}{4}\Delta t\right), \\
&U^{n+1} \leftarrow U + \frac{8}{15}\Delta t G.
\end{aligned} \tag{2.100}$$

The G here is an intermediate variable. In this pattern, the successive values of U and G overwrite the previous ones, so at any stage, only $2N$ storage locations (for one direction) for U and G are required. The low-storage requirement of this method is beneficial when the problem systems are very large.

We summarize the coefficients of the 3rd-order Runge-Kutta method in Table 2.3 to be used in Algorithm 2.3 *DG_step_by_RK*3, written for advancing a 2D vector in time. Note that since this method is an explicit time integration method, we have to fulfill the *Courant Friedrichs Lewy (CFL) condition* for stability, which we will present in Section 2.3.2.

m	a_m	b_m	g_m
0	0	0	1/3
1	-5/9	1/3	15/16
2	-153/128	3/4	8/15

Table 2.3: Coefficients of the third-order low storage Runge-Kutta method.

To apply Runge-Kutta time integration we input the current time t_n, time step Δt and the time derivatives $\dot{\Phi}$. By using the coefficients in the table above, the time integration can be easily achieved.

Algorithm 2.3 DG_step_by_RK3: time integration by using third-order low-storage Runge-Kutta method.

Input: tn, Δt, $\dot{\Phi}$

Output: Φ

for $m = 0 \to 3$ **do**
  $t = tn + b_m \Delta t$
  $\dot{\Phi}_{i,j}, \quad i = 0, ..., N, \;\; j = 0, ..., M$ ▷ The time derivatives of solutions.
  for $j = 0 \to M$ **do**
    for $i = 0 \to N$ **do**
      $G_{i,j} = a_m G_{i,j} + \dot{\Phi}_{i,j}$
      $\Phi_{i,j} \mathrel{+}= g_m \Delta t G_{i,j}$
    end for
  end for
end for

Stability Analysis

Stability is one of the important features of every numerical method for solving differential equations. Study of the absolute stability of a differential equation is helpful in determining a suitable time step size. In this section, we first show the region of absolute stability of 3rd-order Runge-Kutta method. Then, a very important criterion to investigate the stability of solutions to partial differential equations (PDEs) under finite difference approximations, which is known as the *CFL condition*, is introduced. Finally, we apply Fourier analysis to the space discretization scheme and obtain the overall stability criterion of the numerical approximation.

Runge-Kutta Stability Analysis

In this section, we analyze the stability of time integration method — 3rd-order low-storage Runge-Kutta (RK). The stability is examined by using the model ODE problem:

$$\frac{dy}{dt} = y' = \lambda y, \tag{2.101}$$

whose exact solution is $y = e^{\lambda t}$.

Let $z = h\lambda$, where h is the time step size. Therefore, the solution converges to zero when $z < 0$ as $t \to \infty$, *i.e*, $\lambda < 0$ since $h > 0$. z is defined in the complex plane. To obtain a stable result, we require the real component of z smaller than zero ($Re(z) < 0$), therefore, we are only interested in the left-hand-side of the complex plane (Fig. 2.6).

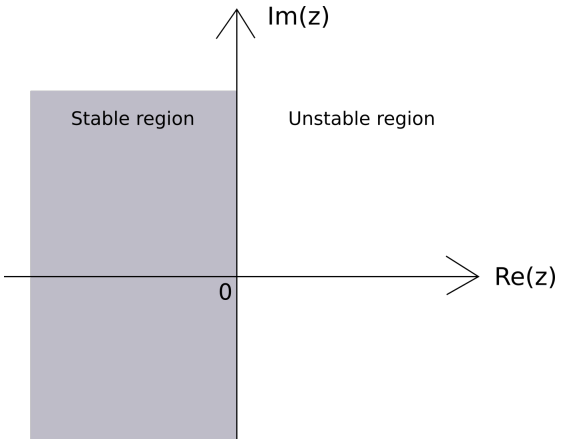


Figure 2.6: Stable region of the model function $y = e^{\lambda t}$. $z = h\lambda$, where h is the step size. The stable region is shaded in grey.

For a single step of length h, the exact solution is effectively multiplied by e^z:

$$e^{\lambda(t+h)} = e^{\lambda t} \cdot e^{\lambda h} = e^{\lambda t} \cdot e^{z} \tag{2.102}$$

For the numerical approximation, taking a new step h is equivalent to multiply the computed solution by a function of z, which is the so-called *stability function R(z)*. To get a similar convergence profile, we want our stability function to behave like the function e^z.

Using a single step method, approximating the model problem can be written as:

$$y_{n+1} = R(z)y_n, \quad z = h\lambda, \tag{2.103}$$

where y_n is the solution at current time t_n and y_{n+1} is the approximated solution at the next time step. $R(z)$ is the stability function which can be seen as an amplification factor from the current time to the future time. Thus, the behaviour of the $R(z)$ function indicates the stability of the numerical approach.

Since we want a stable approach and the solution is approximated by multiplying $R(z)$ at each step, for the solution to approach to zero as $n \to \infty$ we require:

$$|R(z)| < 1, \tag{2.104}$$

where $|\cdot|$ is the modulus (absolute value) operator. Therefore, we obtain the region where the solution stays stable:

$$\{z \in \mathbb{C} : |R(z)| \leq 1\}, \tag{2.105}$$

where $\mathbb{C}$ represents the set of complex numbers. The region for all z to fulfill Equation 2.105 is called the *region of absolute stability.*

Next, we derive the stability of a general RK method. Recall that a simple step s-stage RK method can be written as:

$$\begin{aligned} k_1 &= f\left(y_0 + h(a_{11}k_1 + a_{12}k_2 + \ldots + a_{1s}k_s)\right) \\ k_2 &= f\left(y_0 + h(a_{21}k_1 + a_{22}k_2 + \ldots + a_{2s}k_s)\right) \\ &\vdots \\ k_s &= f\left(y_0 + h(a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{ss}k_s)\right). \end{aligned} \tag{2.106}$$

The k stages Equations 2.106 are the same as those in Equation 2.96 that we previously defined. Here, we ignore the variables t, c and focus on the second half of the $f(t, y)$.

Let $Y = (k_1, k_2, \ldots, k_s)^T$ be a vector of k stage values. We define a column vector $\mathbf{1} = (1, 1, \ldots, 1)^T$, which contains s entries of 1. The vector $\mathbf{1}$ is used to simplify the expression of the equations we derive later. Then Y can be expressed as:

$$Y = \lambda\left(\mathbf{1}y_0 + hAY\right) = \lambda\mathbf{1}y_0 + zAY. \tag{2.107}$$

Rearranging Equation 2.107 we obtain:

$$Y = \left(I - zA\right)^{-1}\lambda\mathbf{1}y_0. \tag{2.108}$$

Substituting into the next time step y_{n+1} (Equation 2.95) we obtain:

$$\begin{aligned} y_1 &= y_0 + h\mathbf{b}^T Y \\ &= y_0 + h\mathbf{b}^T\left(I - zA\right)^{-1}\lambda\mathbf{1}y_0 \\ &= y_0 + z\mathbf{b}^T\left(I - zA\right)^{-1}\mathbf{1}y_0 \\ &= y_0\left[1 + z\mathbf{b}^T\left(I - zA\right)^{-1}\mathbf{1}\right]. \end{aligned} \tag{2.109}$$

Comparing Equation 2.109 with Equation 2.103, we obtain the stability function $R(z)$:

$$R(z) = 1 + z\mathbf{b}^T\left(I - zA\right)^{-1}\mathbf{1}. \tag{2.110}$$

Setting $|Re(z)| \leq 1$ gets a general stability for the RK method. It can be applied to either explicit methods or implicit methods.

We apply the theory of *geometric series of matrices* to term $(I - T)^{-1}$:

$$\left(I - T\right)^{-1} = \sum_{k=0}^{\infty} T^k, \tag{2.111}$$

where T is a matrix and I is the identical matrix.

Then the stability function becomes:

$$\begin{aligned} R(z) &= 1 + z\mathbf{b}^T \left(I - zA\right)^{-1} \mathbf{1} \\ &= 1 + z\mathbf{b}^T \left(I + zA + z^2A^2 + \ldots\right) \mathbf{1}. \end{aligned} \tag{2.112}$$

For explicit methods, we have the relationship:

$$\mathbf{b}^T A^k \mathbf{1} = \mathbf{b}^T A^{k-1} \mathbf{c}, \tag{2.113}$$

where $\mathbf{c}$ is the coefficient vector which satisfies the relationship $c_i = \sum_{j=0}^{s} a_{ij}$ $(i = 1, \ldots, s)$ with matrix A, *i.e*, $\mathbf{c} = A\mathbf{1}$.

Applying this relationship to the stability function yields

$$\begin{aligned} R(z) &= 1 + z\mathbf{b}^T\mathbf{1} + z^2\mathbf{b}^T A\mathbf{1} + z^3\mathbf{b}^T A^2\mathbf{1} + \ldots \\ &= 1 + z\mathbf{b}^T\mathbf{1} + z^2\mathbf{b}^T A^0\mathbf{c} + z^3\mathbf{b}^T A\mathbf{c} + \ldots \end{aligned} \tag{2.114}$$

Now we can compare our stability function 2.114 to the Taylor expansion of the model equation solution:

$$e^z = \sum_{p=0}^{\infty} \frac{z^p}{p!} = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots. \tag{2.115}$$

We would like the stability function $R(z)$ to behave like the exponential function. Thus, the order of the Runge-Kutta method is the highest value of p for which the two equations (Equation 2.114 and 2.115) agree.

For instance, the coefficients of the 3rd-order low-storage RK method are given in Table 2.4.

0	0		
1/3	1/3	0	
3/4	−3/16	15/16	0
	1/6	3/10	8/15

Table 2.4: Butcher tableau of explicit 3rd-order low-storage Runge-Kutta method.

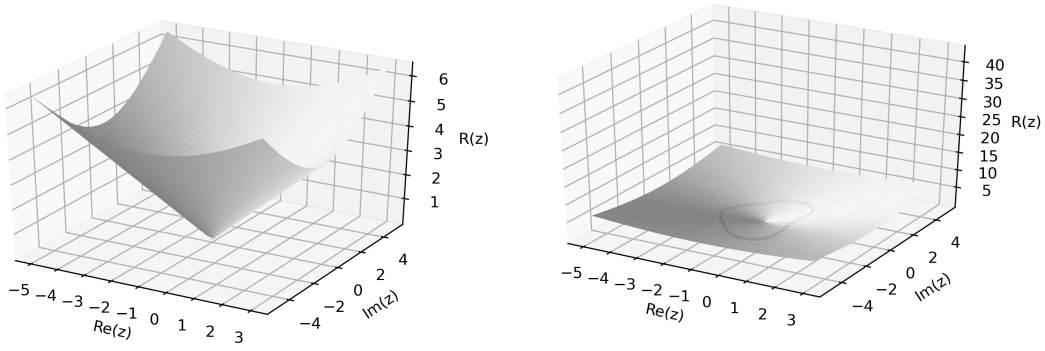
Plugging in the coefficients of a_{ij}, b_j and c_i into the stability function 2.114, we obtain:

$$\begin{aligned} \mathbf{b}^T\mathbf{1} &= 1 \\ \mathbf{b}^T A^0\mathbf{c} &= \frac{1}{2} \\ \mathbf{b}^T A^1\mathbf{c} &= \frac{1}{6} \\ \mathbf{b}^T A^2\mathbf{c} &= 0 \\ \ldots &= 0 \end{aligned} \tag{2.116}$$

Comparing with Equation 2.115, we deduce that the highest order p for which the two equations agree is 3. Therefore, we prove that our RK method is indeed a third order method. The stability function is:

$$R(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6}, \tag{2.117}$$

where $z = a + ib$ is in the complex plane.



(a) Stability function of the 3rd-order low-storage RK method

(b) Contour plot of absolute stable region ($|R(z)| = 1$). The contour is coloured in grey.

Figure 2.7: The configuration of stability function $R(z)$ of the 3rd-order low-storage RK method. $Re(z)$ is the real axis, $Im(z)$ is the imaginary axis. The absolute value of $R(z)$ is plotted in the vertical axis. The contour plot of $|R(z)| = 1$ is shown as a grey line in (b).

We plot the stability function $R(z)$ of our RK method in Fig. 2.7a. It forms a cone shape. To generate the absolute stability region, we slice the surface plot with a plane $|R(z)| = 1$ and get its contour (Fig. 2.7b). A 2D absolute stability region plot is shown in Fig. 2.8.

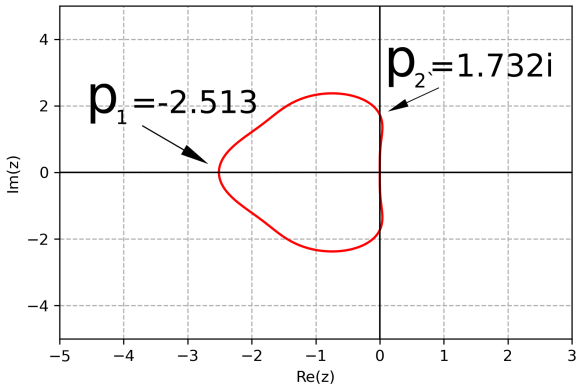


Figure 2.8: Absolute stability region of the 3rd-order low-storage RK method. Point p_1 is the intersection point on the real axis. Points p_2 is the intersection point on the imaginary axis.

The maximum magnitude of the real value inside the absolute stability region is 2.513 (point p_1), which agrees with Williamson's conclusion [79]. The intersections on the imaginary axis have magnitudes of 1.732 (point p_2). These values define the boundary of the absolute stability region.

CFL Condition

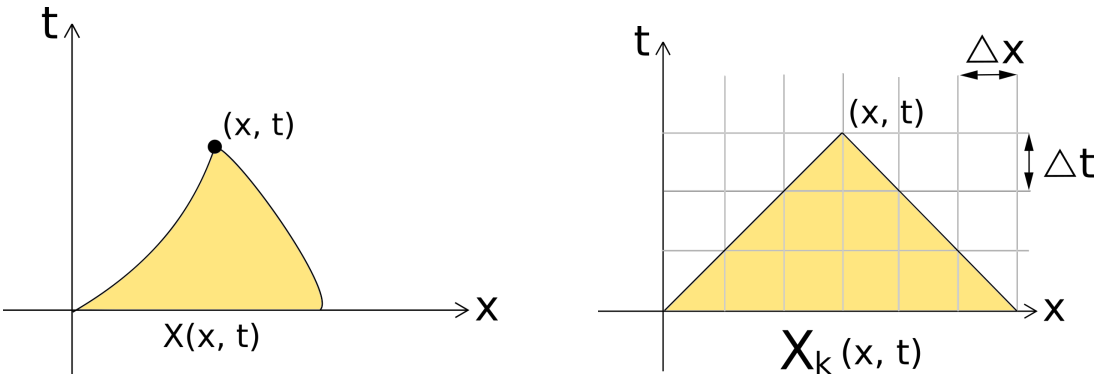
In 1928, Richard Courant, Kurt Friedrichs and Hans Lewy proposed a technique to prove the existence of solutions to PDEs, namely, the *CFL condition*. The CFL condition is a necessary condition for stability while solving certain PDEs (mostly hyperbolic ones) numerically. It was developed for investigating the solutions of explicit time integration schemes with finite difference approximations. It indicates that the time step must be less than a certain factor of the space step for the solution to be stable. In the following, we use an advection equation as the model equation to illustrate the idea of CFL condition.

The linear advection equation is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \tag{2.118}$$

where u is the solution, $u = u(x,t)$. c is a constant, representing the wave speed. Such a linear advection equation propagates a wave shape towards the right or left according to $c > 0$ or $c < 0$ while preserving the wave shape. We consider an initial-value problem of $u(x,0) = u_0(x)$.

Courant, Friedrichs and Lewy pointed out that in order to obtain a converged result, the *numerical domain of dependence* must cover the *mathematical/analytical domain of dependence*. The mathematical domain of dependence $X(x,t)$ of $u(x,t)$ is shown in Fig. 2.9a: it is the set of points in space where the initial data at $t = 0$ may have some effect on the solution $u(x,t)$ [75]. For our model equation $u_t + cu_x = 0$, $X(x,t)$ is a straight line, which is called the *characteristic curve* for the PDE. Fig. 2.9b shows the numerical domain of dependence $X_k(x,t)$. For an explicit finite difference scheme, the solution of the new time step value u_j^{n+1} depends on a finite range of value of the previous step. If the time step Δt and space step Δx are fixed, the $X_k(x,t)$ fans out backward in a triangle shape.



(a) Analytical domain of dependence.

(b) Numerical domain of dependence.

Figure 2.9: Domain of dependence.

Next, we use a small example to further illustrate the CFL condition. Suppose we apply the Euler method to the time derivative and upwind discretization to the spatial derivative. Then we have:

$$Euler: \frac{\partial u}{\partial t} = \frac{u_j^{n+1} - u_j^n}{\Delta t}, \tag{2.119}$$

$$Upwind: \frac{\partial u}{\partial x} = \frac{u_j^n - u_{j-1}^n}{\Delta x}. \tag{2.120}$$

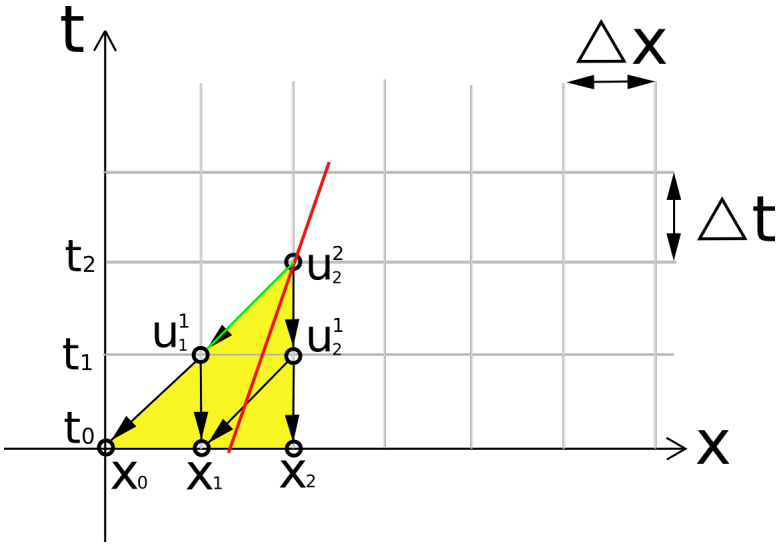


Figure 2.10: Domain of dependence example: Euler method in time and upwind spatial scheme to solve the advection equation. The numerical domain of dependence is coloured in yellow. The analytical solution is a straight line with slope $\frac{1}{c}$, which is indicated by the red line. To fulfill the CFL condition, the slope of the numerical domain of dependence (green line) should be less than the slope of the analytical domain of dependence (red line).

We pick a point in space x_2 and a time t_2 and analyze the domain of dependence of a solution u_2^2 at (x_2, t_2). According to the upwind scheme, the value of u_2^2 comes from its former solutions u_1^1 and u_2^1 (see Fig. 2.10). Similar rules apply to solutions u_1^1 and u_2^1, whose solutions come from u_0^0, u_1^0 and u_1^0, u_2^0, respectively. The previous solutions, form the numerical domain of dependence, which is coloured in yellow in Fig. 2.10. The analytical domain of dependence is simple: since the wave travels with a constant speed c, its characteristic curve is a straight line with the slope $\frac{1}{c}$ (indicated by the red line in Fig. 2.10). To fulfill the CFL condition, the analytical domain of dependence should lie within the numerical domain of dependence, otherwise the numerical scheme is not using the relevant previous solutions to generate the new one. Thus, we have:

$$\left|\frac{\Delta t}{\Delta x}\right| \leq \left|\frac{1}{c}\right|, \tag{2.121}$$

where $\frac{\Delta t}{\Delta x}$ is the slope of the left boundary of the numerical domain of dependence (represented by the green line in Fig. 2.10). Therefore, we obtain the relationship between time step and space step:

$$\Delta t \leq \frac{\Delta x}{|c|}. \tag{2.122}$$

Equation 2.122 indicates that to maintain the stability of the numerical approximation, the time step and space step should decrease/increase proportionally. For a fixed mesh, if one increases the wave speed, a smaller time step should be used in the numerical calculation.

We rearrange Equation 2.122 and get a conventional form of the CFL condition:

$$\left|\frac{c\Delta t}{\Delta x}\right| \leq 1. \tag{2.123}$$

This is the CFL condition for the linear advection equation. The number 1 on the right-hand side is called the *Courant number*. Note that it does not always equal 1. The value of the Courant number depends on the spatial and time discretization methods.

The CFL condition is a necessary but not a sufficient condition for the stability of a well-posed linear PDE [75]. In many cases, the spatial discretization method will impact the stability of the time discretization. Therefore, to properly analyze the stability of a PDE system, one should apply Fourier analysis instead. Here, we illustrate the insufficiency of the CFL condition to the numerical stability.

In the previous example, we derived the CFL condition for a first-order time integration scheme. The Courant number we obtain is 1. Now we investigate the Courant number for a higher order of time discretization.

For most time integration methods, the higher the order of discretization, the stricter the time step needs to be. However, this is not true for the Runge-Kutta method. We still consider the linear advection model Equation 2.118.

For a 3rd-order RK method, we choose the coefficients as:

0	0	0	0
1/2	1/2	0	0
1	−1	2	0
	1/6	2/3	1/6

Table 2.5: Butcher tableau of Runge-Kutta 3rd-order method.

This time we choose to use a 2nd-order central difference discretization scheme to tackle the spatial term:

$$\frac{\partial u(x_j)}{\partial x} \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}. \tag{2.124}$$

Then the equation can be approximated as:

$$\begin{aligned} u_j^{n+1} &= u_j^n + \Delta t \left(\frac{1}{6}k_1 + \frac{2}{3}k_2 + \frac{1}{6}k_3 \right), \\ k_1 &= u_x \left(u_j^n \right), \\ k_2 &= u_x \left(u_j^n + \frac{1}{2}\Delta t k_1 \right), \\ k_3 &= u_x \left[u_j^n + \Delta t \left(-k_1 + 2k_2 \right) \right]. \end{aligned} \tag{2.125}$$

According to the spatial discretization scheme (Equation 2.124) for u_x, k_1 depends on solution u_{j-1}^n and u_{j+1}^n. Applying the same logic, k_2 depends on solutions from u_{j-2}^n to u_{j+2}^n. As for k_3, it depends on solutions from u_{j-3}^n to u_{j+3}^n. Fig. 2.11 offers a clear illustration of the dependence relationships.

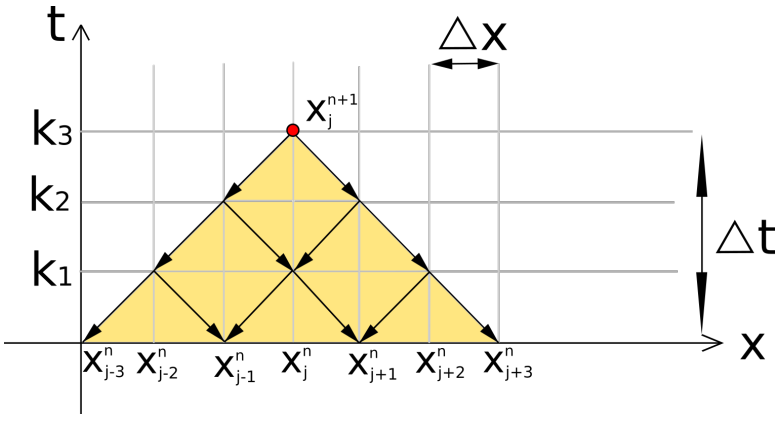


Figure 2.11: Numerical domain of dependence for 3rd-order RK method using central difference scheme. k_i $(i = 1, 2, 3)$ are the intermediate stages of RK. Solution at point x_j^{n+1} depends on solutions at points range from x_{j-3}^n to x_{j+3}^n. The numerical domain of dependence is coloured in yellow.

Comparing the domain of dependence in Fig. 2.11 and Fig. 2.10, it is clear that the high-order time and space stepping methods largely widen the stencil, and hence the domain of dependence. Thus, in this RK method, the Courant number should be enlarged to be 3 according to the slope:

$$\begin{aligned} \left|\frac{\Delta t}{3\Delta x}\right| &\leq \left|\frac{1}{c}\right| \\ \Rightarrow \left|\frac{c\Delta t}{\Delta x}\right| &\leq 3. \end{aligned} \tag{2.126}$$

The solution at time t^{n+1} requires the data from 3 grid points to the left and 3 grid points to the right. Thanks to the 3 intermediate stages in the RK method, the stencil fans out. Additionally, the second-order accurate central difference method does not contribute to the stability of the numerical approximation.

However, the numerical approximation will become unstable past a Courant number of 1.7, which is smaller than 3 we just derived. This is because the CFL condition is a necessary condition for numerical stability but not a sufficient one. The stability is affected not only by the time integration method, but also the space discretization method.

According to Trefethen [75], to obtain an absolute stability, the eigenvalues of the space discretization should lie in the stability region of the time-stepping method. If we apply Fourier analysis to the central difference formula, which we will explain in the next section, we can find out that its eigenvalues cover the complex interval $\left[-\frac{|c|i}{\Delta x}, \frac{|c|i}{\Delta x}\right]$. To restrain the eigenvalues of space discretization inside the absolute stability region in Fig. 2.8, we require:

$$|eig(time)| \geq |eig(space)|\,. \tag{2.127}$$

From the derivation in Section 2.3.2 we have:

$$i\,|z| \in [0, 1.73i]\,, \quad z = \lambda\Delta t, \tag{2.128}$$

$$\Rightarrow |\lambda| \in \left[0, \frac{1.73}{\Delta t}\right]. \tag{2.129}$$

To fulfill the stability condition 2.127 we have:

$$\begin{aligned} 0 \leq \frac{|c|}{\Delta x} &\leq \frac{1.73}{\Delta t} \\ 0 \leq \left|\frac{c\Delta t}{\Delta x}\right| &\leq 1.73. \end{aligned} \tag{2.130}$$

Thus, we obtain the proper CFL condition for 3rd-order RK scheme. The correct Courant number is 1.73.

Fourier Analysis of Hyperbolic Equation

Since the stability of the numerical approach depends not only on the time stepping method,

but also the space discretization scheme, we have to take both of the discretization influences into consideration. A potent way to analyze the numerical stability is by using *Fourier analysis*.

We still consider the linear advection equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \tag{2.131}$$

where u is a function of time and space: $u = u(t, x)$. c is a constant speed. Its sign determines the wave propagation direction.

In Fourier analysis, we consider the problem on domain $x \in [0, 2\pi]$ with periodic boundary condition. The solution u can be seen as a superposition of infinite sinusoidal waves with different wave numbers:

$$u(t, x) = \sum_{k=-\infty}^{+\infty} \widehat{u}_k(t) e^{ikx}, \tag{2.132}$$

$$e^{ikx} = cos(kx) + isin(kx). \tag{2.133}$$

$\widehat{u}_k(t)$ is the kth sinusoidal wave which has real and imaginary components. It is a function only of time t. The spatial component of $u(t, x)$ is separated as the exponential term, which is a combination of sinusoidal waves with different wave numbers. By taking Fourier transforms, we successfully separate time and spatial variables in the solution function and transform the mixed (partial differential) equation into to a set of ordinary differential equations in time as follows.

The Fourier transform facilitates analysis of the spatial derivatives since it translates the differential operator into a modification of the original function:

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial}{\partial x}\left(\sum_{k=-\infty}^{+\infty} \widehat{u}_k(t) e^{ikx}\right) \\ &= \sum_{k=-\infty}^{+\infty} \widehat{u}_k(t) \cdot (ik) \cdot e^{ikx} \\ &= \sum_{k=-\infty}^{+\infty} ik \cdot \widehat{u}_k(t) e^{ikx}. \end{aligned} \tag{2.134}$$

As for time derivative we have:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{d}{dt}\left(\sum_{k=-\infty}^{+\infty} \widehat{u}_k(t)\right) e^{ikx} \\ &= \sum_{k=-\infty}^{+\infty} \frac{d\widehat{u}_k(t)}{dt} e^{ikx} \end{aligned} \tag{2.135}$$

We plug Equation 2.134 and 2.135 into the advection equation 2.131 and obtain:

$$\sum_{k=-\infty}^{+\infty} \frac{d\widehat{u}_k(t)}{dt} e^{ikx} = -c \sum_{k=-\infty}^{+\infty} ik \cdot \widehat{u}_k(t) e^{ikx}. \tag{2.136}$$

The summation of $\widehat{u}_k(t)$ can be seen as the coefficients of e^{ikx}. For the system of equations to hold for each wave number, the coefficients on the both sides of the equation should be equal. Therefore, we have:

$$\frac{d\widehat{u}_k(t)}{dt} = -ick \cdot \widehat{u}_k(t). \tag{2.137}$$

Then a PDE stability problem gets simplified into an ODE one. Moreover, we know the analytical solution of Equation 2.137:

$$\widehat{u}_k(t) = e^{-ickt}. \tag{2.138}$$

Thus, the solution for every wave component oscillates in a sinusoidal way.

Next, we apply discrete Fourier stability analysis (also known as the *von Neumann stability analysis*) to the differential equation. Recall our discretization schemes (from the former section) of time being 3rd-order RK and for space central difference.

The Fourier transform of the discrete function u becomes:

$$u_j = u(x_j) = u(j\Delta x) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \widehat{u}_k(t) e^{ikj\Delta x}, \tag{2.139}$$

where $N = \frac{2\pi}{\Delta x}$ is the interval number of domain $[0, 2\pi]$. By restraining k in $\left[-\frac{N}{2}, \frac{N}{2} - 1\right]$, we avoid *aliasing* [2].

We first investigate the stability condition for finite difference approximation, it can help us further illustrate the stability condition for spectral approximation.

Plugging Equation 2.139 into the advection equation we obtain:

$$\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \frac{d\widehat{u}_k(t)}{dt} e^{ikj\Delta x} + c\frac{\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \widehat{u}_k(t) e^{ik(j+1)\Delta x} - \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \widehat{u}_k(t) e^{ik(j-1)\Delta x}}{2\Delta x} = 0. \tag{2.140}$$

Similar to continuous Fourier analysis, we have:

$$\frac{d\widehat{u}_k(t)}{dt} + c\widehat{u}_k(t)\frac{e^{ik\Delta x} - e^{-ik\Delta x}}{2\Delta x} = 0. \tag{2.141}$$

We expand the exponential components in sinusoidal forms:

$$e^{ik\Delta x} = cos(k\Delta x) + isin(k\Delta x), \tag{2.142}$$

$$e^{-ik\Delta x} = cos(k\Delta x) - isin(k\Delta x). \tag{2.143}$$

Substituting these two components into Equation 2.141 we obtain:

$$\frac{d\widehat{u}_k(t)}{dt} = -c\widehat{u}_k(t)\frac{isin(k\Delta x)}{\Delta x}. \tag{2.144}$$

[2] Aliasing is an effect causing different waves to become indistinguishable when sampled.

The eigenvalues of the solutions $\widehat{u}_k(t)$ are $-c\frac{isin(k\Delta x)}{\Delta x}$. Since $sin(k\Delta x) \in [-1, 1]$, the eigenvalues cover the imaginary axis from $-\frac{|c|i}{\Delta x}$ to $\frac{|c|i}{\Delta x}$.

To enforce the stability, we need the eigenvalues of the space discretization to lie inside the absolute stability region of the time discretization. That is why the RK method is essential in this case: its stability region covers some portion of imaginary axis. For 3rd-order RK we have (Equation 2.129):

$$|\lambda| \in \left[0, \frac{1.73}{\Delta t}\right]. \tag{2.145}$$

Thus, we require:

$$\begin{aligned} 0 \leq \frac{|c|}{\Delta x} &\leq \frac{1.73}{\Delta t} \\ \Rightarrow 0 \leq \left|\frac{c\Delta t}{\Delta x}\right| &\leq 1.73 \end{aligned} \tag{2.146}$$

to obtain a stable numerical approximation when using central differences in space.

Next, we investigate the CFL condition for spectral element approximation. We define our initial value problem as:

$$\begin{aligned} &u_t + cu_x = 0,\ x \in (-1, 1)\,,\ c > 0, \\ &u(x, 0) = u_0(x),\ x \in [-1, 1]\,, \\ &u(-1, t) = g(x), \end{aligned} \tag{2.147}$$

where c is a positive sound speed, $u_0(x)$ is the initial condition, and $g(x)$ is the left boundary condition. Since $c > 0$, we only need to impose a boundary condition on the upwind side of the domain, namely, on the left boundary.

Recall we approximate the solution u as:

$$u(x, t) \approx u_h(x, t) = \sum_{j=0}^{N} a_j(t) L_j(x) = \sum_{j=0}^{N} \widetilde{u}_j(t) l_j(x), \tag{2.148}$$

where $L_j(x)$ are the Legendre polynomials and $l_j(x)$ the polynomials in Lagrange form.

As in Section 2.3.2, we write the Equation 2.147 in weak form:

$$(u_t, v) + (cu_x, v) = 0, \tag{2.149}$$

where v is the test function. In the Galerkin approach, the test function is approximated in the same function space as the solution function. Therefore, it can be expanded as:

$$v(x) \approx v_h(x) = \sum_{j=0}^{N} b_j L_j(x) = \sum_{j=0}^{N} \widetilde{v}_j l_j(x). \tag{2.150}$$

Inserting Equation 2.150 into Equation 2.149 we obtain:

$$\sum_{j=0}^{N} \left[(u_t, l_j) + (cu_x, l_j)\right] \widetilde{v}_j = 0. \tag{2.151}$$

We obtain $(N+1)$ equations since the nodal values $\widetilde{v}_j$ are independent. Thus, Equation 2.151 requires the factors to vanish individually, *i.e.*:

$$(u_t, l_j) + (cu_x, l_j) = 0,\ j = 0, \cdots, N. \tag{2.152}$$

We apply integration by parts to Equation 2.152 to separate the boundary terms:

$$(u_t, l_j) + [cul_j]_{-1}^{1} - \left(cu, l_j^{'}\right) = 0. \tag{2.153}$$

Then we substitute the boundary condition in Equation 2.153:

$$(u_t, l_j) + c\left[u(1,t)l_j(1) - g(t)l_j(-1)\right] - \left(cu, l_j^{'}\right) = 0. \tag{2.154}$$

The term $u(1,t)$ is determined by the approximated solution since no boundary condition is applied on the right side.

Next, we plug the approximated solution (Equation 2.148) into Equation 2.154 and obtain:

$$\sum_{n=0}^{N} \dot{\widetilde{u}}_n \left(l_n, l_j\right) + c\left[u(1,t)l_j(1) - g(t)l_j(-1)\right] - c\sum_{n=0}^{N} \widetilde{u}_n \left(l_n, l_j^{'}\right) = 0, \tag{2.155}$$

where the dot on the top of $\widetilde{u}$ represents the time derivative.

The inner products are evaluated using Gauss quadrature:

$$(l_n, l_j) = \sum_{k=0}^{N} l_n(x_k) l_j(x_k) w_k = \delta_{nj} w_j, \tag{2.156}$$

$$\left(l_n, l_j^{'}\right) = \sum_{k=0}^{N} l_n(x_k) l_j^{'}(x_k) w_k = \delta_{nk} w_k l_j^{'}(x_k) = w_n l_j^{'}(x_n). \tag{2.157}$$

Substituting the inner products into Equation 2.155 we obtain:

$$\sum_{n=0}^{N} \dot{\widetilde{u}}_n \delta_{nj} w_j + c\left[u(1,t)l_j(1) - g(t)l_j(-1)\right] - c\sum_{n=0}^{N} \widetilde{u}_n l_j^{'}(x_n) w_n = 0, \tag{2.158}$$

$$\Rightarrow \dot{\widetilde{u}}_j w_j + c\left[u(1,t)l_j(1) - g(t)l_j(-1)\right] - c\sum_{n=0}^{N} \widetilde{u}_n l_j^{'}(x_n) w_n = 0. \tag{2.159}$$

Then, we rearrange Equation 2.159 as:

$$\dot{\widetilde{u}}_j = -c\left\{\left[u(1,t)\frac{l_j(1)}{w_j} - g(t)\frac{l_j(-1)}{w_j}\right] - \sum_{n=0}^{N} \widetilde{u}_n l_j^{'}(x_n)\frac{w_n}{w_j}\right\}, \tag{2.160}$$

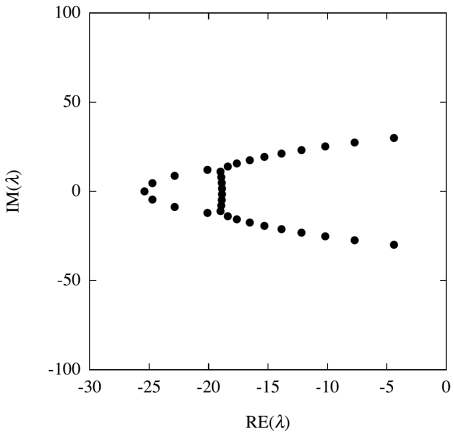
$$\Rightarrow \dot{\widetilde{u}}_j = -c\left\{\left[u(1,t)\frac{l_j(1)}{w_j} - g(t)\frac{l_j(-1)}{w_j}\right] + \sum_{n=0}^{N} \widehat{D}_{jn}\widetilde{u}_n\right\}, \tag{2.161}$$

where

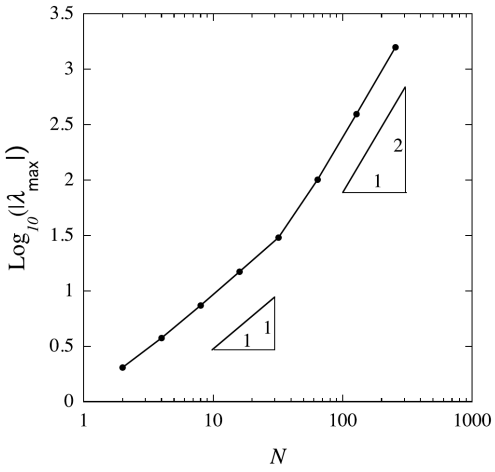
$$\widehat{D}_{jn} = -\frac{D_{nj} w_n}{w_j}, \tag{2.162}$$

and $D_{nj} = l_j^{'}(x_n)$ is the standard first order derivative matrix.

The time step limitation depends on the largest eigenvalue of the first derivative matrix $\widehat{D}_{jn}$. Analytic representations of the eigenvalues of the polynomial derivative matrices are not known, so we must find the eigenvalues numerically [40]. It turns out the eigenvalues of the first order derivative matrix have both real parts and imaginary parts (2.12a) which makes the stability of the spectral method harder to evaluate.



(a) Distribution of the eigenvalues for the Legendre first order derivative matrix when polynomial order $N = 32$.



(b) The growth of the maximum eigenvalue of the Legendre first order derivative has a second order (N^2) relationship with respect to the polynomial order N. Overall, $|\lambda_{max}| \sim 0.08N^2$

Figure 2.12: (a) The distribution of the eigenvalues of the Legendre first order derivative of order $N = 32$. (b) The relationship between the growth of the maximum eigenvalue with respect to the polynomial order. (Reprinted from [40])

In Fig. 2.12b we can observe that when polynomial order $N \leq 32$, the growth of the maximum eigenvalue is linear, however, when $N > 32$, the growth becomes second order, which imposes a stricter constraint on the time step size. To obtain a stable simulation, we need $c\lambda\Delta t$ resides inside the absolute stability region. From Fig. 2.12a, we observe that the largest eigenvalue lies on the real axis. Therefore, to ensure $c\left|\lambda\right|_{max}\Delta t$ stays inside the absolute stability region of third-order Runge-Kutta (Fig. 2.8), we require $c\left|\lambda\right|_{max}\Delta t \leq 2.51$. Since the largest eigenvalue grows as:

$$\begin{cases} \left|\lambda\right|_{max} \propto N, & N < 32, \\ \left|\lambda\right|_{max} \propto N^2, & N \geq 32, \end{cases} \tag{2.163}$$

we can get an approximated upper stability limit of the time step [40] of:

$$\Delta t \leq \begin{cases} \frac{2.5}{cN}, & N < 32, \\ \frac{38\times 2.51}{cN^2}, & N \geq 32. \end{cases} \tag{2.164}$$

In practice, however, for high order SEM or DG-SEM, we will take a smaller time step in order to have comparable accuracy in time as in space.

Apply DG-SEM to Multiple Elements

Assuming we are now on a square domain $\Omega = [0, L] \times [0, L]$. Recall the conservation law:

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0. \tag{2.165}$$

We derive the discontinuous Galerkin approximation from the weak form of the Equation 2.165. To get the weak form of the Equation 2.165, we multiply the integral of the equation with test functions ϕ:

$$\int_0^L \int_0^L (\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y)\, \phi dx dy. \tag{2.166}$$

In the Galerkin approach, the test functions are chosen as the basis functions. Since the solutions are computed on the Gauss-Legendre points inside each element, the tests functions do not have to be continuous on the element interfaces (Fig. 2.13). Then we can benefit from the higher precision of Gaussian quadrature on Gauss-Legendre points.

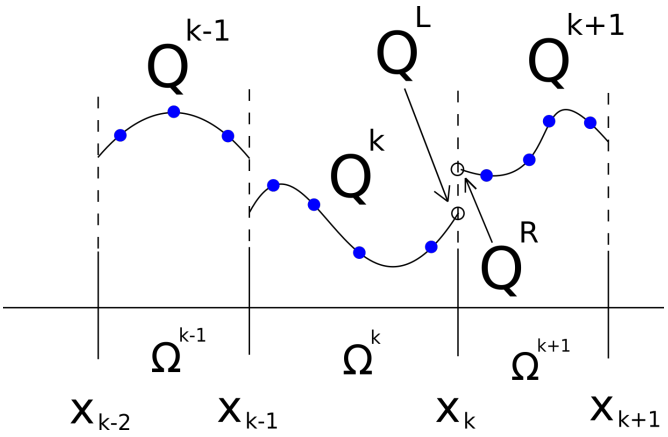


Figure 2.13: Solution discontinuity between elements. The solutions Q^k of kth element (Ω^k) are defined on the Gauss points inside the element region. The solutions (Q^L on the left and Q^R on the right) on the element interface x_k between element Ω^k and Ω^{k+1} are not required to be equal.

We divide the domain into K_x and K_y sub-intervals in the x and y directions respectively. The nth sub-interval in the x direction and the mth sub-interval in the y direction are:

$$\begin{aligned} \Omega_x^n &= [x_{n-1}, x_n], n = 1, \cdots, K_x, \\ \Omega_y^m &= [y_{m-1}, y_m], m = 1, \cdots, K_y. \end{aligned} \tag{2.167}$$

There is no restriction on the relative size or number of the sub-intervals. The sub-intervals form our spectral elements. We can rewrite Equation 2.166 as the sum of the integrations over

each sub-interval:

$$\sum_{m=1}^{K_y}\sum_{n=1}^{K_x}\int_{y_{m-1}}^{y_m}\int_{x_{n-1}}^{x_n}\mathbf{q}_t\phi dxdy+\sum_{n=1}^{K_x}\int_{x_{n-1}}^{x_n}\mathbf{f}_x\phi dx+\sum_{m=1}^{K_y}\int_{y_{m-1}}^{y_m}\mathbf{g}_y\phi dy=0. \tag{2.168}$$

Therefore, for one element we get

$$\int_{y_{m-1}}^{y_m}\int_{x_{n-1}}^{x_n}\mathbf{q}_t\phi dxdy+\int_{x_{n-1}}^{x_n}\mathbf{f}_x\phi dx+\int_{y_{m-1}}^{y_m}\mathbf{g}_y\phi dy=0. \tag{2.169}$$

In DG-SEM, the approximation is coupled to the boundary and to neighbour elements through the fluxes ($\mathbf{f}_x$ and $\mathbf{g}_y$). The discontinuities between sub-intervals are treated as Riemann problems.

We map each element interval $x\in[x_{n-1},x_n]$ to the reference space $\xi\in[-1,1]$:

$$x=x_{n-1}+\frac{\xi+1}{2}(x_n-x_{n-1})=x_{n-1}+\frac{\xi+1}{2}\Delta x_n. \tag{2.170}$$

Then we can deduce

$$dx=\frac{\Delta x_n}{2}d\xi,\quad \frac{\partial}{\partial x}=\frac{\partial}{\partial\xi}\frac{\partial\xi}{\partial x}=\frac{2}{\Delta x_n}\frac{\partial}{\partial\xi}. \tag{2.171}$$

Supposing we are at the $(n,m)th$ element (nth element in the x direction, mth element in the y direction), by changing the integrals in Equation 2.169 from elemental intervals to reference intervals, we obtain

$$\frac{\Delta y_m}{2}\frac{\Delta x_n}{2}\int_{-1}^{1}\int_{-1}^{1}\mathbf{q}_t\phi d\xi d\eta+\int_{-1}^{1}\mathbf{f}_x\phi d\xi+\int_{-1}^{1}\mathbf{g}_y\phi d\eta=0. \tag{2.172}$$

The rest of the derivation is the same as in Section 2.3.2: we apply integration by parts to the flux integrals and evaluate the integrals by Gauss quadrature. Finally, we need to couple the elements. We again use numerical fluxes $\mathbf{F}^*$ to replace the fluxes at element boundaries. The numerical fluxes can be computed as Riemann problems, involving the two states on the element interface: one on the left and another one on the right. Using the element ordering in Fig. 2.13, the numerical fluxes of the nth element (changing the index k to n) in x direction can be computed as:

$$\begin{aligned}\mathbf{F}(-1)&\approx\mathbf{F}^*\left(Q^{n-1}(1),Q^n(-1),-\widehat{x}\right),\\ \mathbf{F}(1)&\approx\mathbf{F}^*\left(Q^n(1),Q^{n+1}(-1),+\widehat{x}\right).\end{aligned} \tag{2.173}$$

Replacing the fluxes at element boundaries by the approximated numerical fluxes in Equation 2.173, and dividing Equation 2.172 by $\frac{\Delta y_m}{2}\frac{\Delta x_n}{2}$ we obtain the final form of the discontinuous

Galerkin approximation of the $(n,m)\,th$ element:

$$\begin{aligned}
&\frac{d\mathbf{Q}^{n,m}(\xi_i,\eta_j)}{dt} \\
&+\frac{2}{\Delta x_n}\left\{\left[\frac{l_i(-1)}{w_i^{(\xi)}}\mathbf{F}^*\left(Q^{n-1,m}(1),Q^{n,m}(-1),-\widehat{\xi}\right)+\frac{l_i(1)}{w_i^{(\xi)}}\mathbf{F}^*\left(Q^{n,m}(1),Q^{n+1,m}(-1),+\widehat{\xi}\right)\right]\right. \\
&\left.+\sum_{k=0}^{N}\mathbf{F}(\xi_k,\eta_j)\widehat{D}_{i,k}^{(\xi)}\right\} \\
&+\frac{2}{\Delta y_m}\left\{\left[\frac{l_j(-1)}{w_j^{(\eta)}}\mathbf{F}^*\left(Q^{n,m-1}(1),Q^{n,m}(-1),-\widehat{\eta}\right)+\frac{l_j(1)}{w_j^{(\eta)}}\mathbf{F}^*\left(Q^{n,m}(1),Q^{n,m+1}(-1),+\widehat{\eta}\right)\right]\right. \\
&\left.+\sum_{l=0}^{M}\mathbf{G}\left(\xi_i,\eta_l\right)\widehat{D}_{j,l}^{(\eta)}\right\}=0,\quad i=0,...,N.\quad j=0,..,M.
\end{aligned} \tag{2.174}$$

Since each of the elemental integrations is zero, the sum of the integration over each sub-interval, as in Equation 2.168, equals zero. From Equation 2.174, we can observe that except for the element size factor $\frac{2}{\Delta x_n}$ and $\frac{2}{\Delta y_m}$, the approximation is almost the same as the single element approximation (Equation 2.69). The elements are coupled together by the numerical fluxes, which is the only information neighbour elements need to exchange. This feature contributes to an easy-to-parallelize numerical scheme. As long as the fluxes are done spectrally, the DG-SEM formulation exhibits exponential convergence despite the discontinuities [40].

Implementation of Solver

In this section, we present succinct algorithms for the whole DG-SEM wave equation solver.

The overall procedure is shown as Algorithm 2.4 *Driver_for_DG_approximation*. After one constructs the basis data for the approximation and initializes the solutions to the mesh based on the initial conditions, the DG approximation starts. Every time step is integrated by the Runge-Kutta method. Inside the time integration algorithm, time derivatives are computed based on Equation 2.174. We present the algorithm for the time derivative in Algorithm 2.5 *DG_time_der*.

Algorithm 2.4 Driver_for_DG_approximation: driver for DG-SEM approximation.

Use Algorithms: **Algorithm 2.3 DG_step_by_RK3**

Algorithm 2.5 DG_time_der

1: Construct basis data: GL points, weights, first derivative matrix, Lagrange interpolation polynomials.
2: Initialize the solution.
3: **for** $i = 0 \rightarrow total\ time\ steps$ **do** ▷ Marching in time.
4: **for** $k = 0 \rightarrow 3$ **do** ▷ Time integration by third-order Runge-Kutta method.
5: Get the current time derivatives: use *DG_time_der*.
6: Time integration and get the solutions at the current time step: use *DG_step_by_RK3*.
7: **end for**
8: **end for**

According to Equation 2.174, the element time derivatives are the negative sum of the element spatial derivatives in each direction. The spatial derivatives are the sum of the element interface fluxes and the element interior fluxes. Therefore, we first interpolate the solutions to the element boundaries. Then we solve the numerical fluxes as a Riemann problem. Finally, with the numerical fluxes and the element interior fluxes, we obtain the element spatial derivatives and the time derivatives. The process is shown in Algorithm 2.5 *DG_time_der*.

Algorithm 2.5 DG_time_der: compute the time derivatives.

Use Algorithms: **Algorithm 2.1 Riemann_x**

Algorithm 2.2 Riemann_y

1: Use Lagrange interpolation to interpolate the interior solutions to the element interfaces.
2: Compute the element numerical fluxes: use *Riemann_x*, *Riemann_y* ▷ Solve element boundary fluxes by using Riemann solvers.
3: Compute the element interior fluxes.
4: Calculate the element spatial derivatives then obtain the time derivatives.

2.3.3 Benchmark Solution: Plane Wave Propagation

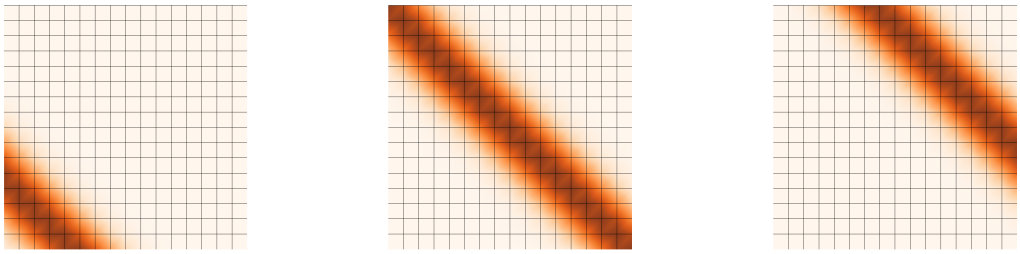
We use a Gaussian plane wave propagation problem as a benchmark solution to Equation 2.46 to showcase the performance of the discontinuous Galerkin spectral element method.

The plane wave solution is defined by

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{k_x}{c} \\ \frac{k_y}{c} \end{bmatrix} e^{-\frac{\left[k_x(x-x_0)+k_y(y-y_0)-ct\right]^2}{d^2}}, \tag{2.175}$$

where the k_x and k_y make up the wavevector $\mathbf{k} = k_x\widehat{x} + k_y\widehat{y}$. The magnitude of the wavevector $k = |\mathbf{k}|$ is the wavenumber. We normalize the wavevector to one: $k_x^2 + k_y^2 = 1$. c is the speed of

sound, here we set it to be one, *i.e.*, $c = 1$. The parameter d is given by $d = \frac{0.2}{2\sqrt{ln2}}$. The x_0 and y_0 give the initial location of the wave. We set them both to be zero, so that the wave starts from the lower left corner of the square domain $[0, 1] \times [0, 1]$. The configurations of the wave at three time snapshots are shown in Fig. 2.14. These are the numerical solutions obtained by the DG-SEM method using 256 elements and polynomials of order 6 using a sufficiently small time step $\Delta t = 1 \times 10^{-5}$. The exact solutions are imposed on the boundaries.



(a) $t = 0.2$ (b) $t = 0.7$ (c) $t = 1.0$

Figure 2.14: Propagation of Gaussian plane wave at three time snapshots. Colour scheme: the magnitude of the pressure. Domain size: $[0, 1] \times [0, 1]$. The wave propagates from the lower left corner to the upper right corner. The DG-SEM numerical solution is obtained by using $K = 16 \times 16 = 256$ elements, polynomials of order $p = 6$ in each direction and a time step of $\Delta t = 1 \times 10^{-5}$.

Convergence Performance

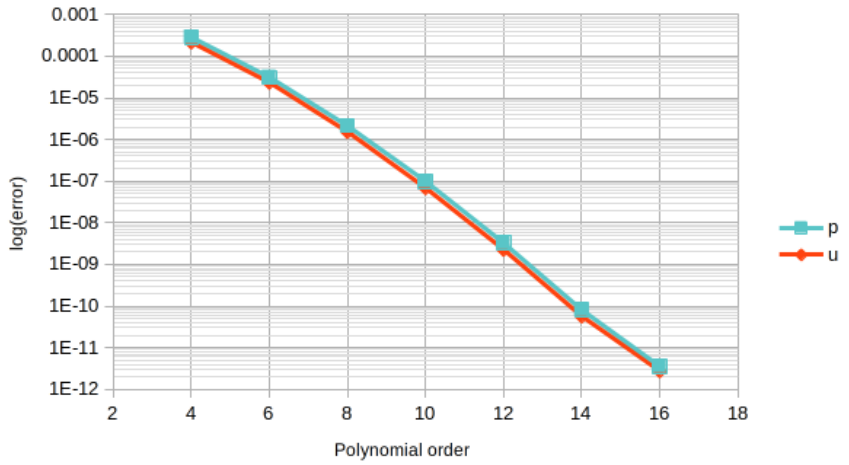


Figure 2.15: Exponential error convergence is obtained for both pressure p and velocity u. The green line represents the $L2$ error in the pressure. The orange line is for velocity. The two lines both obey the exponential convergence rate.

As mentioned previously, one of the strengths of DG-SEM approximation is that it provides exponential convergence of the error with polynomial order as confirmed by Fig. 2.15, where we present the $L2$ norm of the error in the variables pressure p and velocity u.

Chapter 3

Adaptive Mesh Refinement

Modern supercomputers enable scientists and engineers to tackle large scale problems. However, applying a static uniform fine mesh to the whole computational domain can be extremely expensive for a grand challenge problem, and sometimes impractical. For example, Vinuesa *et al.* [78] simulated the turbulent flow around a NACA4412 wing section with a uniform C-mesh shown in Fig. 3.1. This large-scale problem utilized 3.2 billion grid points. To obtain a reasonable execution time, the calculation took place on the Beskow HPC system in Sweden with 16,384 cores. 35 million CPU hours were needed to get the convergence of turbulence.

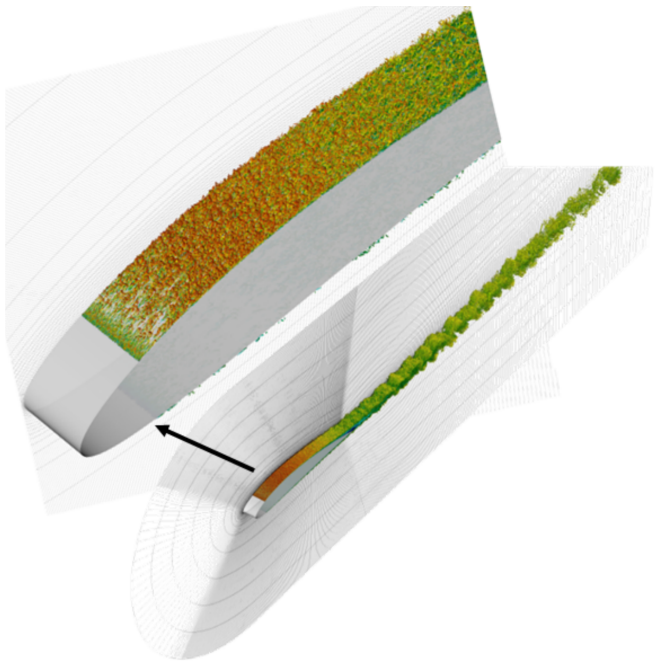


Figure 3.1: Large-scale problem example: simulation of turbulent flow around a NACA4412 wing section with a uniform C-mesh. 3.2 billion grid points are used to get a well-resolved result. Even with 16,384 cores, 35 million CPU hours are needed to get the convergence of turbulence. Reprinted from [78].

Further, many CFD problems aim to track features that are much smaller than the overall scale of the problem. In some problems high resolutions are only required for computationally difficult regions. If we take the simulation of the NACA4412 wing section as an example, since the study only focused on the flow characteristics around the wing, a small grid domain is constructed. However, to reproduce free flight conditions, the domain needed to be $13,000$ times larger [78]. Enlarging the applied mesh in Vinuesa *et al*'s work to the required scale for free flight conditions would result in an unrealistic computation. Moreover, the fine mesh far away from the wing configuration is a waste since the important turbulent development congregates around the wing.

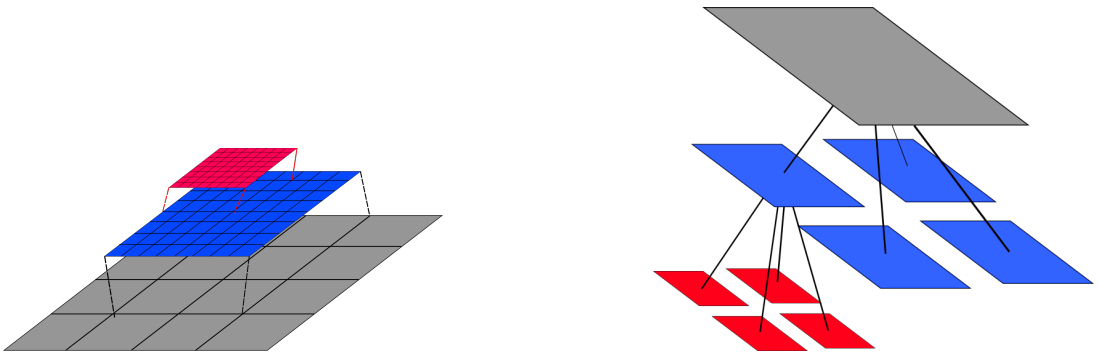
An intuitive strategy to tackle this problem is to generate high-resolution meshes in needed computational regions. Berger and Oliger [10] first proposed the concept of *adaptive mesh refinement and coarsening (AMR)* in 1984. In [9], Berger shows AMR can save a factor of 8 in CPU time and a factor of 2.2 in memory, comparing with uniform fine meshes for the prescribed resolution in their 2D hyperbolic conservation laws solver. Bell *at el.* [8] expand Berger's AMR algorithm to 3D for solving hyperbolic systems of conservation laws. In their work, the memory overhead is reduced by a factor of 22, and the net speedup with AMR is a factor of 55 in comparison with a static uniform grid that achieves the same resolution. Thus, AMR provides high computational savings over static meshes. In our work, we combine DG-SEM with AMR to obtain a computationally efficient and also resolutionally potent program.

In this chapter, we address the issues pertinent to adaptive mesh refinement. We adopt a structured grid AMR strategy, as described in Section 3.1, where we discuss two existing approaches: block-structured and tree-structured AMR. Section 3.2 gives the refinement options relevant to high order methods: h- and p-adaptivity. In order to adapt, error estimators are needed to trigger refinement/coarsening in regions of poor/superfluous resolution: *a posteriori* error estimators based on the high order approximation are given in Section 3.3. Refinement criteria are then developed in Section 3.4. Finally, we integrate the refinement strategies into the wave equation solver and showcase the validity and efficiency of our AMR strategies. Results are presented in Section 3.5.

3.1 Structured Grid AMR

In this work, structured grid AMR is implemented on a Cartesian grid. Although unstructured grids have a higher potential to deal with complex geometries and boundaries [48], the discretization scheme for control-volume-based finite element method is generally more cumbersome than those using structured meshes. Generally, structured grids have a low overhead, both from the computational and storage points of view. The unstructured mesh point renumbering process could become the bottleneck of transient problem applications [49]. Since different data types (point-, element-, face- and edged-based) are employed in unstructured meshes, to keep them related to each other in a dynamic mesh application incurs performance penalty. Moreover,

the search operations involved in unstructured meshes require excessive CPU-time [50]. Thus, structured mesh AMR can benefit from the flexibility in mesh discretization while maintaining the computation efficiency embedded in Cartesian grids.



(a) Block-structured AMR. A layer of mesh represents a hierarchy.

(b) Tree-structured AMR. One node in the tree-structure represents an individual cell.

Figure 3.2: The data structures of two AMR approaches. Colour schemes represent different hierarchies of mesh. (a) Blocked-structured AMR. (b) Three-structured AMR

.

Two popular AMR approaches are block-structured AMR (Fig. 1.4a) and tree-structured AMR (Fig. 1.4b). Block-structured AMR "patches" (Fig. 3.2a) the coarse mesh with smaller sizes of uniform fine meshes where high-resolution is needed [9]. The different layers of meshes are defined with corresponding hierarchies. Since the element sizes are the same on each patch, the algorithm can be coded cheaply and the uniform-mesh code can be reused on different hierarchies of patches. However, this introduces numerical complexity due to the different resolutions among patches, especially for high-order discretization [8]. Additionally, block-structured AMR is limited in flexibility in the sense that this congregating way of defining hierarchy of meshes could result in highly refined meshes in places that have no need for them [37].

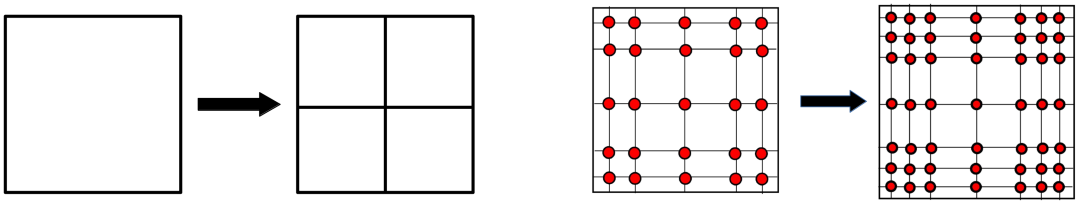
Tree-structured AMR uses a quadtree/octree (2D/3D) data structure to represent the data hierarchies. Unlike block-structured AMR that uses a layer of grid to represent a hierarchy, in tree-structured AMR, one node represents an individual element or cell (Fig. 3.2b). This type of refinement also uses recursive algorithms but allows non-overlapping refinement, which ensures higher computational efficiency than block-structured AMR. And since refinement can be imposed locally, tree-structured AMR offers a better control of grid resolution. Tree-structured adaptive mesh refinement toolkits have been developed in applications like *p4est* [16].

However, it is difficult to implement and maintain a tree-structured data structure, especially in parallel. Moreover, getting access to the neighbour cells, one needs to traverse the tree-structure and find the nearest common ancestor shared between its neighbour and itself. Tree traversal is hard to parallelize and vectorize. In the worst case, one has to traverse to the root node in the tree [69], which will largely degrade the performance of a finite volume method based

CFD program that needs to frequently query cell neighbours. Some progress has been made to increase the efficiency of the neighbour query operation. In Khokhlov's work [39] a *fully threaded tree (FTT)* is developed to offer fast neighbour access. Pointers are used to record the memory addresses of one cell's parent, children and neighbours. The finite element library *LibMesh* [71] replicates the global mesh on all the processors to assist the tree operations. However, by doing so, the memory consumption will grow linearly with respect to the problem scale. *p4est* proposes scalable algorithms to distribute and maintain the tree structures in parallel. Each octant/element has a globally unique ID, and algorithms are offered to compute all the possible neighbours' IDs [16].

In our work, we proposed to use a new data structure to tackle the data storage and management problem: *a hash table AMR*. The hash table offers us high control of mesh resolution like the tree structure. It supports efficient random access of the items inside the table which favours our neighbour query operation. The idea of hash table AMR is easy to understand and implement. We present and explain our approach in Chapter 5.

3.2 hp-adaptivity



(a) h-refinement: subdividing one element into four children elements.

(b) p-refinement: increasing element's polynomial order. In this case both x and y direction's polynomial orders are increased by two.

Figure 3.3: hp-refinement illustration (2D).

In this work, two types of refinement methods are invoked: *h-refinement* and *p-refinement* (Fig. 3.3). h-refinement subdivides a target element into smaller children elements. In our work, one element is divided into four equally-sized children elements (2D). p-type refinement is exclusive to high-order methods. The mesh resolution can be improved by increasing its polynomial order with respect to the target direction. In our work, one refined element gets an increment of polynomial order in both x and y directions by two.

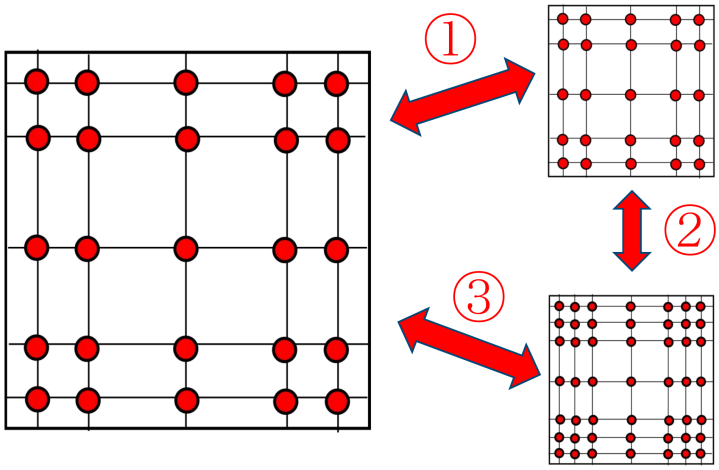
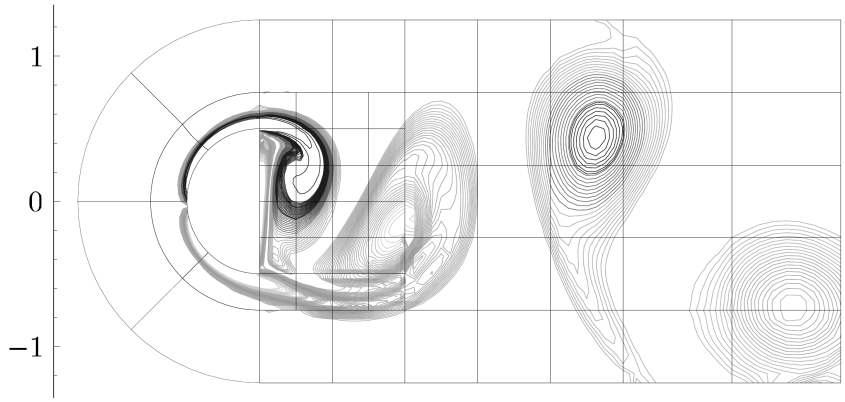


Figure 3.4: Three types of element non-conforming interfaces. Type one: geometrically non-conforming. Type two: functionally non-conforming. Type three: geometrically and functionally non-conforming.

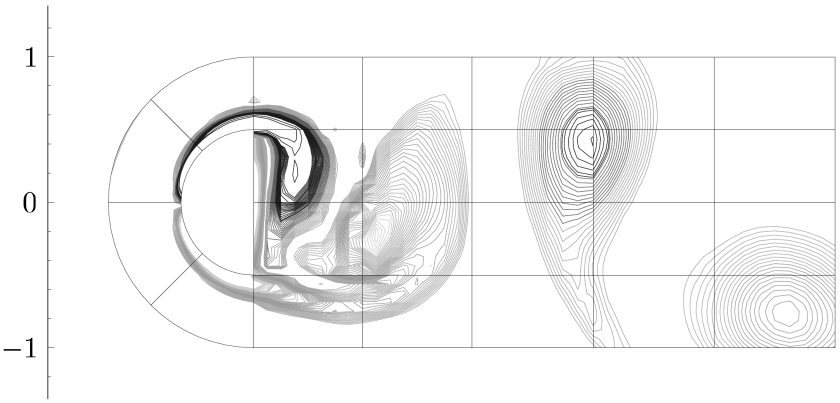
For adjacent elements who share element interfaces with the same polynomial order on both side, such interfaces are defined as *conforming interfaces*. On the contrary, if the element interface is not exactly the same as both sides or the polynomial order is different on either side, then these are *non-conforming interfaces* (Fig. 3.4). In Fig. 3.4, we present three types of element non-conforming interfaces. The red dots in the figure represent the Gauss-Legendre points. If two elements are different sizes but share the same polynomial order (case ①), the interface is called geometrically non-conforming. The interface is called functionally non-conforming if different polynomial orders are used its two sides (case ②). Case ③, geometrical non-conforming and functionally non-conforming, is a combination of cases ① and ②.

Decomposing the computational domain into non-conforming quadrilateral/hexahedral (2D/3D) elements in structured meshes or into triangles/tetrahedra in unstructured meshes is called *unstructured discretization*. The irregular mesh generated by unstructured discretization allows one to improve efficiency of the calculation by concentrating resolution to where it is needed. It provides crucial resolution improvements to the unresolved grids of the computational domain. Karniadakis and Sherwin [38] use the calculation of an unsteady flow past a half circular cylinder on a conforming and a non-confirming grid to show how unstructured discretization benefits the simulation. In Fig. 3.5, 276 elements are used in the conforming case and 176 elements in the non-conforming case. The contours of vorticity are presented in the figure. Comparing Fig. 3.5a with Fig. 3.5b, the second figure is clearly more "noisy" than the first one. Since the vorticity is obtained from $\bigtriangledown \times \mathbf{v}$, the "noise" in the conforming mesh solution can be interpreted as a measure of the under-resolution of the small-scale features [38]. The non-conforming discretization of the domain isolates singular corners within a few smaller elements close to the cylinder

surface and resolves the fine scales present in the forming vortex. Therefore, non-conforming discretization can be very important to fully resolve the small-scale features of a complex flow. Moreover, fine meshes can be avoided in the regions of smooth solutions, for instance, far away from the cylinder in Fig. 3.5a. Thus, nonconforming discretizations save both computational time and storage.



(a) Non-conforming discretization. 176 elements are used.

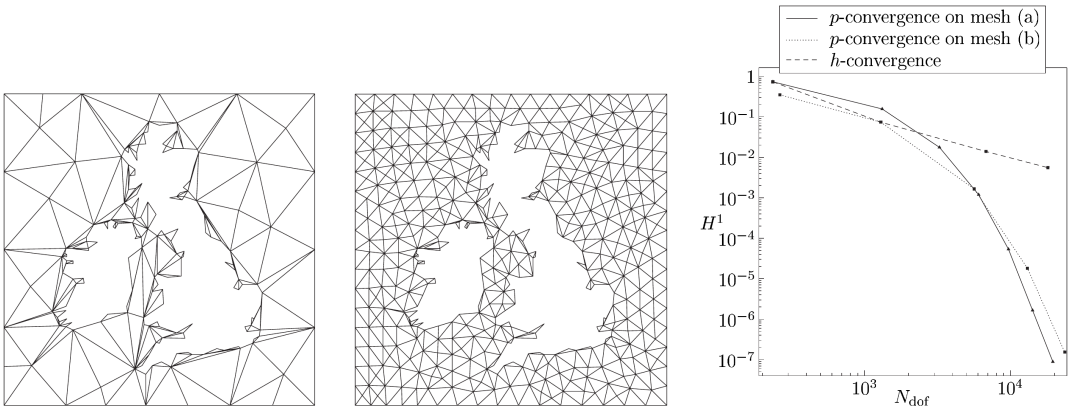


(b) Conforming discretization. 276 elements are used.

Figure 3.5: Vortex shedding simulation using (a) non-conforming and (b) conforming discretization. Shown are instantaneous vorticity contours in the near-wake. (Reprinted from [38].)

hp-type refinement has better convergence than single h-type or p-type refinement. The numerical error decays algebraically in h-type refinement. As for p-refinement, when the solutions are smooth, exponential convergence is obtained, which is much faster than h-type refinement. An example [38] is given to illustrate the characteristics of these two types of refinement methods. An elliptic Helmholtz problem is solved on an unstructured mesh. The $H1$ error norm is plotted as a function of the total number of degrees of freedom. h-type convergence is obtained by refining the meshes (from mesh (a) to mesh (b)) keeping the polynomial order as 2. p-type

convergences are obtained by increasing the polynomial discretization on meshes (a) and (b) keeping a static elemental mesh. As one can observe in Fig. 3.6c, at the beginning, h-refinement resolves the solution faster than p-refinement, but then the asymptotic exponential convergence of the p-refinement outperforms h-refinement. This example demonstrates a valuable strategy to take advantage from both of the refinement methods: when the solution is poor, h-refinement should be imposed to eliminate the numerical error. Once a smooth solution is obtained, exponential convergence can be performed by prescribing p-refinement. This strategy is employed in our refinement algorithm. Combining h- and p-type refinements provide us the fast convergence.



(a) Coarse mesh

(b) Fine mesh

(c) h-type and p-type convergence comparison

Figure 3.6: Convergence comparison between h- and p-type refinement for an elliptic Helmholtz problem. Plot of the $H1$ norm error as a function of the total number of degrees of freedom for coarse (a) and fine (b) meshes. (Reprinted from [38].)

To accelerate the convergence, strategic decisions should be made between h- and p-type refinements. Therefore, algorithms are needed to measure the error of the solutions and guide the decision process. A promising error estimator has been proposed by Mavriplis [52]. Details of the error estimator are presented in the next section (Section 3.3). Additionally, non-conforming interfaces complicate the numerical scheme between elements.

For AMR applications, to treat interfaces between non-conforming elements in order to retain spectral accuracy, we use the *mortar method* [52, 42], explained in Chapter 4.

3.3 Error Estimator

Adaptive mesh refinement applications require programs to identify the computationally difficult regions and resolve the small-scale fluid flows with respect to the prescribed accuracy. Therefore, we have two major concerns in developing our AMR strategy: how to locate the regions needing

resolution and how to decide which type of refinement to implement. In our work, the solver detects the undersolved regions by evaluating the local numerical errors.

3.3.1 Truncation Error

We use orthogonal polynomial series to approximate the solutions in non-periodic problems. Here we use Legendre polynomials to develop spectral approximation to PDEs.

A function in $[-1, 1]$ can be represented by Legendre polynomials as an infinite series:

$$u(x) = \sum_{n=0}^{\infty} a_n L_n(x), \tag{3.1}$$

where $L_n(x)$ is the Legendre polynomial of degree n and a_n are the coefficients, also known as the spectrum.

For numerical computation, we take a finite sum (to N) of Equation 3.1. So our approximation becomes:

$$\begin{aligned} u(x) &= \sum_{n=0}^{\infty} a_n L_n(x) \\ &= \sum_{n=0}^{N} a_n L_n(x) + \sum_{n=N+1}^{\infty} a_n L_n(x) \\ &= \sum_{n=0}^{N} a_n L_n(x) + \tau \\ &\approx \sum_{n=0}^{N} a_n L_n(x), \end{aligned} \tag{3.2}$$

where $\tau = \sum_{n=N+1}^{\infty} a_n L_n(x)$ is the truncation error. To obtain the norm of the truncation error $\|\tau\| = \|u_{trunc}\|$ we need the value of the coefficients a_n beyond N. We make use of the orthogonality property of the Legendre polynomials:

$$(L_n, L_m) = \int_{-1}^{1} L_n(r) L_m(r) dr = \frac{2}{2n+1} \delta_{nm}, \tag{3.3}$$

where δ_{nm} is the Kronecker delta. Using Equation 3.3, the coefficients a_n can be founded by taking the inner product of the approximated solutions $u_h(x)$ with the corresponding Legendre polynomial $L_n(x)$:

$$\begin{aligned} (u_h, L_n) &= \int_{-1}^{1} u_h(x(r)) L_n(r) dr \\ &= \sum_{k=0}^{N} a_k \int_{-1}^{1} L_k(r) L_n(r) dr \\ &= \frac{2}{2n+1} a_n. \end{aligned} \tag{3.4}$$

Therefore, the coefficients a_n are:

$$a_n = \frac{2n+1}{2} \int_{-1}^{1} u_h(x(r)) L_n(r) dr. \tag{3.5}$$

The truncation error is the part of the spectral approximation beyond N. Therefore, we can only predict the error instead of actually calculating it. The predicted $\widetilde{a}_n$ are computed by extrapolation of the a_n we have. Thus, the numerical $L2$ error due to truncation is approximated as:

$$\begin{aligned} \|u_{trunc}\| &= \left(\int_{-1}^{1} \sum_{n=N+1}^{\infty} \left(\widetilde{a}_n L_n(r)\right)^2 dr \right)^{1/2} \\ &= \left(\sum_{n=N+1}^{\infty} \widetilde{a}_n^2 \int_{-1}^{1} L_n(r) L_n(r) dr \right)^{1/2}. \end{aligned} \tag{3.6}$$

Using the orthogonality of Legendre polynomials shown by Equation 3.3 we obtain the final expression for the truncation error:

$$\|u_{trunc}\| = \left(\sum_{n=N+1}^{\infty} \frac{\widetilde{a}_n^2}{\frac{(2n+1)}{2}} \right)^{1/2}. \tag{3.7}$$

In 2D it is

$$\|u_{trunc}\| = \left(\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}. \tag{3.8}$$

For a non-linear system, part of the spectrum ($\widetilde{a_n}$) will give an accurate approximation to the true solutions and part of it will be polluted [32]. The order of the magnitude of the local error is examined by the decay rate of the tail of the local polynomial spectrum. The decay rate indicates that when the polynomial basis fails to give a good description of the solution. This method is suitable for sufficiently high polynomial order, typically for polynomial order $p \geq 5$, so that the local spectrum can have an exponential decay [32, 52].We evaluate the $\widetilde{a}_n$ by extrapolation. The distribution of the spectrum $\widetilde{a}_n$ is treated as an exponential decay [52]:

$$\widetilde{a}_n = Ce^{-\sigma n}, \tag{3.9}$$

where C is a constant. Taking logarithms of both sides, one obtains a linear relationship between spectrum $\widetilde{a}_n$ and decay rate σ

$$ln(\widetilde{a}_n) = ln(C) - \sigma n. \tag{3.10}$$

We apply a linear least squares best fit to the last four points of the spectrum $\widetilde{a}_n$. Then we obtain the slope of the straight line, $-\sigma$, expressed by Equation 3.10, or the *error indicator*. The magnitude of the error indicator characterizes the quality of the approximation results. For a well-resolved solution we expect a fast decay of a_n, *i.e.*, $\sigma > 1$ [53].

Recall that the linear least squares best fit finds the best-fitting straight line to a given set of points by minimizing the sum of the squares of the offsets of the points from the line. So to find the best-fitting line

$$y = \alpha + \beta x, \tag{3.11}$$

we have

$$\beta = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2}, \quad i = 1, ..., n, \tag{3.12}$$

$$\alpha = \overline{y} - \beta\overline{x}, \tag{3.13}$$

where $\overline{x}, \overline{y}$ are the averages of the data. Then we obtain the decay indicator $\sigma = -\beta$, and $C = e^{\alpha}$.

3.3.2 Quadrature Errors

Since we approximate integrals by Gauss quadrature, it is also possible that there are errors in the spectrum.

If the real solution to the numerical problem is

$$u(x) = \sum_{n=0}^{N} b_n L_n(x), \tag{3.14}$$

and the approximated solution is

$$u_h(x) = \sum_{n=0}^{N} a_n L_n(x), \tag{3.15}$$

then we get the additional error as

$$\begin{aligned} \|u_{quad}\| &= \left\{ \int_{-1}^{1} \sum_{n=0}^{N} \left[b_n L_n(r) - a_n L_n(r) \right]^2 dr \right\}^{1/2} \\ &= \left(\sum_{n=0}^{N} \frac{(b_n - a_n)^2}{\frac{2n+1}{2}} \right)^{1/2}, \end{aligned} \tag{3.16}$$

here $\|\cdot\|$ is the $L2$ norm.

The quadrature error evaluation depends on the choice of collocation points. For Gauss-Legendre-Lobatto (GLL) quadrature, the b_n are obtained exactly for integrands of polynomial degree $n \leq 2N - 1$. For $n \geq 2N$, the coefficients are approximated. For most of the integrals in our wave equation solver have at most integrands of degree $2N$ for linear advection. For non-linear terms, both present in the Navier-Stokes equations, the integrands are of degree $> 2N$. Therefore, we must consider the quadrature error:

$$\|u_{quad}\| = \left(\frac{(b_N - a_N)^2}{\frac{2N+1}{2}} \right)^{1/2}, \tag{3.17}$$

and in order to provide an upper bound, we take

$$\|u_{quad}\| \approx \left(\frac{(a_N)^2}{\frac{2N+1}{2}} \right)^{1/2}. \tag{3.18}$$

For 2D

$$\|u_{quad}\| \approx \left(\sum_{j=0}^{M} \frac{(a_{Nj})^2}{\frac{2N+1}{2}} + \sum_{i=0}^{N} \frac{(a_{iM})^2}{\frac{2M+1}{2}} \right)^{1/2}. \tag{3.19}$$

If Gauss-Legendre (GL) points are used, then the b_n can be exactly approximated for integrands of polynomial degree $n \leq 2N+1$. Therefore, there is no quadrature error for linear problems. In this work, GL points are used since they provide a more precise approximation than GLL points, and we do not need element boundary points.

3.3.3 Summary

The approximated error is a combination of truncation error and quadrature error. If GLL quadrature is used, we estimate the error as

$$1D: \quad \epsilon_{est} \approx \left(\frac{a_N^2}{\frac{2N+1}{2}} + \sum_{n=N+1}^{\infty} \frac{\widetilde{a}_n^2}{\frac{2n+1}{2}} \right)^{1/2}, \tag{3.20}$$

$$2D: \quad \epsilon_{est} \approx \left(\sum_{j=0}^{M} \frac{(a_{Nj})^2}{\frac{2N+1}{2}} + \sum_{i=0}^{N} \frac{(a_{iM})^2}{\frac{2M+1}{2}} + \sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}. \tag{3.21}$$

If GL quadrature is used, only truncation error is involved, then the approximated errors are:

$$1D: \quad \epsilon_{est} \approx \left(\sum_{n=N+1}^{\infty} \frac{\widetilde{a}_n^2}{\frac{2n+1}{2}} \right)^{1/2}, \tag{3.22}$$

$$2D: \quad \epsilon_{est} \approx \left(\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}. \tag{3.23}$$

The infinite sums of Equations 3.22 and 3.23 can be written as integrals (the denominators are omitted to provide an upper bound):

$$1D: \quad \epsilon_{est} = \left(\int_{N+1}^{\infty} a_n^2 dn \right)^{1/2}, \tag{3.24}$$

$$2D: \quad \epsilon_{est} = \left(\int_{M+1}^{\infty} \int_{N+1}^{\infty} (a_{nm})^2 dn dm \right)^{1/2}. \tag{3.25}$$

With these expressions and Equation 3.9 we are able to evaluate the truncation error. For the

1D case, the evaluation is straightforward:

$$\begin{aligned}\epsilon_{est} &= \left(\int_{N+1}^{\infty} a_n^2 dn\right)^{1/2} \\ &= \left(\int_{N+1}^{\infty} \left(Ce^{-\sigma n}\right)^2 dn\right)^{1/2} \\ &= \left(\frac{C^2}{2\sigma}\right)^{1/2} e^{-\sigma(N+1)}.\end{aligned} \tag{3.26}$$

For the 2D case, the spectrum in Equation 3.25 is not a simple monotonically decaying spectrum. In fact, it forms a 2D mesh. Thus, we need to average the spectrum to produce a single spectrum as in the 1D case. In this work, we assume the same polynomial order in x and y directions. We produce an equivalent one-dimensional spectrum $\overline{a}_n$ as:

$$\overline{a}_N = |a_{N,N}| + \sum_{i=0}^{N-1} \left(|a_{i,N}| + |a_{N,i}|\right). \tag{3.27}$$

Next, we can use this averaged spectrum to estimate the error

$$\begin{aligned}\epsilon_{est} &= \left(\int_{M+1}^{\infty}\int_{N+1}^{\infty} (a_{nm})^2 dndm\right)^{1/2} \\ &= \left(\int_{N+1}^{\infty} \overline{a}_p^2 dp\right)^{1/2} \\ &= \left(\int_{N+1}^{\infty} Ce^{-2\sigma p} dp\right)^{1/2} \\ &= \left(\frac{C}{2\sigma} e^{-2\sigma(N+1)}\right)^{1/2} \\ &= \left(\frac{C}{2\sigma}\right)^{1/2} e^{-\sigma(N+1)},\end{aligned} \tag{3.28}$$

where the error indicator σ is calculated by linear least squares best fit to the spectrum $\overline{a}_n$ as explained in Section 3.3.1.

3.4 Refinement Criteria

The refinement criterion is simple: refine the element when its estimated error $\epsilon_{est}^{(k)}$ exceeds the threshold:

$$\epsilon_{est}^{(k)} \geq \epsilon \left\| u_h^{(k)} \right\|, \tag{3.29}$$

where ϵ is the discretization tolerance. The $L2$ norm is taken on a single (kth) element.

Given estimated errors on each element, the refinement is applied to those elements that do not fulfill criterion 3.29. The decision between the h- and p-type refinement is given by the error indicator σ. $\sigma > 1$ indicates that the spectrum has an exponential decay rate, which implies a

smooth solution, so we prescribe p-refinement. Otherwise, if $\sigma \leq 1$, then the local solution is not well resolved and further p-refinement will not be efficient until we reach the exponential decay rate regime; so h-refinement should be applied. This idea is illustrated in Fig. 3.7.

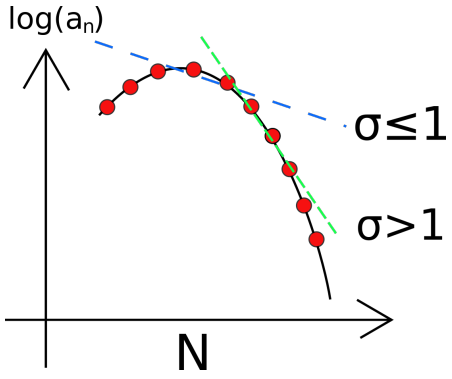


Figure 3.7: Spectrum decay trend: when the solution is smooth, the spectrum a_n has an exponential decay, and the magnitude of the slope of the least squares best fit is larger than 1 (green dashed line). When the solution is poor, the magnitude of the slope will be less than 1 (blue dashed line).

Coarsening is applied when the estimated error is smaller than the defined smallest tolerance, ϵ_{min}:

$$\epsilon_{est}^{(k)} \leq \epsilon_{min} \left\| u_h^{(k)} \right\|. \tag{3.30}$$

When an element is flagged to be coarsened, it first decreases its polynomial order since high-order polynomials are more computationally expensive ($O(N^4)$). h-coarsening is prescribed when all children elements from the same parent are flagged to be coarsened.

The error estimator makes good use of the feature of the local polynomial spectrum, which makes it work independently of the system of equations being solved.

3.5 Adaptivity Performance

3.5.1 Solution Resolution Improvement Demonstration

In the former sections, we introduced AMR, the potency of hp-type refinement, the mathematical theory to identify computationally difficult regions, the strategy of hp-refinement, and finally, the refinement criteria. Thus, now we have all the pieces to form a valid and efficient AMR algorithm. The *a posteriori* error estimator is applied to the wave equation solver. The solution qualities of a coarse mesh and a refined mesh are compared.

The AMR strategy is examined on the wave solver by using the plane wave benchmark solution we proposed in Section 2.3.3. The computation takes place on domain $[0, 1] \times [0, 1]$. Sound

speed is kept as $c = 1$. The wavevector is chosen to be $\mathbf{k} = (0, 1.0)$, therefore, we simplify the problem to a 1D one (Fig. 3.8). We set $x_0 = y_0 = 0$. Then we compare the computed solution to the exact one on a cross-section perpendicular to $x = 0.55$ at time $t = 0.5$. We use a time step $\Delta t = 1.0 \times 10^{-5}$ and refinement frequency $f = 500$ (refine every 500 steps). Refinement tolerance is set at $\epsilon = 10^{-6}$. The simulation starts with a uniform grid of four elements (two elements in each direction) and polynomial order six in each direction on each element.

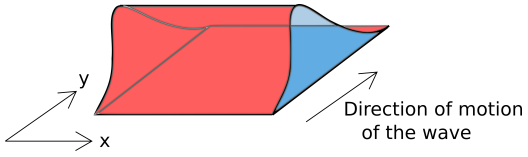
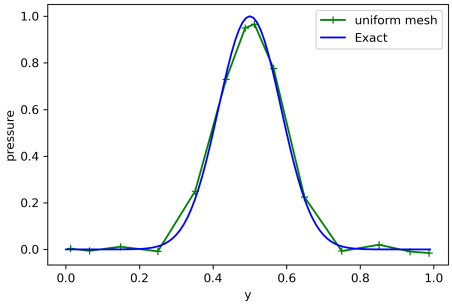
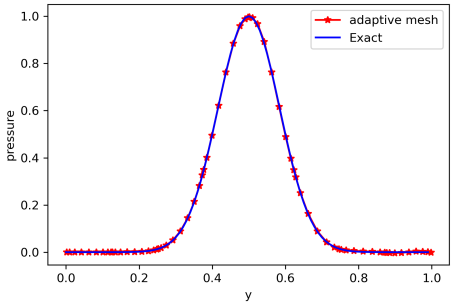


Figure 3.8: Wave illustration.



(a) DG-SEM uniform grid solution. $K = 2$ elements with polynomial order $p = 6$ in the y direction. The green line stands for the computed solution and the blue line the exact solution. Some oscillations can be observed from the computed result.



(b) Adaptive gid DG-SEM solution. hp-refinement is triggered based on the error estimator. On the cross-section, $K = 8$ elements are used with polynomial orders p ranging from 6 to 8. The red line represents the computed solution. The computed solution after refinement coincides well with the exact solution (blue line) with minimum oscillations.

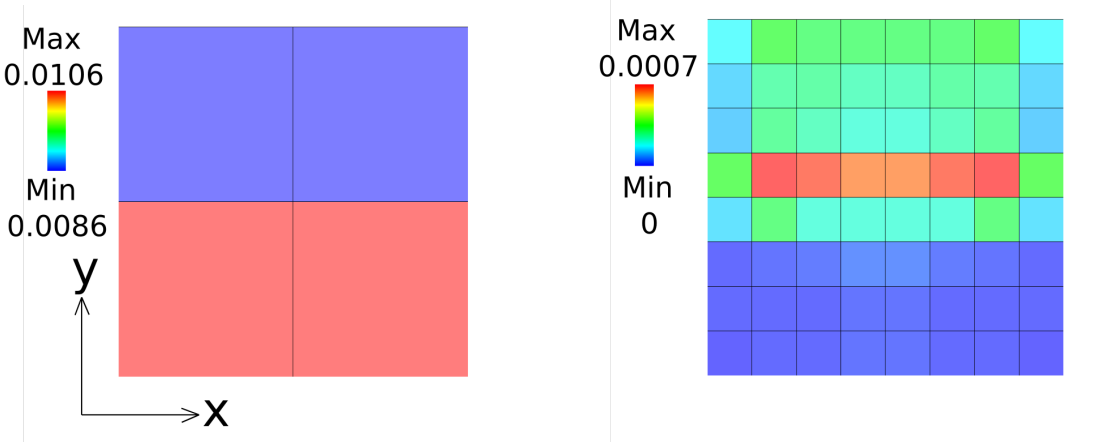
Figure 3.9: Computed and exact pressure are compared on the cross-section $x = 0.55$ at time $t = 0.5$.

	Uniform mesh	Adaptive mesh
$L2$ norm	0.019	0.002

Table 3.1: The values of the $L2$ norm error on the domain at time $t = 0.5$ for uniform mesh and adaptive mesh.

The computed pressure on a coarse mesh is presented in Fig. 3.9a. Two elements with polynomial order 6 are applied. The computed solution is shown by the green line and the exact

solution by the blue line. Oscillations are observed. In the AMR result (Fig. 3.9b), the approximated solution (red line) coincides well with the exact solution. Due to hp-refinement, by time $t = 0.5$, $K = 8$ elements are used on the cross-section with polynomial orders p ranging from 6 to 8. The mesh resolution is largely improved after the refinement. The $L2$ norms of the error of both cases are presented in Table 3.1. The error decreases by an order of magnitude after the refinement.



(a) Pressure solution error of uniform mesh. The colour bar ranges from 0.008 to 0.01.

(b) Pressure solution error of adaptive mesh. The colour bar range from 0 to 7×10^{-4}.

Figure 3.10: Computed error in solution at time $t = 0.5$. The magnitude of the $L2$ norm of the error is presented by the colour scheme. Note that the range of the colour bars are different between the two figures.

Fig. 3.10a provides an overall view of the error distribution on the domain. By comparing Fig. 3.10a with Fig. 3.10b, it is clear that the magnitudes of the error are largely reduced (indicated by the colour bars): the largest error on the refined mesh is 700 times smaller than that on the uniform mesh. The elements which reside in $0 \leq y \leq 0.5$ have smaller error values than elements in $0.5 \leq y \leq 1.0$ in Fig. 3.10b, since higher p-refinements are employed in the former region.

Next, let us have a closer look at the refinement result. In Fig. 3.11, a red dashed line separates the cross-section into two parts. Elements in the left part have polynomial order $p = 8$ and those in the right part have polynomial order $p = 6$. This is because at time $t = 0.5$, the wave propagates to the middle of the domain, *i.e.*, $y = 0.5$. As the wave reaches the left half of the domain first ($y = 0 \sim 0.5$), the elements residing there employed p-refinement. Since a finer mesh is formed on the left part of the domain, the solutions where the left arrow point are smoother than where the right arrow points. Further refinement are prescribed to the region on the right side of the dashed line at later time.

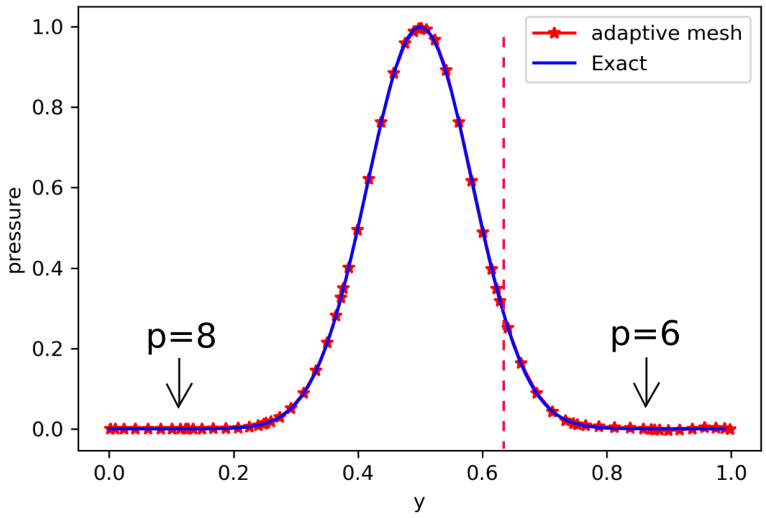
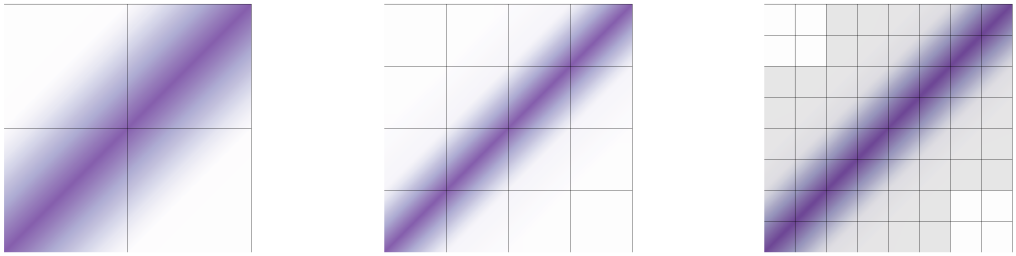


Figure 3.11: Adaptive mesh solution: computed and exact pressure at time $t = 0.5$, $K = 8$ elements are used on the cross-section. The elements on the left side of the red dashed line have polynomial order $p = 8$ and elements on the right have polynomial order $p = 6$.

By using this experiment, we demonstrate how AMR can improve the mesh resolution and contributes to a more precise result. More importantly, it also verifies the validity and efficiency of our hp-refinement strategy.

3.5.2 Experiment One: Resolve Plane Wave Starting from a Coarse Mesh

In this section, we apply hp-refinement to the plane wave benchmark problem presented in Section 2.3.3 with wavevector $\mathbf{k} = \left(\sqrt{2}/2, -\sqrt{2}/2\right)$ and sound speed $c = 1$. We show how computationally difficult regions are identified and spatial and temporal high-resolutions are triggered at the needed places. Finally, the wave is fully-resolved starting from a coarse mesh.



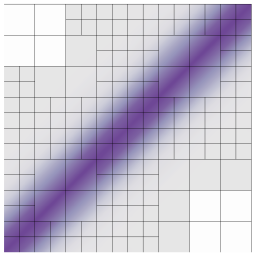
(a) The simulation starts with a coarse mesh of 4 elements. Time $t = 0$. $K = 4$, $p = 6$.

(b) h-refinement is applied to improve the mesh resolution. Time $t = 2.0 \times 10^{-5}$. $K = 16$, $p = 6$.

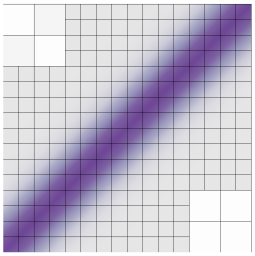
(c) h-refinement is applied to the far-field regions and hp-refinement is invoked in the wave regions. Time $t = 5.0 \times 10^{-5}$. $K = 64$, $p = 6 \sim 10$.

Figure 3.12: AMR Experiment One Scene One: resolve a Gaussian plane wave starting from a coarse mesh. The configuration of the pressure is shown in purple. The polynomial order p of each element is indicated by the degree of greyness in the figure, ranging from 6 to 12. K is the number of elements.

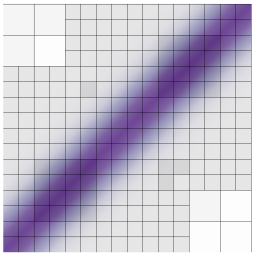
We start the simulation from a coarse mesh: only $K = 4$ elements (Fig. 3.12a) with polynomial order $p = 6$ on domain $[0, 1]^2$. The wave starts from the centre of the domain ($x_0 = y_0 = 0.5$). The time step is chosen as $\Delta t = 10^{-8}$. The refinement tolerance as $\epsilon = 10^{-6}$. The domain is initialized with the exact solution. Exact boundary conditions are applied to the four boundaries of the domain. The h-refinement level ranges from 0 to 3 and the polynomial order ranges from 6 to 12. At the initial state the wave is under-resolved. Then, the program identifies the regions that need finer meshes. From Fig. 3.12a to Fig. 3.12b, while only h-refinement is invoked in all four elements to improve the mesh resolution. As a result, the wave is better resolved. Next, in Fig. 3.12c, hp-refinement is applied in the wave area and h-refinement is placed in the far-field regions.



(a) More aggressive h-refinement is observed in the wave region. Time $t = 6.0 \times 10^{-5}$. $K = 184$, $p = 6 \sim 10$.



(b) Wave region obtains a fine mesh. Time $t = 7.0 \times 10^{-5}$. $K = 232$, $p = 6 \sim 10$.



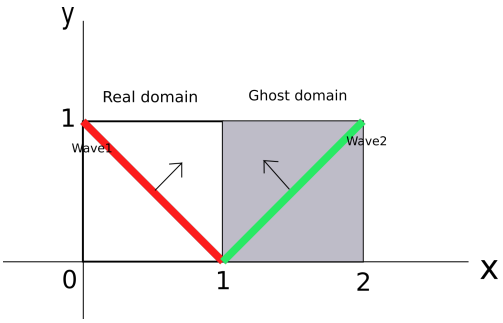
(c) p-refinement is mainly prescribed in the wave region. Time $t = 8.0 \times 10^{-5}$. $K = 232$, $p = 6 \sim 12$.

Figure 3.13: AMR Experiment One Scene Two: resolve a Gaussian plane wave from a coarse mesh. The configuration of the pressure is shown in colour purple. The polynomial order p of each element is indicated by the degree of greyness in the figure, ranging from 6 to 12. K is the number of elements.

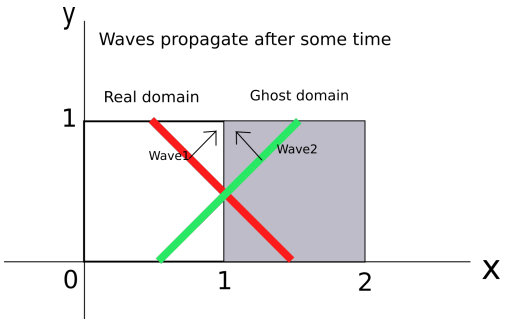
As time marches on, a more aggressive h-refinement is observed in the wave area (Fig. 3.13a and Fig. 3.13b). The wave in Fig. 3.13b is better resolved, compared with the initial mesh solution in Fig. 3.12a. As the mesh reaches its maximum h-refinement level (Fig. 3.13b), more p-refinement is prescribed in the wave region (Fig. 3.13c). This demonstrates that the error estimator successfully identifies the difficult regions, so that high-resolution meshes are triggered there. The solver successfully applies hp-refinement to an extremely coarse mesh and resolves the profile of the wave.

3.5.3 Experiment Two: Reflecting Waves

In this experiment, the same Gaussian wave parameters are kept as in Experiment One on domain $[0, 1]^2$. The time step is chosen as $\Delta t = 10^{-6}$. The refinement is applied every 2×10^3 steps during a runtime of $t = 0.5$. Refinement tolerance is set as $\epsilon = 10^{-6}$. The computation starts with a uniform mesh of $K = 256$ elements with polynomial order $p = 6$. A reflection, *i.e.*, wall boundary condition is applied to the right boundary of the domain. The reflection boundary condition means that the left moving wave w^- is a mirror image of the right moving wave w^+ at the boundary, *i.e.*, $w^- = w^+$ (Fig. 2.5). Thus, the vertical velocity on the reflecting boundary is zero (the boundary is similar to a wall). We use the *method of images* to derive the boundary conditions.



(a) The Wave 1 is reflected across the right boundary.

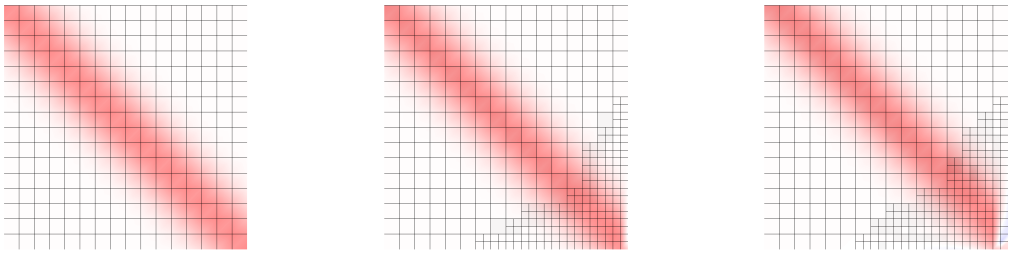


(b) The reflection of Wave 1 can be seen as Wave 2 moves inside the real domain from a mirror image of the ghost domain.

Figure 3.14: Wave 1 (in red) is propagating towards the upper right corner of the real domain. Its reflection on the right boundary can be seen as a mirror image wave (in green), moving with the same vertical velocity (with respect to the reflection boundary) in magnitude but in the opposite direction.

The initial wave (Wave 1) starts from the centre of the domain and propagates towards to upper right corner (Fig. 3.14a). By reflecting, a second wave (Wave 2) is formed. Wave 2 is a reflection of Wave 1. It can be seen as a mirror image of Wave 1 across the right boundary and propagates from the ghost domain to the real domain (Fig. 3.14b). Therefore, the imposition of the reflecting wall boundary condition is equivalent to the superposition of the wave and its image.

We initialize Wave 1 with the exact solution, therefore, it has a smooth solution profile at the beginning. As time marches on, the reflected wave (Wave 2) starts to form, but with a relatively poor solution. We compare the refinement strategies imposed on the two different waves.



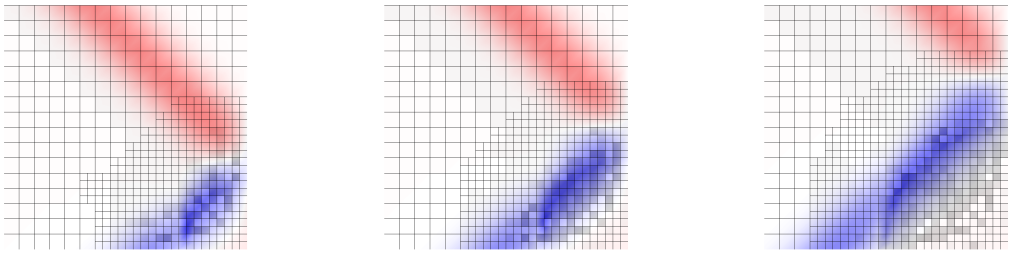
(a) Initialize Wave 1 (red wave) with exact solution. Time $t = 0$. $K = 256$. $p = 6$.

(b) Mainly h-refinement is prescribed at the lower right corner where Wave 2 starts to form. Time $t = 0.006$. $k = 382$. $p = 6 \sim 8$.

(c) Wave 2 (coloured in blue) appears. h-refinement congregates at the lower right corner. p-refinement refinement is assigned along Wave 1. Time $t = 0.038$. $K = 406$. $p = 6 \sim 10$.

Figure 3.15: AMR Experiment Two Scene One: compare the refinement strategies applied to a well-resolved wave (red) and a mal-resolved (reflected) wave (blue). The magnitude of the velocities u are shown in red and blue. The polynomial order p of each element is indicated by the degree of greyness in the figure. K is the number of elements.

From Fig. 3.15a to Fig. 3.15c, one can observe that massive h-refinement is triggered to alleviate the error at the lower right corner of the domain, where Wave 2 starts to form. As for Wave 1, p-refinement is prescribed in the wave region, indicating that a relatively smooth solution, thus p-refinement is chosen to effectively resolve the wave.



(a) h-refinement is triggered in the region of interaction between the two waves. Time $t = 0.258$. $K = 517$. $p = 6 \sim 14$.

(b) p-refinement captures the trajectory of Wave 2 (blue). Time $t = 0.358$. $k = 574$. $p = 6 \sim 14$.

(c) High-order elements (deep grey cells) are congregated in the Wave 2 region, indicating the program is trying to resolve Wave 2. Time $t = 0.500$. $k = 697$. $p = 6 \sim 14$.

Figure 3.16: AMR Experiment Two Scene Two: compare the refinement strategies applied to a well-resolved wave (red) and a mal-resolved (reflected) wave (blue). The magnitude of the velocities u are shown in red and blue. The polynomial order p of each element is indicated by the degree of greyness in the figure. K is the number of elements.

Fig. 3.16 showcases how the program utilizes the AMR strategy to tackle the difficult regions. h-refinement is prescribed at the interaction region of the two waves. p-refinement is triggered dynamically on the trajectory of Wave 2. Comparing the mesh around the two waves, it is clear that the error estimator efficiently identifies the difficult areas and employ h- and p-type refinements according to the solution qualities. With the ability to correctly perform hp-refinement, a cost-efficient application is demonstrated.

In summary, the validity and the efficiency of the error estimator is indicated by the experiments above. Starting with a coarse mesh, the program successfully applies hp-refinement to the computational domain and gives sufficient resolution to the important regions. To get an fast convergence, the program should make decisions between h- and p-refinement. From what we observe, when the mesh is coarse, mainly h-refinement is triggered. After the mesh resolution is highly improved around the wave, p-refinement starts to accelerate the convergence inside the wave. With our hp-adaptive strategy, a cost-efficient solver is formed. The error estimator we adopted lays a solid foundation for the efficiency of the AMR technique.

Chapter 4

Non-conforming Mortar Element Discretization

In this section, we first explain element non-conforming interfaces (Section 4.1). Then the mortar element method is presented in Section 4.2. Finally, an example is given to justify the validity of the mortar element method.

4.1 Element Interfaces

Two adjacent elements with the same size of element interfaces and the same polynomial discretization share a *conforming* interface. Otherwise, if one of the two criteria is violated, they have a *non-conforming* interface. Three types of non-conforming interfaces are incurred by hp-adaptivity (Fig. 4.1).

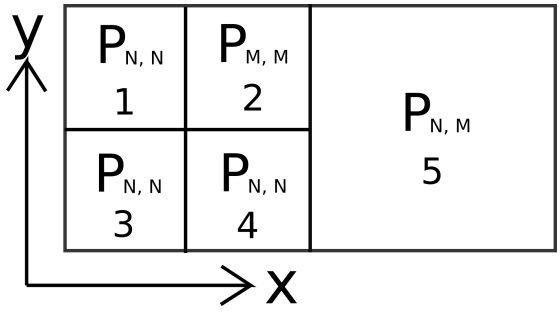


Figure 4.1: Element interfaces: an element with polynomial order N in x direction and M in y direction is represented as $P_{N,M}$. Element 1 and 3 share conforming interfaces. Element 2 and 5 are geometrically non-conforming. Element 2 and 4 are functionally non-conforming. Element 4 and 5 are geometrically and functionally non-conforming.

In a hp-adaptive application, four element interface circumstances can be encountered: conforming (element 1 and element 3 in Fig. 4.1), functionally non-conforming (element 2 and element 4), geometrically non-conforming (element 2 and element 5), geometrically and functionally non-conforming (element 4 and element 5). Extra work needs to be performed at element interfaces to ensure the consistency of the numerical computation.

In order to tackle all the possible interface combinations while maintaining the exponential convergence of the spectral element method, the mortar element method is invoked [42, 52]. The mortar element method restrains the projection error between element interfaces to be as small as the spectral approximation error. Explanations and derivations are presented in the following sections.

4.2 Mortar Element Method

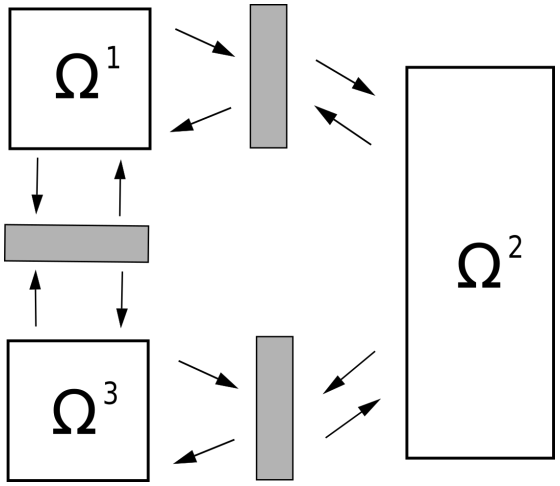


Figure 4.2: Elements use mortars to communicate between interfaces. Ω^k represents kth element and the grey blocks are the mortars between elements.

When solving the hyperbolic wave equation, elements need to communicate the numerical fluxes. This requires the grid points on the two adjacent element interfaces to coincide [41], *i.e.*, the element interfaces have to be conforming. In order to adapt the solver to deal with non-conforming circumstances, a "patch" is put between two non-conforming elements, which is the so-called mortar (the "cement"). For 2D elements, the mortar is a 1D polynomial function (indicated by the grey blocks in Fig. 4.2). The mortar method for elliptic problems is slightly different from the mortar method for hyperbolic problems. For elliptic problems, details can be found in [52, 4, 12]. Here we consider the mortar method for hyperbolic problems [41, 42].

For hyperbolic problems, where the system of equations are treated in conservation form, two conditions need to be fulfilled across the element interfaces. The first condition is that the

approximation retains global conservation. The second one requires that it satisfies the *outflow condition*. The outflow condition means that, at the element interface, the solution from which the characteristic comes ("upwind side") is used. So the wave passes though an interface, unaffected by the downwind contribution. Therefore, it is crucial to make sure the solution on the upwind side interface remains unchanged after projecting it to the mortar and also when projecting back to the element [42]. Both the outflow condition and the conservation can be satisfied by the least squares matching of the face and mortar solution [42], so the wave can propagate through non-conforming interfaces properly.

The mortar projection is explained in the subsections below.

4.2.1 From Element to Mortar

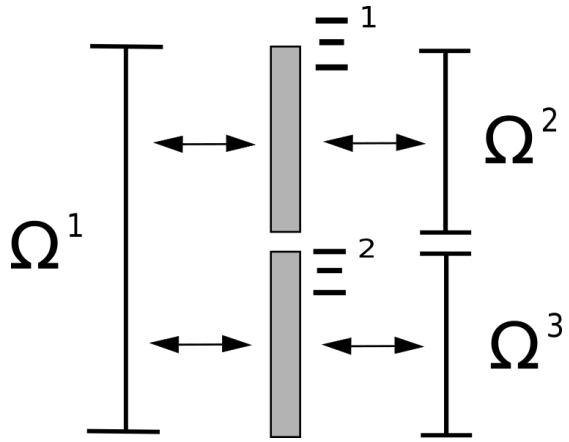


Figure 4.3: Mortar configuration. Ω refers to the element and Ξ represents mortar.

Fig. 4.3 shows the mortar configuration. We always define the mortar's size as the size of the smaller element on the interface. In this topology, "L" (left) and "R" (right) are used to refer to the interface's side. Ξ represents the mortar and Ω represents the element.

The solution U on the element boundary and its projection Ψ on the mortar are discretized as

$$U(s) = \sum_{k=0}^{N} U(s_k) l_k(s), \tag{4.1}$$

$$\Psi(z) = \sum_{k=0}^{J} \Psi(z_k) l_k(z), \tag{4.2}$$

where l is the Lagrange polynomial. s is the element side coordinate and z is the mortar side coordinate. The projection is imposed on the mortar, thus the reference space is on the mortar, *i.e.*, $z \in [-1, 1]$. The corresponding coordinate on the element can be computed as $s = a + bz$, where a and b are two scalars.

N is the polynomial order of the contributing element. To fulfill the outflow condition, the polynomial order of the mortar space must be sufficiently large, so we take $J = max(N^L, N^R)$.

To project the solution U on element interfaces to the mortar, we require

$$\int_{-1}^{1} \left(\Psi(z) - U(a+bz)\right) l_j^{\Xi} dz = 0, \quad j = 0, ..., J. \tag{4.3}$$

Substituting Equation (4.1) and (4.2) into (4.3) we get

$$\int_{-1}^{1} \sum_{m=0}^{J} \Psi(z_m) l_m^{\Xi}(z) l_j^{\Xi}(z) dz = \int_{-1}^{1} \sum_{k=0}^{N} U(a+bz_k) l_k^{\Omega}(a+bz) l_j^{\Xi}(z) dz, \quad j = 0, ..., J. \tag{4.4}$$

We can write Equation (4.4) in matrix form

$$M\Psi = SU, \tag{4.5}$$

where

$$M = \int_{-1}^{1} l_m^{\Xi}(z) l_j^{\Xi}(z) dz, \quad j, m = 0, 1, ..., J, \tag{4.6}$$

$$S = \int_{-1}^{1} l_k^{\Omega}(a+bz) l_j^{\Xi}(z) dz, \quad j = 0, 1, ..., J,\ k = 0, 1, ..., N. \tag{4.7}$$

M is a $(J+1) \times (J+1)$ matrix and S is a $(J+1) \times (N+1)$ matrix. $M\Psi,\ SU \in \mathbb{R}(J+1)$.

Therefore, in order to get Ψ we bring M to the other side of the equation

$$\Psi = M^{-1}SU = P^{\Omega \to \Xi} U. \tag{4.8}$$

Here, we define $P^{\Omega \to \Xi} = M^{-1}S$ be to the *projection matrix*.

Then we use *Gaussian quadrature* to evaluate M and S.

$$\begin{aligned} M_{jm} &= \int_{-1}^{1} l_m^{\Xi}(z) l_j^{\Xi}(z) dz \\ &= \sum_{k=0}^{J} l_m^{\Xi}(z_k) l_j^{\Xi}(z_k) w_k^{\Xi} \\ &= \sum_{k=0}^{J} w_k^{\Xi} \delta_{mk} \delta_{jk}, \quad j, m = 0, ..., J, \end{aligned} \tag{4.9}$$

where w_k^{Ξ} is the Gauss Legendre weight for the Gauss point on the mortar. The product of $l_m^{\Xi}(z) l_j^{\Xi}(z)$ has the property of:

$$l_m^{\Xi}(z_k) l_j^{\Xi}(z_k) = \delta_{mk}\delta_{jk} = \begin{cases} 1, & if \quad m = k = j, \\ 0, & if \quad m \neq k \ or \ j \neq k. \end{cases} \tag{4.10}$$

Therefore, we can simplify Equation 4.9 into a diagonal matrix:

$$M_{jm} = \begin{cases} w_j^{\Xi}, & m = j \\ 0, & m \neq j \end{cases}. \tag{4.11}$$

And for S we have

$$S_{jk} = \int_{-1}^{1} l_k^{\Omega}(a+bz) l_j^{\Xi}(z) dz = \sum_{m=0}^{J} l_k^{\Omega}(a+bz_m) l_j^{\Xi}(z_m) w_m^{\Xi} = \sum_{m=0}^{J} l_k^{\Omega}(a+bz_m) w_m^{\Xi} \delta_{jm}. \tag{4.12}$$

S is simplified based on

$$l_j^{\Xi}(z_m) = \delta_{jm} = \begin{cases} 1, & if \quad m = j \\ 0, & if \quad m \neq j. \end{cases} \tag{4.13}$$

Thus, we obtain:

$$S_{jk} = l_k^{\Omega}\left(a + bz_j\right) w_j^{\Xi}, \quad j = 0, ..., J,\ k = 0, ..., N. \tag{4.14}$$

Note that here M and S are evaluated exactly since we use Gauss points and approximate them with polynomial order J.

Therefore, we obtain the projection matrix

$$\begin{aligned} P_{jk}^{\Omega \to \Xi} &= M^{-1} S \\ &= (w_j^{-1})^{\Xi} l_k^{\Omega}(a+bz_j)(w_j)^{\Xi} \\ &= l_k^{\Omega}(a+bz_j), \quad j = 0, ..., J,\ k = 0, ..., N. \end{aligned} \tag{4.15}$$

The L_2 projection from element to mortar is

$$P_{jk}^{\Omega \to \Xi} U_k = \sum_{k=0}^{N} l_k^{\Omega}(a+bz_j) U(a+bz_k), \quad j = 0, ..., J. \tag{4.16}$$

$$= \begin{bmatrix} l_0^{\Omega}(a+bz_0) & \cdots & l_N^{\Omega}(a+bz_0) \\ \vdots & \vdots & \vdots \\ l_0^{\Omega}(a+bz_J) & \cdots & l_N^{\Omega}(a+bz_J) \end{bmatrix} \cdot \begin{bmatrix} U(a+bz_0) \\ \vdots \\ U(a+bz_N) \end{bmatrix}. \tag{4.17}$$

Equation (4.16) shows that the L_2 projection from element to mortar is equivalent to Lagrange interpolation.

4.2.2 From Mortar to Element

The projection from mortar to element is more complicated since there could be multiple mortars contributing to one element. For instance, in Fig. 4.3, Ω_1 should receive the fluxes projected back from two mortars.

Now we seek a flux that satisfies

$$\int_{-1}^{o'} \left(F(s) - \Phi(\frac{s-a}{b}) \right) l_j^{\Omega}(s) ds + \int_{o'}^{1} \left(F(s) - \Phi(\frac{s-a}{b}) \right) l_j^{\Omega}(s) ds = 0, \quad j = 0...N, \quad (4.18)$$

where F is the numerical flux that needs to map to the element interface from the mortar and Φ is the numerical flux we compute on the mortar. Here we take the element interface as the reference space: $s \in [-1, 1]$. So the coordinate transformation from element to the mortar is $z = \frac{s-a}{b}$. Here, a and b inherit the meanings from the former section.

Here we simplified the situation to two mortars facing one element. But the derivation can be extended to more complicated cases.

We rewrite Equation (4.18) as

$$\int_{-1}^{1} F(s) l_j^{\Omega}(s) ds = \int_{-1}^{o'} \Phi(\frac{s-a}{b}) l_j^{\Omega}(s) ds + \int_{o'}^{1} \Phi(\frac{s-a}{b}) l_j^{\Omega}(s) ds. \quad (4.19)$$

Again, we write the solutions in Lagrange interpolation form:

$$F(s) = \sum_{m=0}^{N} F(s_m) l_m^{\Omega}(s), \quad (4.20)$$

$$\Phi(\frac{s-a}{b}) = \sum_{m=0}^{J} \Phi(\frac{s_m - a}{b}) l_m^{\Xi}(\frac{s-a}{b}). \quad (4.21)$$

Here, the order on the mortar must be at least as large as the maximum element order of the all contributing elements. Therefore, $J = max(N^L, N^R)$. $\Phi(\frac{s_m - a}{b})$ are the nodal values on the mortar.

This time we have $MF = S\Phi$. Thus, the projection matrix becomes $P^{\Xi \to \Omega} = M^{-1} S$.

The derivation of matrix M is the same as before:

$$M_{jm} = \int_{-1}^{1} l_m^{\Omega}(s) l_j^{\Omega}(s) ds = \sum_{k=0}^{N} w_k^{\Omega} \delta_{mk} \delta_{jk} = \begin{cases} w_j^{\Omega}, & j = m \\ 0 & j \neq m. \end{cases} \quad (4.22)$$

We obtain a diagonal matrix.

For matrix S we need to change the variable of integration from s on the element to z on the mortar.

$$\int_{-1}^{o'} \Phi(\frac{s-a}{b}) l_j^{\Omega}(s) ds = b \int_{-1}^{1} \sum_{m=0}^{J} \Phi(z_m) l_m^{\Xi} l_j^{\Omega}(a + bz) dz \quad (4.23)$$

So we have

$$\begin{aligned} S_{jm} &= b \int_{-1}^{1} l_m^{\Xi}(z) l_j^{\Omega}(a + bz) dz = b \sum_{k=0}^{J} l_m^{\Xi}(z_k) l_j^{\Omega}(a + bz_k) w_k^{\Xi}, \\ &= b l_j^{\Omega}(a + bz_m) w_m^{\Xi}, \quad j = 0, ..., N, \quad m = 0, ..., J. \end{aligned} \quad (4.24)$$

We notice that $S^{\Xi \to \Omega} = b \left(S^{\Omega \to \Xi} \right)^T$.

Combining all the pieces together we obtain the projection matrix of mortar to element:

$$\begin{aligned} P_{jm}^{\Xi\to\Omega} &= M^{-1}S \\ &= (w_j^{-1})^{\Omega}\, b\, l_j^{\Omega}(a+bz_m)w_m^{\Xi} \\ &= b\, l_j^{\Omega}(a+bz_m)\frac{w_m^{\Xi}}{w_j^{\Omega}}, \quad j=0,...,N,\ m=0,...J. \end{aligned} \tag{4.25}$$

4.2.3 Examples

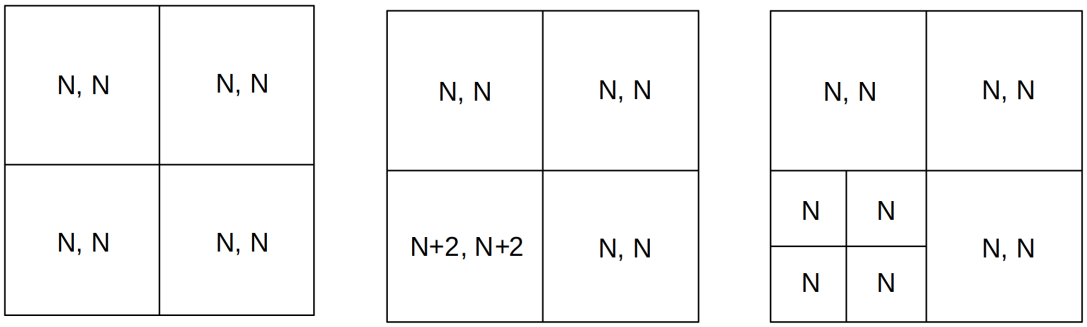
In this section, we showcase the mortar element method's performance on non-conforming interfaces. The wave equation solver is used as a benchmark solution.

Recall the wave equations

$$p_t + c^2(u_x + v_y) = 0, \tag{4.26}$$

$$u_t = -p_x, v_t = -p_y. \tag{4.27}$$

Here we use $c = 1$. The solutions are computed on the domain $[0,1] \times [0,1]$. We expect that the solution to converge exponentially as the polynomial order increases. The results of three cases are compared (see Fig. 4.4).



(a) Conforming interfaces

(b) Functionally non-conforming interfaces

(c) Geometrically non-conforming interfaces

Figure 4.4: Mortar element method demonstration: grids for the convergence using conforming interfaces, functionally non-conforming interfaces, geometrically non-conforming interfaces.

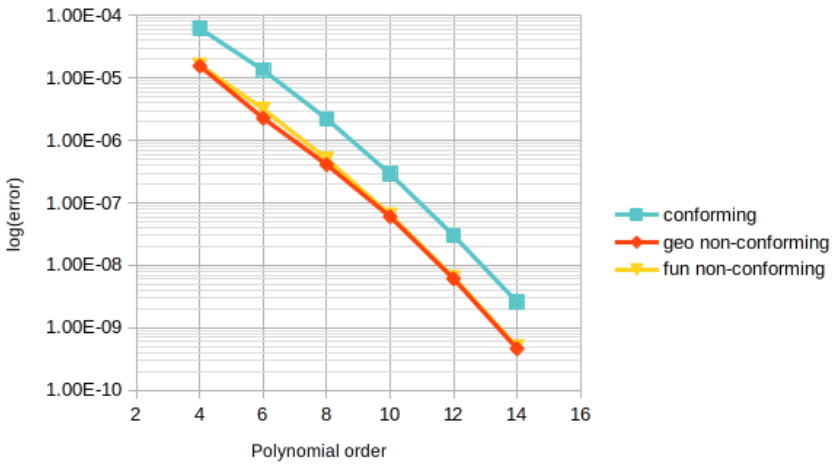


Figure 4.5: A comparison of conforming and non-conforming interface errors as a function of approximation polynomial order. The green line shows the convergence of the conforming interface. The orange line represents geometrically non-conforming interface error convergence. The yellow line is the error performance of functionally non-conforming interface. All of them exhibit exponential convergence, demonstrating the validity of the mortar element method.

The same computation is performed on three different meshes shown in Fig. 4.4: conforming interfaces, functionally non-conforming interfaces, geometrically non-conforming interfaces. We show the $L2$ error norm of the whole domain for each case in Fig. 4.5. The error convergence of the non-conforming cases successfully retains its exponential rate for both the functionally non-conforming case (yellow curve) and the geometrically non-conforming case (orange curve). The mortar element method successfully alleviates the projection error on element interfaces so that it does not exceed the spectral approximation error, thus, exponential convergence is preserved. The L_2 projection technique lays the foundation for a dynamic application. Additionally, the h-type and p-type refinements improve the solution accuracy: both non-conforming cases obtain a smaller error comparing with conforming mesh.

Chapter 5

Data Structure

In this chapter, we first introduce available data structures for AMR encoding (Section 5.1). Next, our proposed hash table AMR is presented in Section 5.2. Finally, we compare the data management operations in the hash table and tree data structure approaches (Section 5.3).

5.1 Data Structures for AMR

In computer science, a data structure is way of organizing and storing data so that the data can be used efficiently. Data structures can be classified into two categories: linear and non-linear. In linear data structures, data items are arranged in an orderly manner, such as arrays and linked lists. The most widely used data structure is the array, whose data is stored in a contiguous memory location. However, an array is a fixed size data structure. For applications with dynamic databases like AMR, the data storage size could vary largely during computation. To deal with this situation using an array, one has two solutions: allocate a large memory buffer for future storage or allocate a new buffer that can hold the new database and copy the data from the old buffer to the new one. Neither way is efficient, since the former one is not memory economic and the latter one is time-consuming ($O(N)$).

A linked-list supports dynamic memory allocation. It offers us the possibility to freely insert or remove elements in the mesh. A linked list has no pre-defined memory size and the order of the data items can be managed by pointers between data items. However, we can no longer access a specific item inside a linked list with constant time complexity ($O(1)$) as with an array. Now, one needs to traverse the linked list to get the target item. This is a linearly time-consuming process ($O(N)$). For a large-scale AMR application with millions of elements, a linear time complexity for random element access is unacceptable.

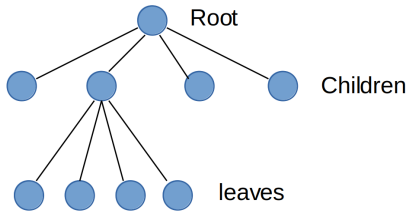


Figure 5.1: (Quad-)tree data structure for 2D meshes. Each node in the tree represents an element in the mesh. In the quadtree data structure, one node is subdivided into 4 children nodes.

According to NASA's *CFD Vision 2030 Study* [2], managing the vast amount of data generated by large-scale simulations will continue to be problematic in years to come. In order to manage the AMR data efficiently and to have good control with the mesh resolution, more complex data structures are invoked. The tree data structure [81], which allows the computation element to be refined or coarsened independently of the other elements, is adopted by many applications. A small example of quadtree data structure (2D grid) is shown in Fig. 5.1: each node in the figure represents an element in the mesh. The first node, or the node at the first level is called the *root* node. One node can be subdivided into four *children* nodes. The nodes at the deepest refinement level are called *leaves*. The divided node is called the *parent* of the four children nodes and the four children nodes are *siblings* to each other. The advantage of using the tree-structure is the flexibility of mesh refinement and coarsening. However, standard grid-based solvers cannot be directly implemented on a tree. The overall time stepping strategy on a tree is different from that on a grid [39]. Moreover, it becomes more difficult to access a neighbour cell in a tree-structure than in an array structure. Nodes in a tree are linked to each other by pointers in a hierarchical manner to facilitate tree traversal and connectivity information retrieval [7]. To access a neighbour cell, one cell has to traverse the tree and find the nearest common ancestor node between its neighbour and itself. In the worst case, a tree must be traversed up to its root. In the case shown in Fig. 5.1, for example, each interior node needs five pointers, the root node has four pointers, and the leaf nodes have no pointer. In this way, the full grid is built on a tree-structure and all the connectivity information is decided by the pointers between the nodes. Although the tree structure offers ultimate refinement flexibility (there is no limitation of refinement level), the memory overhead to maintain a tree is substantial [7]. Additionally, maintaining a tree and performing traversal in parallel poses a huge challenge in parallel AMR.

Khokhlov [39] developed a modified tree-structure called a *fully threaded tree (FTT)* to provide efficient parallel access to the information in a tree and reduce the memory overhead to maintain a tree. By recording pointers to neighbour's parent cells, the FTT can avoid searching elements neighbours by accessing the parent cells. This allows vectorization and parallelization of the tree-based AMR algorithm.

Burstedde *et al.* present their method to encode and maintain the tree-structure AMR in [15, 16, 35]. The algorithms are used in their AMR library *p4est*.

These are all well-developed possible approaches to tackle AMR data-management problem. In this work, a new AMR data structured is proposed: a hash table data structure. The idea itself is inherited from [37]. We modified and tailored it to meet our objectives: (a) a data structure which allows fast and random access to the target elements, (b) offering good control of mesh resolution, (c) which should be memory-usage-friendly.

5.2 Proposed AMR Data Structure: Hash Table AMR

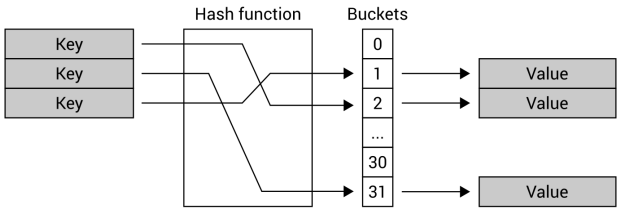


Figure 5.2: Hash table data structure. A value inside a hash table is accessed with its unique key.

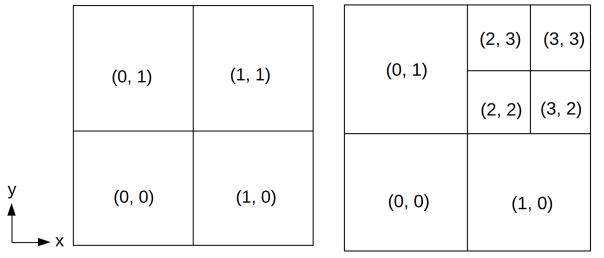
In a hash table data structure, data is stored in an array manner. Every object inside the data is assigned with a unique index (Fig. 5.2). The hash table data structure supports constant time insertion and deletion operations. A specific object in the hash table can be accessed by using its "*key*" value. The hash function translates the key to the object's index. This process is called *hashing*. The hash function links the key with the value, creating an item-access manner similar to that of an array. Therefore, with a unique key, one can access the value in $O(1)$ time. Comparing to $O(logN)$ for a binary tree, where N is the total number of nodes, the random data access process is highly efficient for the hash table data structure. For a CFD application, this feature is important.

The hash table data structure is a non-linear data structure, *i.e.*, the data elements are not managed sequentially and there is no connection between data elements. To maintain a desired order of data elements, a linked list is implemented among the elements in the hash table. The linked list allows the program to loop through the elements with a designed order which is crucial to the load balancing algorithm, which will be discussed in Chapter 7.

5.2.1 Hash Table Construction

The data in a hash table is stored in (key, value) pair manner. With each data value pairing with a unique key, the search, insertion, deletion operations have an average constant time

complexity ($O(1)$) . So ensuring the uniqueness of the key value of each element is important. Since we are dealing with 2D structured meshes in Cartesian coordinates, we can make use of the inherent advantages of Cartesian grids.



(a) Initial mesh coordinates (b) Mesh coordinates after one element refines

Figure 5.3: Element coordinates

We give each element in the mesh an integer coordinate based on its geometrical position. The following example is given in 2D, but the whole idea also works in 3D. Shown in Fig. 5.3a, the elements' coordinates are combinations of their position number in x and y direction (x, y). In Fig. 5.3b, the upper right element refines, generating four children, whose coordinates (x_c, y_c) are computed in the following manner:

$$(x_c, y_c) = (2x_p + n, 2y_p + m), \quad n, m = 0, 1, \tag{5.1}$$

where x_p, y_p are the coordinates of their parent.

One can deduce the parent coordinate from the child coordinate

$$(x_p, y_p) = \left(int(\frac{x_c}{2}), int(\frac{y_c}{2}) \right), \tag{5.2}$$

where $int(\cdot)$ is the operator of taking the integer part of the value inside the brackets.

In order to distinguish elements of different refinement levels, another integer that describes the element's h-refinement level is added to the element coordinates. Element level starts with 0, and increases by 1 every time it subdivides. Thus, an element's coordinates are made up of three parts: integer coordinates in x and y direction, and h-refinement level.

$$Element\, coordinates = (x, y, level). \tag{5.3}$$

The three values in the element coordinate distinguish each element from all others with their positions and refinement levels. However, it is not practical to use a coordinate made up three integers to be the key in a hash table. Therefore, a method to transform the coordinates into a unique integer is needed. In the next section, a pairing function is utilized to perform the coordinate - key mapping.

5.2.2 Pairing Functions

Cantor Pairing Function

In mathematics, a pairing function is a process to uniquely map two natural numbers into a single natural number, *i.e.*, a pairing function is a computable bijection [73]:

$$\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}. \tag{5.4}$$

A widely used pairing function is the *Cantor pairing function*:

$$\pi(k_1, k_2) = \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2, \tag{5.5}$$

where k_1 and k_2 are both non-negative integers.

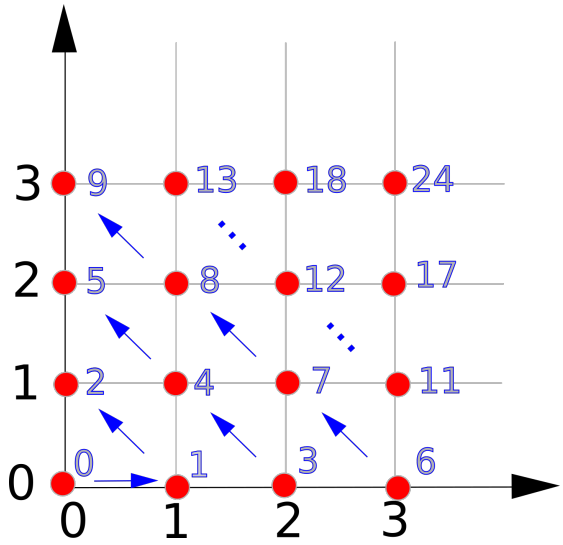


Figure 5.4: Cantor pairing function graphical illustration.

This function assigns consecutive numbers to points along diagonals in the plane as illustrated in Fig. 5.4.

An ordered triple of elements (k_1, k_2, k_3) is defined as an abbreviation for the formula $((k_1, k_2), k_3)$ [25]. So for our application, we want to map a set of three non-negative integers to a one-to-one correspondence integer

$$\pi(k_1, k_2, k_3) = \pi\left(\pi(k_1, k_2), k_3\right). \tag{5.6}$$

This is called a *3-tupling function*, and for higher dimension the logic is the same [66].

Szudzik Pairing Function

A more elegant pairing function is proposed by Szudzik[1]:

$$\pi(k_1, k_2) = \begin{cases} k_1 + k_2^2, & k_1 \neq max(k_1, k_2) \\ k_1^2 + k_1 + k_2, & k_1 = max(k_1, k_2). \end{cases} \tag{5.7}$$

It is a modified version of the *Rosenberg-Strong pairing function* [67]. This pairing function assigns consecutive numbers to points along the edges of squares (Fig. 5.5).

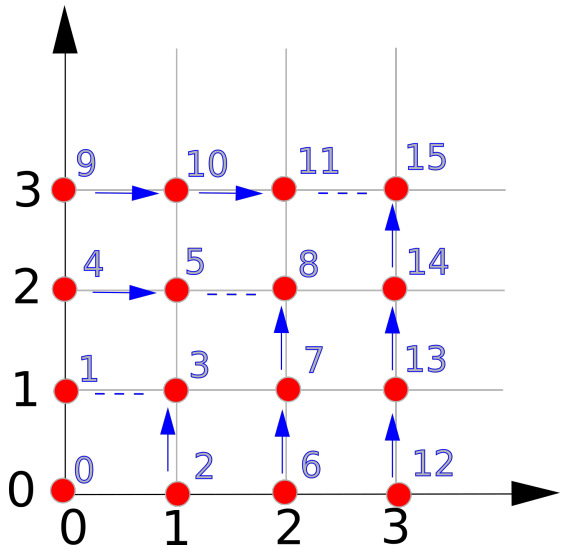


Figure 5.5: Szudzik pairing function graphical illustration.

The value generated by the Szudzik pairing function is more compact than that of the Cantor pairing function, and their computational performances are proven to be almost equivalent. Therefore, we choose to use the Szudzik pairing function to generate the key values for the hash table.

We present Algorithm 5.1 *Szudzik* to compute the bijection mapping number. Then using this function recursively, one can get the key of the target element, giving Algorithm 5.2 *Get_key_fun*. The i, j and k are the element integer coordinates, which are the coordinates in x and y direction, along the element's axes with h-refinement level.

[1] http://szudzik.com/ElegantPairing.pdf

Algorithm 5.1 Szudzik: Szudzik pairing function.

Input: x, y

Output: key

1: **if** $x \geq y$ **then**
2: return $(x^2 + x + y)$
3: **else**
4: return $(y^2 + x)$
5: **end if**

Algorithm 5.2 Get_key_fun: inputs element coordinates, outputs the key of the element.

Input: i, j, k

Output: key

Use Algorithms: Algorithm 5.1 Szudzik

1: $key = Szudzik(i, j)$
2: $key = Szudzik(key, k)$

5.2.3 Element Storage

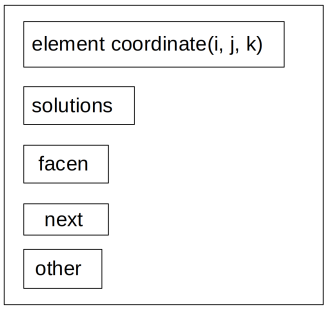


Figure 5.6: Element storage in hash table. Each element stores its coordinate, numerical solutions, neighbour's keys (in $facen$), pointer to the next element ($next$), and other problem related variables.

In our proposed AMR data structure, shown by Fig. 5.6, each element stores its coordinate and the numerical solutions related to the problem. Besides these, the neighbour information is explicitly stored to support direct access. This is a reasonable approach for an application that needs to frequently exchange information between elements. Since the values inside the hash table are accessed by the keys, we store the keys of neighbour elements in the $facen$ variable. Additionally, we number our element in a desired order by a pointer called $next$. The ordering method will be explained in Chapter 7.

Next we compare the memory usage between the tree-structure and the hash table structure

under the same case.

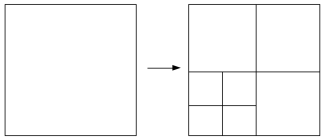
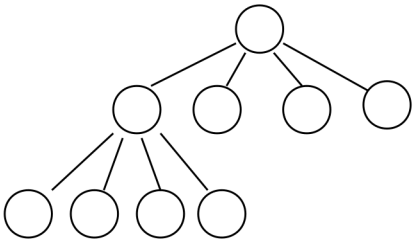
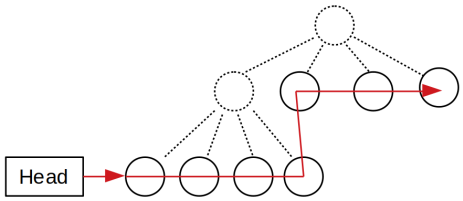


Figure 5.7: Example: left element refines twice and get the right mesh.



(a) Tree data structure. This quadtree has three levels.



(b) Hash table data structure. The elements are linked together through a linked list. The parents' information are discarded (parents are represented by dotted lines).

Figure 5.8: A comparison of two data structures for the same mesh refinement (of Fig. 5.7)

In Fig. 5.7, the left element refines twice to become the right mesh, resulting in a quadtree data structure shown in Fig. 5.8a or a hash table structure shown in Fig. 5.8b. In the tree data structure, 9 elements are stored with 8 pointers to form the tree. In the hash table data structure, since the parents are discarded, we only store 7 elements, with a linked list made up by 7 pointers (6 between elements, and 1 pointer that points to the first element). Thus, we save on memory usage. The memory usage difference between these two data structures is more significant with a larger scale mesh.

For an ordinary quadtree, each node has five pointers: one to its parent and four others to its children. If N inner nodes (node that is not leaf node) are used in a quadtree, then the number of leaf nodes is $(1 + 3N)$. In total, we have $(1 + 4N)$ nodes with $5(1 + 4N)$ pointers. However, by using a hash table data structure, one element only needs one pointer, therefore, in total, with one extra pointer pointing to the head element, $(1 + 4N) + 1 = (2 + 4N)$ pointers are needed. For a large-scale problem with sufficiently large N, the memory occupied by pointers decreases by $\frac{5(1+4N)-(4N+2)}{5(1+4N)} \approx 80\%$ by using the proposed hash table AMR strategy over a quadtree data structure. Thus, a significant memory overhead can be avoided with the hash table data structure.

Full Element Storage Symbols

We disclose here the full element storage to help us explain the algorithms in the following chapters. The variables mentioned in subsequent algorithms related to the elements can be found here. The elements are stored in a local hash table called *local_hash*.

Symbols for element variables	
n	Element polynomial order in x direction.
m	Element polynomial order in y direction.
$index[3]$	Element coordinates.
$status$	Hilbert status.
$child\_position$	ith child among the four siblings.
$facen[4]$	Four neighbours information.
$xcoords[2]$	Physical coordinates in x direction.
$ycoords[2]$	Physical coordinates in y direction.
$solution$	Solutions to the system of equations.
$solution\_int\_l$	Element solution on the left boundary.
$solution\_int\_r$	Element solution on the right boundary.
$ghost$	Ghost layer to store the variables on the MPI boundaries.
$nflux\_l$	Element numerical flux on the left boundary.
$nflux\_r$	Element numerical flux on the right boundary.
$solution\_time\_der$	Solution time derivatives.
G	Coefficient in 3rd-order Runge-Kutta scheme.
$ref\_x[2]$	Element reference boundary in x direction.
$ref\_y[2]$	Element reference boundary in y direction.
$next$	Pointer to the next element.
$hrefine$	h-refinement flag.
$prefine$	p-refinement flag.
$coarsen$	Coarsening flag.
$mortar$	Mortar elements.

Table 5.1: Symbols of terms of element storage

5.3 Comparison Between Hash Table and Tree Structure AMR

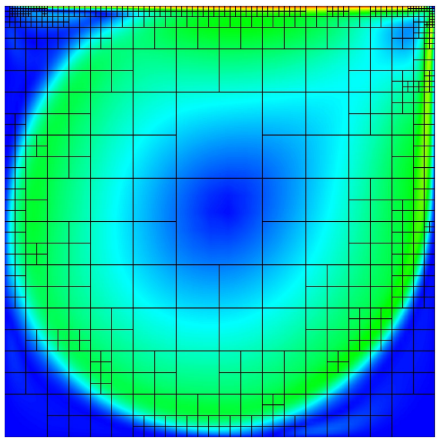
In this section, we compare the major differences between hash table AMR and tree-structured AMR. We summarize the conclusions in Table 5.2.

Features	Tree	Hash table
Mesh resolution flexibility	high	high
Insertion	$O(1)$	$O(1)$
Deletion	$O(1)$	$O(1)$
Neighbour query	$O(logN)$	$O(1)$
Pointer number/element	5	1
Adjacent element size constraints	2:1 balance	None

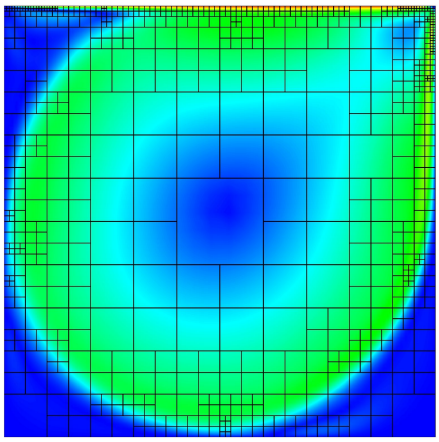
Table 5.2: Tree-structured AMR and hash table AMR features comparison.

The hash table and tree-structured AMR both have high flexibility in mesh resolution. Insertion and deletion operations are both very efficient ($O(1)$) for the two data structures. With known key values, the hash table supports constant time data search whereas for the tree structure, on average it takes $O(logN)$, where N is the number of nodes in the tree. In the hash table data structure, no actual different layers of mesh are kept unlike the tree structure. The hierarchy of the mesh can be indicated by the components of the element coordinate. Therefore, considerable memory can be saved in this implementation when comparing with using the tree structure.

Another preferable feature of the hash table data structure is that there is no refinement constraint in the sense of adjacent element sizes. Tree data structures used in AMR applications usually have a restriction on the relative sizes of adjacent elements, which is called the balance constraint [11, 72, 76]. Most of them use a 2:1 balance, which means the adjacent element edge length cannot exceed 2 times the adjacent element's length. Offermans *et al.* used the *p4est* library to handle the mesh refinement management in their application: a 2D lid-driven cavity [58]. Here, we showcase their refined mesh in Fig. 5.9 to illustrate 2:1 balanced meshes.



(a) Mesh after refinement snapshot one: non-conforming interfaces are imposed with 2:1 balance.

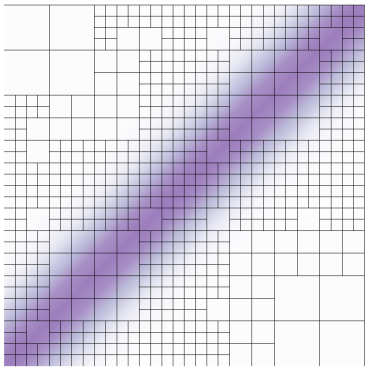


(b) Mesh after refinement snapshot two: non-conforming interfaces are imposed with 2:1 balance.

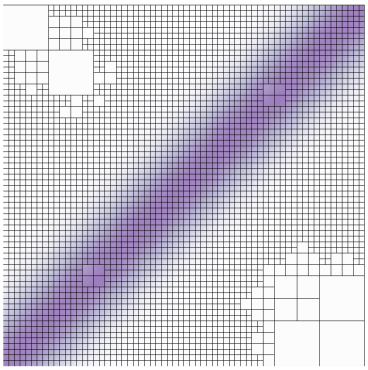
Figure 5.9: 2:1 balance restriction: the element edge length cannot not exceed 2 times its adjacent elements' length in the 2D lid-driven cavity meshes. (Reprinted from [58].)

To enforce this constraint, extra work needs to be done besides building up the tree. The balancing process needs to be imposed locally and between processors. The 2:1 balance constraint is relatively harder to enforce across processors than locally. One of the side effects of the 2:1 balance is that the subdivisions could in turn introduce new imbalance and so the process has to be repeated iteratively. This feature, that an element can affect elements that are not adjacent to each other is called the *ripple effect* [72]. To perform this balance across processors and also take care of the ripple effect, many algorithms have been developed [72, 77]. Many of them have sub-optimal isogranular scalability due to the iterative communication between processors. Moreover, the ripple effect could cause the application to refine at unnecessary locations.

In our proposed data structure, there is no geometrical constraint on element interfaces, so we spare ourselves the ripple effect, which makes this algorithm less problem-prone. An example from our wave equation is presented in Fig. 5.10: the coarse meshes at the upper left corner of the domain are facing elements of different refinement levels. The ability to deal with multi-level non-conforming interfaces enables us to have a better control of the mesh resolution and save storage against a 2:1 balanced mesh.



(a) Mesh after refinement snapshot one: the coarse meshes at the upper left corner of the domain have refinement level differences larger than 1 with some of their neighbours.



(b) Mesh after refinement snapshot two: the coarse meshes at the upper left corner of the domain have refinement level differences larger than 1 with some of their neighbours.

Figure 5.10: h-refinement without geometrical restriction: no geometrical restriction is imposed on the adjacent elements' non-conforming interfaces in the 2D wave problem meshes.

In summary, the proposed hash table AMR provides efficient data management operations. The simplicity of the hash table contributes to a memory-friendly AMR encoding scheme. With minimum constraints to the mesh resolution, this data structure is suitable and reliable for AMR applications.

Chapter 6

Interprocessor boundaries update methods

One of the big challenges of parallel AMR is updating the interprocessor boundaries after applying the refinements. We call the interprocessor boundaries *MPI boundaries* since we are working with *Message Passing Interface (MPI)* in a distributed memory environment.

The most intuitive way to update the MPI boundaries is to exchange the shared boundary information between adjacent processors. And this is the situation in our implementation: only the processors that have shared interfaces exchange their MPI boundary information. Some applications use a ghost layer to store the MPI boundary data [16]. We integrate the ghost layers into our element storage variable $facen$. This is practical since we are dealing with non-conforming element interfaces, which could be functionally non-conforming or geometrically non-conforming: elements on the MPI boundaries need to be aware of their neighbours' polynomial orders and h-refinement levels. Grouping all the essential data in a single vector for the solver to build mortars between the elements adds to the readability of the code and gives the user a better control of these variables.

Since there is no geometrical constraint on element non-conforming interfaces, *e.g.*, 2:1 balance, there is no further refinement to apply on MPI boundaries. Therefore, we avoid the ripple effect [72] previously described in Section 5.3. Additionally, to impose a 2:1 balance many applications perform parallel searches to balance MPI boundaries, which requires iterative communication. Parallel searches are proven to have limited scalability on a large number of processors [72]. In our algorithm, the adjacent processors only need to communicate once.

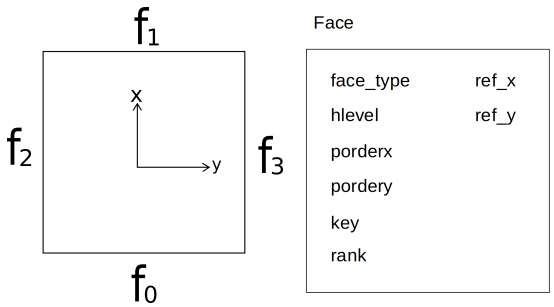
We first present the element interface storage in Section 6.1. An important structure we use in our boundary update method is an MPI boundary table, which is explained in Section 6.2. To obtain an efficient interprocessor communication pattern, MPI derived data types are invoked in Section 6.3. Finally, the algorithms related to the MPI boundary update process are detailed

in Section 6.4.

6.1 Element Interface Storage

In this section, we explain the necessary knowledge one element needs to have for the numerical computation.

We number the element faces in the sequence shown in Fig. 6.1a. The $facen$ variable is made up of 4 vectors of $Face$ containers (Fig. 6.1b), which have variables $face\_type$, $hlevel$, $porderx$, $pordery$, key, $rank$, ref_x, ref_y. Note that we flip the coordinate to the one shown in Fig. 6.1b and use the inverted coordinate in the following section. The inverted coordinate has no impact to the element coordinates we define as Equation 5.3.



(a) Definition of the element interfaces numbers.

(b) $Face$ structure storage.

Figure 6.1: Element interface storage configuration.

These variables are crucial to form the mortars between elements. As explained in Chapter 4, one needs the polynomial orders on the two sides of the mortar, one of which is the polynomial order of the neighbour element, provided by: $porderx$ and $pordery$ for x and y direction respectively, and the coordinate offsets and scaling factors from the two sides, which can be deduced from ref_x and ref_y. These two variables are deduced by comparing the element's root. The element is accessed by its key in the hash table, therefore we record the neighbour's key in the key variable. In order to discriminate local neighbours, remote neighbours and physical boundaries, we define a variable $face\_type$, which describes the character of the neighbour as "L", "M" and "B" for local neighbour, remote neighbour, and physical boundary, respectively. The neighbour's h-refinement level is recorded by $hlevel$. This variable will be useful when we update the MPI boundaries. The $rank$ variable is the rank number of the processor that the neighbour resides on (rank = processor number - 1). For elements on the physical boundary, the $rank$ variable is zero.

$facen$ is a container that holds four sequence containers that represent the four element interfaces that retain the $Face$ structure. Thus, for an element that has multiple neighbours, we can

append new *Face* structures at the end of the container.

6.2 MPI Boundary Table

In order to avoid element-wise communications among MPI boundaries, we pack the MPI boundary information that needs to be exchanged and send it to the neighbour processors so that we can minimize interprocessor communication times. All the packed information is stored in four containers *north*, *south*, *east*, *west* with respect to the interface direction (Fig. 6.2). These four containers are constructed by using the hash table data structure. The neighbour's rank (target rank) is the key to access the value they store. The elements facing the same target rank are grouped together and sent in one shot to the destination. The structure of the exchange information is defined as an *mpi_table* data type, shown in Fig. 6.3. We use the term *MPI tables* to generally refer to these four containers.

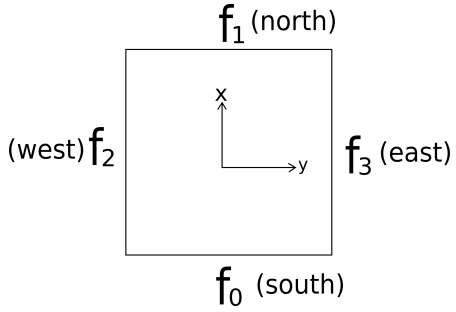


Figure 6.2: Four directions of the element faces: south, north, west and east.

```
struct{
    long long int local_key;
    int mpi_length;
    int owners_rank;
};
```

Figure 6.3: mpi_table data structure.

local_key is the key of the local element. To describe the element size, we define the variable *mpi_length*. One can map the element h-refinement level (a non-negative integer) to element length by

$$length = 2^{l_{max}-l}, \tag{6.1}$$

where l_{max} is the user-defined maximum h-refinement level, and l is the current element's h-refinement level. We then create Algorithm 6.1 *Elem_length* to convert element h-refinement

level into element length.

Algorithm 6.1 Elem_length: return the element boundary length.

Input: l, l_{max}

Output: $length$

1: $length = 2^{l_{max}-l}$

The last variable, *owners_rank*, is the future rank of the current element, which is useless in the current situation, but we shall use this in Chapter 7, when we repartition the grid.

6.2.1 MPI Table Example

Next, a simple example is used to explain how to build MPI tables.

In Fig. 6.4, the elements coloured in pink reside on Rank 0 and the green elements on Rank 1. The annotations starting with "k" are the elements' keys generated by our pairing function. The element coordinates are in the brackets. Here, we assume the maximum h-refinement level is 2 (h-refinement level starts with zero). The MPI boundary between Rank 0 and Rank 1 is indicated by the red dashed line.

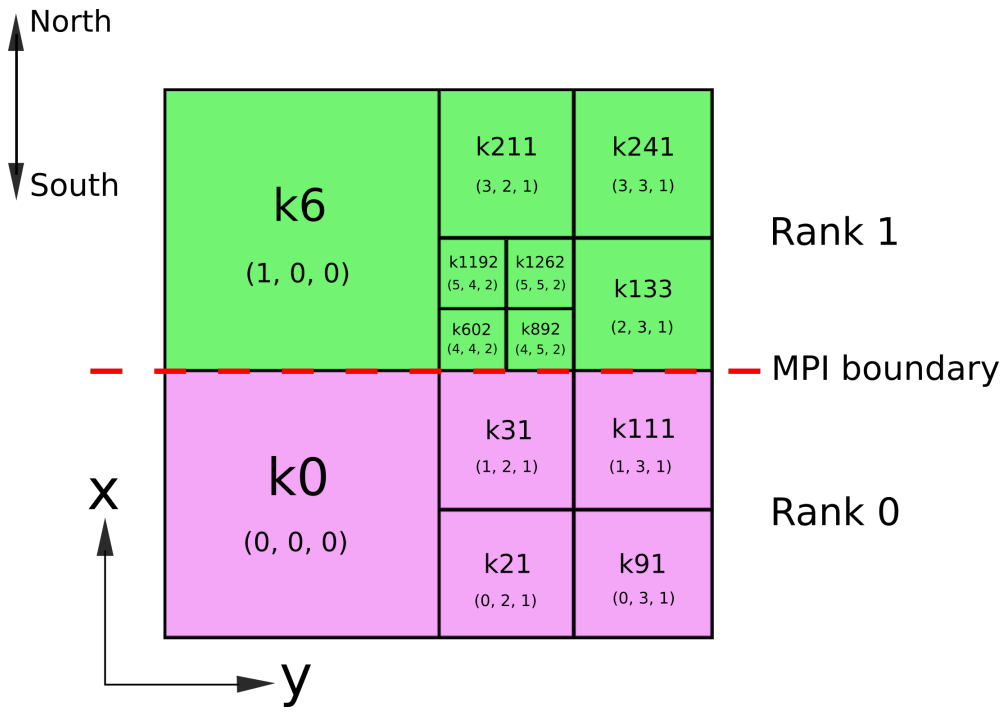


Figure 6.4: MPI table construction example: mesh distribution. The MPI boundary between Rank 0 and Rank 1 is denoted by the red dashed line. The elements coloured in pink reside on Rank 0 and the green elements on Rank 1. The element keys begin with "k", and the element coordinates are in the brackets.

We take the $k111$ element, whose coordinate is $(1, 3, 1)$ in Fig. 6.4, as an instance of the computation of the key, using the Szudzik Pairing Function (Equation 5.7). In this case, we have:

$$key = \pi(k_1, k_2, k_3) = \pi\left(\pi\left(k1, k2\right), k_3\right), \tag{6.2}$$

$$k_1 = 1, k_2 = 3, k_3 = 1. \tag{6.3}$$

Based on the criteria in Equation 5.7, we obtain:

$$key' = \pi(k_1, k_2) = \pi(1, 3) = 1 + 3^2 = 10, \tag{6.4}$$

$$key = \pi(key', k_3) = \pi(10, 1) = 10^2 + 10 + 1 = 111. \tag{6.5}$$

The calculation of other element keys follows the same logic.

Two MPI tables will be formed independently on Rank 0 and Rank 1. Rank 0 and Rank 1 share an MPI boundary across the x direction indicated in Fig. 6.4. Therefore, Rank 0 builds a *north* table and Rank 1 builds a *south* table. The contents of Rank 0's *north* table are given in Table 6.1.

	Rank 0: north table		
local_key	k0	k31	k111
mpi_length	4	2	2
owners_rank	0	0	0

Table 6.1: MPI table construction example: MPI north table of Rank 0.

The MPI table uses the hash table data structure with the neighbour's rank as the key. In this case, both of the processors only have one item in their MPI table since they only share MPI boundaries with each other. For Rank 0, the MPI boundary shared with Rank 1 is composed of three elements: $k0$, $k31$, and $k111$. Their sequence in the table does not affect the consistency of the algorithm. The element's *mpi_length*s are computed by using Equation 6.1. For now, the *owners_rank* remains zero.

Following the same logic, we form the *south* table for Rank 1 in Table 6.2.

	Rank 1: south table			
local_key	k6	k602	k892	k133
mpi_length	4	1	1	2
owners_rank	0	0	0	0

Table 6.2: MPI table construction example: MPI south table of Rank 1.

The idea of MPI tables is to put all the elements that are facing the same rank together. One table could have multiple items that are corresponding to different neighbour processors.

The algorithms to construct MPI tables are presented in the next section (Section 6.2.2). After two complete MPI tables are formed, the next step is to exchange boundary information between Rank 0 and Rank 1. Before we invoke interprocessor communications, we introduce *MPI-derived data types* in Section 6.3. The MPI-derived data types allow us to do point-to-point communication with a mixture of different basic data types, which contributes to a more convenient and more efficient communication pattern.

6.2.2 MPI Table Implementation

In this section, we give the algorithms to form the MPI tables. Besides forming the MPI table, we also deduce the possible neighbours of each element on the MPI boundaries. This is a crucial step for our MPI boundary update algorithms. Since every element's key can be calculated by using the element integer coordinates, for an element who knows its own coordinates, it can easily enumerate all the possible keys of its neighbours. Therefore, one element could search among the keys it received and match the element on the MPI boundaries with its corresponding neighbours. Since each element has a unique key, there is zero chance to mismatch the neighbour-pairs. Since the update algorithm is a "search-and-match" procedure, the sequence of the received keys does not matter. That is why we mentioned that the element sequence inside the MPI table does not matter. This feature contributes to the implementation of our *space-filling curve based* repartitioning algorithm explained in Chapter 7.

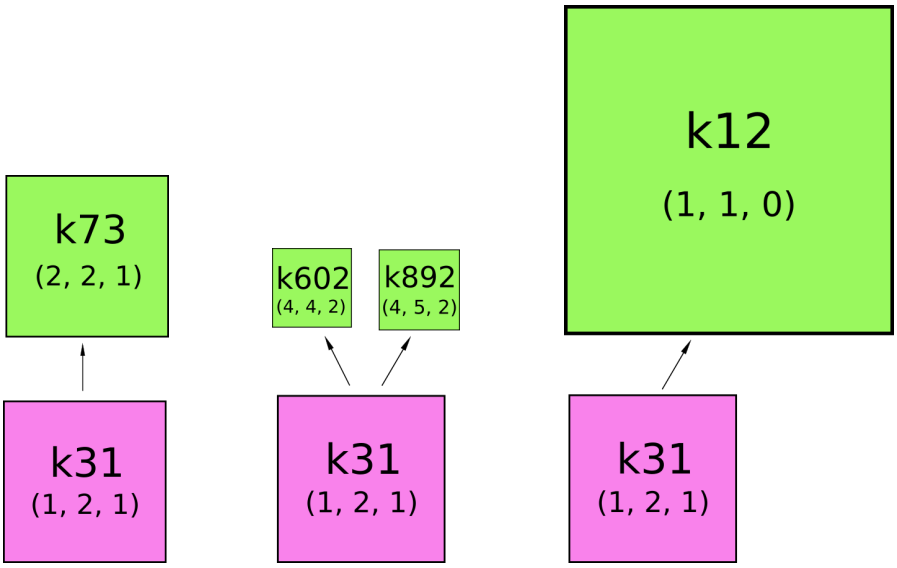
We follow the example in Section 6.2.1 to further illustrate the MPI table construction process. The general algorithms are given later. Suppose we are on Rank 0, to form the MPI table in the x direction, we follow the following process.

1. Put the elements on the MPI boundary into the MPI north table (Algorithm 6.2). Then we have Table 6.3 shown below.

	Rank 0: North Table		
Local Key	k0	k31	k11

Table 6.3: MPI table construction example: record the keys of the elements on the MPI boundary.

2. Deduce the possible neighbours' keys of each element in the north table (Algorithm 6.3). We take element $k31$ as an example: three possible interface situations are shown in Fig. 6.5.



(a) Possibility 1: $k31$ is facing a h-refinement level 1 element.

(b) Possibility 2: $k31$ is facing two h-refinement level 2 element.

(c) Possibility 3: $k31$ is facing a h-refinement level 0 element.

Figure 6.5: MPI table construction example: element $k31$'s north interface possibilities.

In this case, since element $k31$'s h-refinement level is 1 and the maximum refinement level is 2, it could have three different interface situations: (a) it is facing a h-refinement level 1 element, (b) it is facing two h-refinement level 2 elements, or (c) it is facing a h-refinement level 1 element. With the integer coordinate of $k31$ ($(1, 2, 1)$), the neighbours' coordinates corresponding to each possibility can be easily derived. Then, using the Szudzik pairing function, we obtain the keys of the potential neighbours. The possible neighbours are stored for later use as shown in Table 6.4.

	Rank 0: North Table		
Local Key	k0	k31	k11
Possible Neighbours	⋮	k6, k602, k892, k133	⋮

Table 6.4: MPI table construction example: deduce the possible neighbours of the elements on the MPI boundary.

3. Record the MPI boundary length of each element in the north table.

   In this step we record the element interface length that is exposed to the MPI bound-

ary. The exposed length can be calculated based on the current neighbour information (explained in Algorithm 6.8). Then we have our north table shown in Table 6.5.

	Rank 0: North Table		
Local Key	k0	k31	k11
Possible Neighbours	⋮	k6, k602, k892, k133	⋮
MPI length	4	2	2

Table 6.5: MPI table construction example: calculate the element length that is exposed to the MPI boundary.

4. Find out the new neighbours: a search-and-match process.

   Although the MPI interfaces update methods are given in Section 6.4, we choose to disclose the fundamental idea here for the better understanding of the usage of the MPI tables.

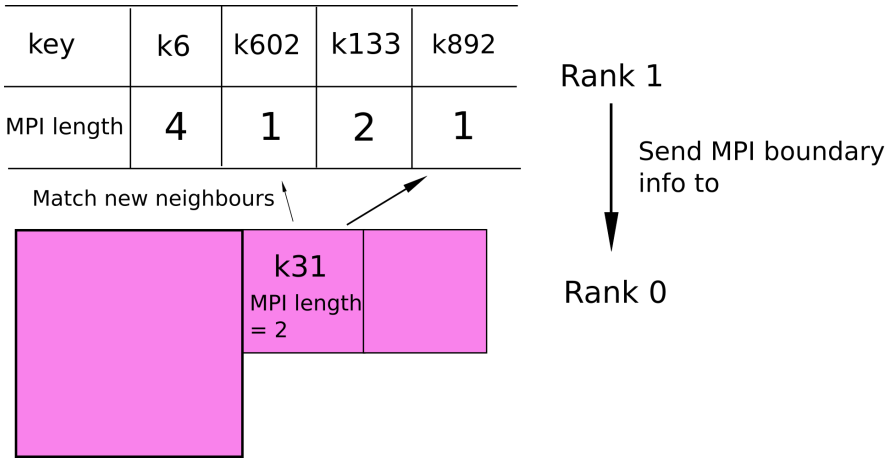


Figure 6.6: MPI table construction example: element $k31$ finds its new neighbours.

In Fig. 6.6, Rank 1 sends its MPI boundary information to Rank 0, then elements in the north table will find their neighbours by comparing and matching the received keys and the deduced keys they computed in the former step. Since the key values are unique, there is no chance to mismatch the neighbours. Keys 602 and 892 are inside $k31$'s possible neighbour list, and the sum of their MPI length equals to the MPI length of $k31$, therefore, we successfully update the new MPI interfaces. The MPI length can serve as a stopping criterion in the search-and-match process.

Now, we provide the general algorithms. First, we present the algorithm to form the MPI table as Algorithm 6.2 *Put_in_mpi_table*. The temporary pointer to the current element is stored in

temp. The iterator[1] $facen\_it$ points to the element *facen* variable of the corresponding direction. The new information is inserted to the MPI table: *table*. The algorithm only stores the key of the current element under the corresponding target rank.

Algorithm 6.2 Put_in_mpi_table: insert current MPI boundary element to the MPI table.

Input: *temp*, $facen\_it$, *table*

Output: *table*

Use Algorithms: Algorithm 5.2 Get_key_fun

1: **if** *table* does not have rank: $(facen\_it \rightarrow rank)$ **then** ▷ If this target rank is not recorded.
2: allocate $table[facen\_it \rightarrow rank]$
3: $local\_key = Get\_key\_fun(temp \rightarrow index[0], temp \rightarrow index[1], temp \rightarrow index[2])$
4: $table \leftarrow \{local\_key, 0, 0\}$ ▷ Put this element into the table, leave *mpi_length* and *owners_rank* as zero.
5: **else** ▷ Table has recorded this rank.
6: $local\_key = Get\_key\_fun(temp \rightarrow index[0], temp \rightarrow index[1], temp \rightarrow index[2])$
7: **if** not found *local_key* under this rank **then** ▷ Avoid duplication.
8: $table \leftarrow \{local\_key, 0, 0\}$
9: **end if**
10: **end if**

Once we put the current boundary element into the MPI table, we need to deduce its possible neighbours and store the possible neighbours vector in *neighbours* variable (see Algorithm 6.3). Neighbour deduction is achieved in Algorithm 6.4 $Neighbour\_array\_x$ and Algorithm 6.5 $Neighbour\_array\_y$, for the x and the y directions respectively. The possible neighbours vector is formed in h-refinement level descending order. Therefore, when processors are matching neighbour pairs the algorithm will start from the most refined possible neighbours.

Algorithm 6.3 *Possible_neighbours* takes the pointer *temp* of the current element and the interface number *facen* to generate a vector of the keys of possible neighbours *neighbours*. The algorithm first checks the current element has not been detected before, to avoid duplicated storage. Then different functions are called corresponding to element face number.

[1] An iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (∗) operators).

Algorithm 6.3 Possible_neighbours: form the possible neighbour array based on the face number.

Input: $temp$, $facen$

Output: $neighbours$

Use Algorithms: **Algorithm 6.4 Neighbour_array_x**
Algorithm 6.5 Neighbour_array_y

1: $local\_key = Get\_key\_fun(temp \rightarrow index[0], temp \rightarrow index[1], temp \rightarrow index[2])$
2: **if** $neighbours$ has record $local\_key$ **then**
3: return
4: **end if**
5: $i = temp \rightarrow index[0]$
6: $j = temp \rightarrow index[1]$
7: $k = temp \rightarrow index[2]$
8: **if** $facen == 0$ **then**
9: $Neighbours\_array\_x(i-1, j, k, local\_key, facen, neighbours)$
10: **else if** $facen == 1$ **then**
11: $Neighbours\_array\_x(i+1, j, k, local\_key, facen, neighbours)$
12: **else if** $facen == 2$ **then**
13: $Neighbours\_array\_y(i, j-1, k, local\_key, facen, neighbours)$
14: **else if** $facen == 3$ **then**
15: $Neighbours\_array\_y(i, j+1, k, local\_key, facen, neighbours)$
16: **end if**

We present Algorithm 6.4 $Neighbour\_array\_x$ to produce the possible neighbour's key in the x direction. i, j, k are the coordinates of the current element's neighbour that shares the same h-refinement level with the current element. $local\_key$ is the key value of the current element. $facenum$ is the element face number. With the h-refinement level k, using function $Total\_neighbours$ (Algorithm 6.6), one can obtain the size of the total possible neighbours list. This function also offers us the index of the same size neighbour $same\_size$. Next, we use the neighbour array's size $total\_n$ to allocate the space for the neighbour array, then put the same size neighbour in the right position in the array. In the remaining section of the code (lines $12 \sim 34$), the possible neighbour's keys are generated iteratively. The strategy applied to the y direction is similar to that in the x direction.

Algorithm 6.4 Neighbour_array_x: form possible neighbour array in h-refinement level descending order (x direction).

Input: i, j, k, $local\_key$, $facenum$

Output: $neighbours$

Use Algorithms: Algorithm 6.6 Total_neighbours

1: $x = 0$
2: **if** $facenum == 0$ **then**
3: $\quad x = 1$
4: **end if**
5: $total\_n = 0$ ▷ The size of the neighbour element list.
6: $same\_size = 0$ ▷ The index of the neighbour who has the same size as the current element in the neighbour list.
7: $Total\_neighbours(k, total\_n, same\_size)$
8: allocate $neighbours[local\_key] \leftarrow total\_n$
9: $neighbours[local\_key][same\_size] = Get\_key\_fun(i, j, k)$ ▷ First record the same size neighbour.
10: $pre\_level[3] \leftarrow \{i, j, k\}$
11: $elem = same\_size - 1$
12: **for** $m = k + 1 \rightarrow hlevel\_max$ **do**
13: $\quad layer\_elem = 2^{m-k}$
14: $\quad pre\_level[0] = 2pre\_level[0] + x$
15: $\quad pre\_level[1] = 2pre\_level[1]$
16: $\quad pre\_level[2] = pre\_level[2] + 1$
17: $\quad$ **for** $n = layer\_elem - 1 \rightarrow 0$ **do**
18: $\quad\quad key = Get\_key\_fun(pre\_level[0],\ pre\_level[1] + n,\ pre\_level[2])$
19: $\quad\quad neighbours[local\_key][elem] = key$
20: $\quad\quad elem = elem - 1$
21: $\quad$ **end for**
22: **end for**
23: $pre\_level[0] = i/2$
24: $pre\_level[1] = j/2$
25: $pre\_level[2] = k - 1$
26: $elem = size\_size + 1$
27: **for** $m = k - 1 \rightarrow 0$ **do**
28: $\quad key = Get\_key\_fun(pre\_level[0],\ pre\_level[1],\ pre\_level[2])$
29: $\quad neighbours[local\_key][elem] = key$
30: $\quad elem = elem + 1$
31: $\quad pre\_level[0] = pre\_level[0]/2$
32: $\quad pre\_level[1] = pre\_level[1]/2$
33: $\quad pre\_level[2] = pre\_level[2] - 1$
34: **end for**

The strategy applied to the y direction is similar to the one explained in the x direction.

Algorithm 6.5 Neighbour_array_y: form possible neighbour array in h-refinement level descending order (y direction).

Input: i, j, k, $local\_key$, $facenum$

Output: $neighbours$

Use Algorithms: Algorithm 6.6 Total_neighbours

1: $y = 0$
2: **if** $facenum == 2$ **then**
3: $\quad y = 1$
4: **end if**
5: $total\_n = 0$
6: $same\_size = 0$
7: $Total\_neighbours(k, total\_n, same\_size)$
8: allocate $neighbours[local\_key] \leftarrow total\_n$
9: $neighbours[local\_key][same\_size] = Get\_key\_fun(i, j, k)$ ▷ First record the same size neighbour.
10: $pre\_level[3] \leftarrow \{i, j, k\}$
11: $elem = same\_size - 1$
12: **for** $m = k + 1 \rightarrow hlevel\_max$ **do**
13: $\quad layer\_elem = 2^{m-k}$
14: $\quad pre\_level[0] = 2pre\_level[0]$
15: $\quad pre\_level[1] = 2pre\_level[1] + y$
16: $\quad pre\_level[2] = pre\_level[2] + 1$
17: $\quad$ **for** $n = layer\_elem - 1 \rightarrow 0$ **do**
18: $\quad\quad key = Get\_key\_fun(pre\_level[0],\ pre\_level[1] + n,\ pre\_level[2])$
19: $\quad\quad neighbours[local\_key][elem] = key$
20: $\quad\quad elem = elem - 1$
21: $\quad$ **end for**
22: **end for**
23: $pre\_level[0] = i/2$
24: $pre\_level[1] = j/2$
25: $pre\_level[2] = k - 1$
26: $elem = size\_size + 1$

27: **for** $m = k - 1 \rightarrow 0$ **do**
28: $key = Get\_key\_fun(pre\_level[0], pre\_level[1], pre\_level[2])$
29: $neighbours[local\_key][elem] = key$
30: $elem = elem + 1$
31: $pre\_level[0] = pre\_level[0]/2$
32: $pre\_level[1] = pre\_level[1]/2$
33: $pre\_level[2] = pre\_level[2] - 1$
34: **end for**

Algorithm 6.6 $Total\_neighbours$ is the routine to calculate the size of the possible neighbour array. $level\_now$ is the current element's h-refinement level. The algorithm generates the outputs of the size of the neighbour array all and the index of the same size neighbour $my\_position$.

Algorithm 6.6 Total_neighbours: compute the total number of possible neighbours.

Input: $level\_now$

Output: all, $my\_position$

1: **for** $i = hlevel\_max \rightarrow 0$ **do**
2: **if** $i > level\_now$ **then**
3: **else if** $i == level\_now$ **then**
4: $all = all + 1$
5: $my\_position = all - 1$
6: **else**
7: $all = all + 1$
8: **end if**
9: **end for**

Algorithm 6.7 $Construct\_mpi\_table$ forms the complete process to build an MPI table. Looping the temporary pointer through the linked list, the elements having *face_type* as "M" (facing MPI boundary) will be put in an MPI table. This algorithm guarantees that under a specific rank, there would be no element storage duplication. For each direction (x or y), two faces (*face1* and *face2*) will be examined. Besides recording the elements' keys, we also record the length of their boundaries that are exposed to MPI boundaries. We present Algorithm 6.8 $Record\_length$ to accomplish the length recording. After the execution of Algorithm 6.7, we obtain the MPI tables corresponding to the their *face*s and arrays of possible neighbours of each element inside the MPI table.

Algorithm 6.7 Construct_mpi_table: construct MPI table in one direction.

Input: $local\_elem\_num$, $face1$, $face2$

Output: $table1$, $table2$, $neighbours1$, $neighbours2$

Use Algorithms: **Algorithm 6.2 Put_in_mpi_table**
Algorithm 6.3 Possible_neighbours
Algorithm 6.8 Record_length

1: $temp = head$ ▷ temporary pointer points to the head of the linked list.
2: **for** $k = 0 \rightarrow local\_elem\_num$ **do**
3: **for** $it = temp \rightarrow facen[face1].begin \rightarrow temp \rightarrow facen[face1].end$ **do**
4: **if** $it \rightarrow face\_type == M$ **then**
5: $Put\_in\_mpi\_table(temp, it, table1)$ ▷ Put this element in MPI table.
6: $Possible\_neighbours(temp, neighbours1, face1)$ ▷ generates its possible neighbour's keys
7: $Record\_length(temp \rightarrow hlevel, it \rightarrow index[2], it \rightarrow rank, table1)$ ▷ Record the length that exposes to the MPI boundary.
8: **end if**
9: **end for**
10: **for** $it = temp \rightarrow facen[face2].begin \rightarrow temp \rightarrow facen[face2].end$ **do**
11: **if** $it \rightarrow face\_type == M$ **then**
12: $Put\_in\_mpi\_table(temp, it, table2)$
13: $Possible\_neighbours(temp, neighbours2, face2)$
14: $Record\_length(temp \rightarrow hlevel, it \rightarrow index[2], it \rightarrow rank, table2)$
15: **end if**
16: **end for**
17: $temp = temp \rightarrow next$ ▷ Pointer moves to the next element.
18: **end for**

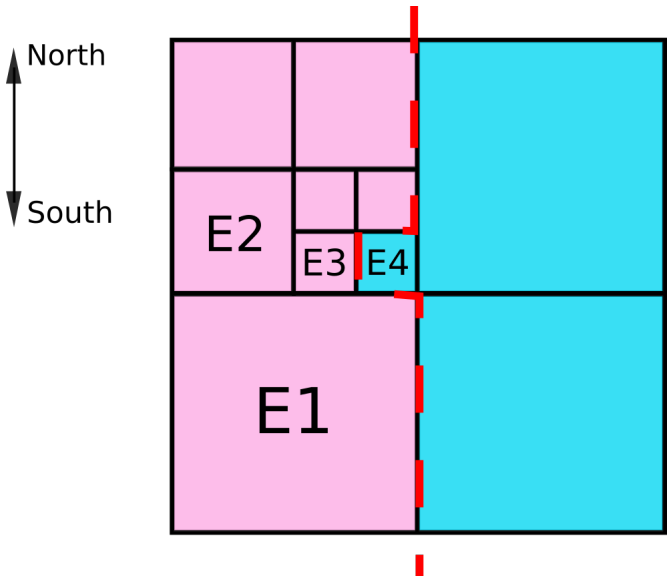


Figure 6.7: Relationship between element boundary length and element length exposed to the MPI boundary. MPI boundary is indicated by the red dashed line. Processors are discriminated by the colour scheme. Face 1 (north face) of element $E1$ is partially exposed to the MPI boundary, therefore, element boundary length $\leq$ element length on the MPI boundary.

Algorithm 6.8 *Record_length* calculates the element length that is facing the MPI boundary. Note that not all elements expose their complete boundaries to MPI boundaries. Here we use a small example depicted in Fig. 6.7 to demonstrate. The pink elements belong to rank 0 and the blue elements rank 1. Element $E1$ is on the MPI boundary since its boundary is shared with element $E4$. Suppose the maximum h-refinement level is 2 and $E1$'s h-refinement level is 0. Although $E1$'s length is $2^{2-0} = 4$, its face 1 only exposes length 1 to the MPI boundary, which is the full length of $E4$. Therefore, the element's length on the MPI boundary is less than or equal to its full boundary length, *i.e.*, $mpi\_length \leq 2^{l\_max-l}$. Therefore, in Algorithm 6.8, we need to compare the h-refinement level between the current element, *my_level*, and its neighbour's level, *n_hlevel*. We record the smaller element's length.

Algorithm 6.8 Record_length: Record the element length that is exposed to a certain neighbour processor.

Input: *my_level*, *n_hlevel*, *target_rank*

Output: *my_table*

Use Algorithms: Algorithm 6.1 Elem_length

1: **if** $n\_level \leq my\_level$ **then**
2: $\quad my\_table[target\_rank].back.mpi\_length = Elem\_length(my\_hlevel)$
3: **else**
4: $\quad my\_table[target\_rank].back.mpi\_length \mathrel{+}= Elem\_length(n\_hlevel)$
5: **end if**

Up to now we are able to build the MPI table for the four directions: north, south, east and west. Each processor forms four MPI tables after each refinement, and we exchange the MPI boundary elements' information. One of the advantages provided by the MPI table is that we restrict the communication to processors with shared boundaries, therefore, global communication is avoided.

Before we introduce the MPI boundaries update method, we first create our *MPI derived data types* to support the interprocessor communications.

6.3 MPI Derived Data Types

The point-to-point communication in MPI is based on using contiguous buffers with a sequence of data of the same type. Therefore, this rule limits the formats of the message that can be exchanged. However, in an application, one might need to pass messages with different data types, *e.g.*, an integer follows two double precision float numbers. Or sometimes, one needs to send non-contiguous data, *e.g.*, a column of a matrix. A possible solution is to send one type of data at one time, or pack the non-contiguous data into a contiguous buffer. Although this is doable, it is not efficient since it increases the communication time or invokes non-necessary copy operators.

Luckily, MPI offers a way to customize one's data type, which means one can define a data type with mixed basic data types and use a non-contiguous buffer. This feature not only increases the efficiency of the message exchange process, but also improves the program's readability and portability.

6.3.1 Implementation

Since the MPI derived data type is a very versatile option, one has numerous ways to customize the desired data type, in this work we only focus on the MPI derived data that serves our application.

We pack the data we need to send based on the corresponding MPI table. We define the derived MPI data type as *facen_pack*. The customized data type is defined in Listing 6.1.

```
struct facen_pack{

    long long int local_key;          // local element's
        key

    int hlevel;                  // h-refinement level

    int porderx;      // polynomial order in x direction
```

```
    int pordery;     // polynomial order in x direction

    int mpi_length; // element length

    double ref_x[2];          // reference space in x
        direction
    double ref_y[2];          // reference space in y
        direction

};
```

Listing 6.1: MPI derived data type: *facen_pack*

As shown above, the derived data type is a mixture of basic data types. To define a general MPI derived data type, one needs to specify two things: a sequence of basic data types and a sequence of integer displacements (in bytes). There are two ways to define the data displacement in MPI. The first one is to use *MPI_Type_get_extent*, and the other is to use *MPI_Get_address*. In the following section, we explain the reason why the latter function is a safer option.

Using Extents

The most intuitive way to define data sequence displacement is to get the data type's extent in bytes and add up the former data type's extent. MPI offers the built-in function *MPI_Type_get_extent* to return the lower bound and the extent of a data type.

The customized data type can be created within three steps:

1. Define the data sequence displacements and data sequence types.
2. Create new data types by using appropriate MPI routines.
3. Commit the new data type.

Following this logic we create our derived *Facen_pack* type as follows in Listing 6.2.

```
// there are three basic data types in the struct
int num = 3;

// Number of elements in each block (array of integers)
int elem_blocklength[num]{1, 4, 4};

// Byte displacement of each block (array of integers).
MPI_Aint array_of_offsets[num];
```

```
// integer extent, long long int extent, double extent
MPI_Aint intex, longintex, doubleex;
MPI_Aint lb;     // lower bound extent

// get extent in bytes
MPI_Type_get_extent(MPI_INT, &lb, &intex);
MPI_Type_get_extent(MPI_LONG_LONG_INT, &lb, &longintex);
MPI_Type_get_extent(MPI_DOUBLE, &lb, &charex);

array_of_offsets[0] = (MPI_Aint) 0;
array_of_offsets[1] = array_of_offsets[0] + longintex;
array_of_offsets[2] = array_of_offsets[1] + intex * 4;

MPI_Datatype array_of_types[num]{MPI_LONG_LONG_INT, MPI_INT,
    MPI_DOUBLE};

// create and MPI datatype
MPI_Type_create_struct(num, elem_blocklength,
   array_of_offsets, array_of_types,
Facen_type);

// commit data type
MPI_Type_commit(&Facen_type);
```

Listing 6.2: Create derived data type by using data extension

However, this is not a safe or portable way to define new data types, especially if the new data type is based on the *struct* in C/C++. In C/C++ *struct*s might be padded by the compiler. This is called *data structure alignment* in computer memory management. In order to read or write to memory more efficiently, the CPUs of modern computers perform *natural alignment*, *i.e.*, the data's memory address is a multiple of the data size. Therefore, to ensure natural alignment, the compiler sometimes inserts some padding between structure elements or after the last element of the structure. For instance, on a 32 bit machine, the processing word size is four bytes.

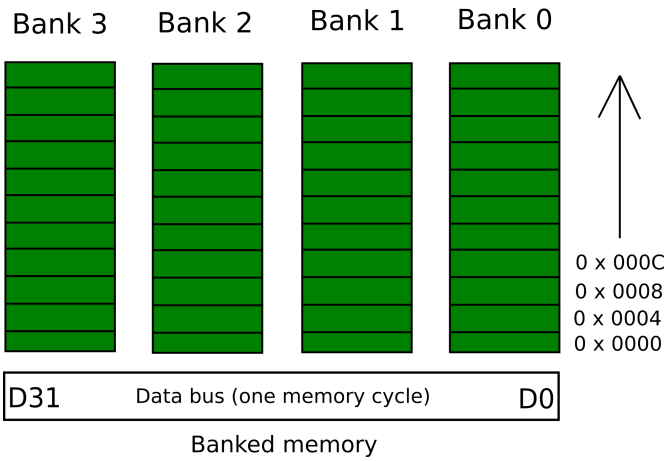


Figure 6.8: 32 bit machine memory bus. A single bank is one byte width. A machine with 4 banks needs one read cycle to fetch a 4 byte integer.

If the memory is arranged as single bank of one byte width, then a machine with four banks in a row only needs one read cycle to fetch a 4 byte integer (Fig. 6.8). To increase data access efficiency, an integer will be addressed at the starting rank of the memory bus, so that it only occupies a whole row. Even before this integer stores a data type that uses less than 4 banks, the CPU will put this integer in a new row and leave the former unused ranks vacant. This is the so-called *data structure padding*. Fig. 6.9 shows a small data structure padding example.

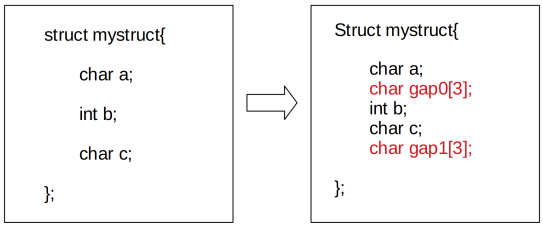


Figure 6.9: Data structure padding example. A character occupies 1 byte of memory and an integer occupies 4 bytes of memory. In order to place integer b on a whole row of memory bus, 3 bytes are kept vacant after character a. The same logic is applied to the padding after character c.

Although the first variable character a only takes a single byte of memory, in order to maintain natural alignment, after a there is a padding of 3 bytes, so that integer b can be put on the next row on the memory bus. The same logic applies to the padding behind character c. As we can see in this example, a C/C++ *struct* could occupy more memory than the actual data size requires. Therefore, to set the data displacement based on the data extent is not safe since it could ignore the data padding applied by the compiler.

Instead of using data extents to handle *struct*s, we suggest to use $MPI\_Get\_address$ to get the

addresses of each object in the data structure. In this way, the real relative data displacements are determined.

Using Addresses

Another way to set up the data displacements is to get the addresses of each data type, and subtract them to get the correct relative displacements. By doing this, we include the data padding when subtracting the addresses. *MPI_Get_address* returns the data's memory address. Using this method to define structure data type in MPI is less problem-prone and it increases the portability of the program. Our *Facen_pack* data type then can be defined as follows in Listing 6.3.

```
// There are three basic data type in the struct.
int num = 3;

int elem_blocklength[num]{1, 4, 4};

// Data type array
MPI_Datatype array_of_types[num]{MPI_LONG_LONG_INT, MPI_INT,
    MPI_DOUBLE};

// Array of offsets.
MPI_Aint array_of_offsets[num];
MPI_Aint baseadd, add1, add2;

// Create a vector using the target data structure.
std::vector<facen_pack> myface(1);

// Obtain data addresses.
MPI_Get_address(&(myface[0].local_key), &baseadd);
MPI_Get_address(&(myface[0].hlevel), &add1);
MPI_Get_address(&(myface[0].ref_x[0]), &add2);

// Obtain the relative offsets by subtracting the actual
   data addresses.
array_of_offsets[0] = 0;
array_of_offsets[1] = add1 - baseadd;
array_of_offsets[2] = add2 - baseadd;

// Create the user-defined data type
```

```
MPI_Type_create_struct(num, elem_blocklength,
    array_of_offsets, array_of_types, &Facen_type);

// check that the extent is correct
MPI_Aint lb, extent;
MPI_Type_get_extent(Hash::Facen_type, &lb, &extent);
if(extent != sizeof(myface[0])){

        MPI_Datatype old = Facen_type;
        MPI_Type_create_resized(old, 0, sizeof(myface[0]), &
            Facen_type);
        MPI_Type_free(&old);
}

// Commit the new data type.
MPI_Type_commit(&Facen_type);
```

Listing 6.3: Create derived data type by using data address

The memory displacements are set up by subtracting the data memory location. At the bottom of the code, we add a few functions to check that the extension of the structure is correctly defined. This part is used to make sure the padding at the end of the data structure is also being considered.

At this point, we have successfully defined our customized data structure in MPI, so we can send the message more efficiently. Note that one can derive all the necessary data types at the beginning of the program and use *MPI_Type_free* to free the data types at the end of the program.

6.4 Update Interprocessor Interfaces

Now we are ready to exchange the MPI boundaries information. Each processor packs all the element information based on the MPI table and sends it to the neighbour processors. We exchange the information in x and y directions separately. The MPI boundary update algorithms are identical in both directions.

Following the example in Section 6.2.1, rank 0 and rank 1 need to exchange MPI boundary information across the x direction. Each processor loops through the MPI table and packs the necessary information (variables in structure *facen_pack*) of the corresponding face. We use a non-blocking communication scheme to exchange MPI boundary information.

Since MPI point-to-point communication not only needs the contribution of the sender, but also

requires the participation of the receiver, the two processors have to make a match read, *i.e.*, the receiving side needs to call MPI_Recv to accomplish the operation. Luckily, the receiver can call MPI_Recv based on the local MPI table. The point-to-point communication of the MPI boundary information update is presented in Algorithm 6.9 $Sender\_recver$.

In Algorithm 6.9 $Sender\_recver$, each processor first sends out its message then receives messages from its neighbours. Note here *sender_table* and *recver_table* are tables coming from opposite directions. For example, passing information in the x direction, if *sender_table* is *north* table, then *recver_table* must be *south* table. Then in the next round we interchange the two tables. To avoid *deadlock* in MPI communication, we use *non-blocking communication* operations. To keep the sender's buffer intact before we ensure the message reaches the receiver, we define the buffer as a hash table with the *target_rank* as the key to the values. When packing the MPI boundary element's information, we allocate the buffer *send_info* by using the MPI-derived date type $Facen\_type$, so that we can send the message with a mixture of data types. To ensure the message is safely received by the destination, $MPI\_Waitall$ must be called.

After the sending phase is completed, processors start to call MPI_Recv to receive the information. The challenge here is that, although receivers know from whom to receive, they do not know the message size, which is a mandatory variable in MPI_Recv routine. Therefore, we implement a dynamic receiving scheme with the help of MPI_Probe. Instead of creating a huge buffer to handle the receiving message, one can use MPI_Probe to query the message size before actually receiving it. After MPI_Probe returns, the size of the message can be extracted by MPI_Get_count. Lastly, we can allocate the receive buffer and get the message.

Algorithm 6.9 Sender_recver: processors communicate to update MPI boundaries.

Input: : *sender_table*, *recver_table*, *update_dir*, *neighbours_recv*

Use Algorithms: Algorithm 6.10 Update_hash

1: $s = sender\_table.size$ ▷ s is the size of the *sender_table*.
2: $r = recver\_table.size$ ▷ r is the size of the *recver_table*.
3: hash map: $send\_info < int, vector < facen\_pack >>$ ▷ Buffer for the sending out information. Use a hash table data structure. The key is the target rank number, the value is a vector using $facen\_pack$ structure.
4: $isend = 0$ ▷ Count the send operation number.
5: **if** $s > 0$ **then** ▷ If there is something to send.
6: **for** $v \in sender\_table$ **do**
7: $target\_rank = v.first$ ▷ Extract the key of the (key, value) pair in hash table.
8: $num\_elem = v.second.size$ ▷ Get the number of element that is facing this rank.

```
9:        it = v.second.begin                          ▷ We will use iterator it to loop through
                                                          the second part of the pair.
10:       allocate send_info[target_rank] = vector < facen_pack > [num_elem]  ▷ Ready to
                                  pack up sending info. Using the MPI derived data type: Facen_type.
11:       for k = 0 → num_elem do
12:           send_info[target_rank][k].local_key = it → local_key
13:           send_info[target_rank][k].hlevel = local_hash[it → local_key] → index[2]      ▷
                                                  Get the essential information inside the element hash table.
14:           send_info[target_rank][k].porderx = local_hash[it → local_key] → n
15:           send_info[target_rank][k].pordery = local_hash[it → local_key] → m
16:           send_info[target_rank][k].mpi_length = it → mpi_length
17:           send_info[target_rank][k].ref_x = local_hash[it → local_key] → ref_x
18:           send_info[target_rank][k].ref_y = local_hash[it → local_key] → ref_y
19:       end for
20:       MPI_Isend(buffer : send_info, count : num_elem, MPI_Data_type : Facen_type,
              destination : target_rank, tag : my_rank)
21:       ++isend
22:   end for
23: end if
24: if r > 0 then
25:   for v ∈ recver_table do
26:       num = 0
27:       MPI_Probe(source : v.first,  tag : v.first)
28:       MPI_Get_count(MPI_Datatype : Facen_type, count : num)
29:       allocate recv_info(num)
30:       MPI_Recv(buffer : recv_info, cout : num, MPI_Datatype : Facen_type,
                source : v.first, tag : v.first)
31:       Update_hash()
32:   end for
33: end if
34: if s > 0 then
35:   MPI_Waitall()
36: end if
```

After receiving the message, we need to update the element $facen$ variable. This algorithm is presented as Algorithm 6.10 *Update_hash*. To update the element $facen$ variable, we first loop through the boundary elements in the MPI table. Since the neighbour array of an element can be completely different after refinement, we erase the old neighbour element messages related to the neighbour rank using Algorithm 6.11 *Erase_old_face*. What Algorithm 6.11 does is first located the neighbour elements that are facing the current *target_rank* and delete them. Next,

we match the keys received on the other side of the MPI boundary with the possible neighbour array of the current element. Now we can make use of the *mpi_length* variable inside the MPI table. It is the element length that is exposed to the MPI boundary. We execute the matching procedure until we reach the full *mpi_length*. Due to the uniqueness of the key value, there is zero possibility of mismatching neighbours. Therefore, the algorithm we propose is robust.

Algorithm 6.10 Update_hash: update variable $facen$ (neighbour information) in element hash table.

Input: $recv\_info$, $table$, $facei$, num, $target\_rank$, $neighbours$

Use Algorithms: Algorithm 6.11 Erase_old_face

1: **for** $it = table[target\_rank].begin \rightarrow table[target\_rank].end$ **do**
2: $it\_face = local\_hash[it \rightarrow local\_key] \rightarrow facen[facei].begin$
3: $Erase\_old\_face(it\_face, it, target\_rank, facei)$ ▷ Erase the old face information related to this *target_rank*.
4: $l\_tol = 0$
5: **for** $itn = neighbours[it \rightarrow local\_key].begin \rightarrow neighbours[it \rightarrow local\_key].end$ **do** ▷ Find the matching element in the possible neighbour array.
6: **for** $k = 0 \rightarrow num$ **do**
7: **if** $recv\_info[k].local\_key == *(itn)$ **then** ▷ Find the neighbour. $*()$ is the operation to attract the value of the iterator itn.
8: **put** $recv\_info[k]$ **in** $local\_hash[it \rightarrow local\_key] \rightarrow facen[facei]$ ▷ Store the neighbour's information in the $facen$ variable.
9: $l\_tol \mathrel{+}= recv\_info[k].mpi\_length$ ▷ Accumulate the length of neighbour elements.
10: **break**
11: **end if**
12: **end for**
13: **if** $l\_tol \geq (it \rightarrow mpi\_length)$ **then** ▷ If neighbour length adds up to be equal or greater to the current element's *mpi_length*, then we can stop matching.
14: **break**
15: **end if**
16: **end for**
17: **end for**

Algorithm 6.11 Erase_old_face: erase the old MPI boundary information on target face: $facei$.

Input: it_face, it, $target\_rank$, $facei$

1: **for** $it\_face = it\_face \rightarrow local\_hash[it \rightarrow local\_key] \rightarrow facen[facei].end$ **do**
2: **if** $it\_face \rightarrow face\_type == 'M'$ **and** $it\_face \rightarrow rank == target\_rank$ **then**
3: $local\_hash$ **erase** this face
4: **end if**
5: **end for**

At this point, we indicate how to update the MPI boundary information on one side. For the solver, each direction needs to call the $Sender\_recver$ routine twice to fully update the element boundaries. The full MPI boundary update algorithm is presented as Algorithm 6.12 $Update\_mpi\_boundaries$.

Algorithm 6.12 Update_mpi_boundaries: algorithm to update MPI boundaries in one direction.

Input: $table1$, $table2$, $face1$, $face2$, $neighbours1$, $neighbours2$

Use Algorithms: Algorithm 6.9 Sender_recver

1: $Sender\_recver(table1, table2, face2, neighbours2)$
2: $Sender\_recver(table2, table1, face1, neighbours1)$

Combining the algorithms of updating the MPI table with the refinement algorithms, the complete process is formed: Algorithm 6.13 *Adapt*. The algorithm works as follows: first, elements are examined by the error estimator. Elements that need to be refined are flagged by their refinement type. Then, the actual refinement is applied. Before we update the MPI boundaries, we clear the old MPI tables, and reconstruct them. Finally, we update the MPI boundary information of the boundary elements.

Algorithm 6.13 Adapt: apply hp-refinement to local element and update the interprocessor boundaries.

Use Algorithms: Algorithm 6.7 Construct_mpi_table
Algorithm 6.12 Update_mpi_boundaries

1: Flag element that needs to be refined.
2: Apply refinement routines.
3: Clear old MPI tables.
4: $Construct\_mpi\_table(north, 1, neighbours\_north, south, 0, neighbours\_south)$ ▷ x direction.
5: $Update\_mpi\_boundaries(north, 1, neighbours\_north, south, 0, neighbours\_south)$
6: $Construct\_mpi\_table(east, 3, neighbours\_east, west, 2, neighbours\_west)$ ▷ y direction.
7: $Update\_mpi\_boundaries(east, 3, neighbours\_east, west, 2, neighbours\_west)$

At this point, we have fully explained the interprocessor boundaries update methods. The algorithms we have presented support valid parallel communications of a dynamic mesh. In the next chapter, we introduce the dynamic load balancing algorithms.

Chapter 7

Dynamic Load Balancing

Today, more and more scientists and engineers turn to High-Performance Computers (HPCs) to parallelize intensive calculations, in order to expand the complexity of problems that can be addressed and produce results in reasonable turnaround times. HPCs have been developed based on distributed memory systems and rely on good load balancing to fully exploit their computing power. The process of redistributing the computational load among processors is called *repartitioning*. For static applications, *i.e.*, applications without load changes, a static partition approach that balances the workload on the computation platform is sufficient to tackle the load imbalance problem. However, for programs that have dynamic workload, for instance, AMR applications, the varying workload could result in high load imbalance among the processors, which degrades the performance of the supercomputers. Therefore, a dynamic load balancing algorithm is required to redistribute the workload. However, a dynamic load balancing routine introduces extra overhead to the application. The load balancing routine not only contributes to the execution time of the program, but also affects the interprocessor boundary configurations, which implicitly impact the communication time. Moreover, memory consumption of the load balancing routines can become the bottleneck of an application on a distributed memory system.

Thus, the objectives for a well-developed repartitioning algorithm are clear

1. The load balancing algorithm should be executed fast with small memory overhead.
2. The repartitioning process should diminish the load imbalance effectively.
3. The new interprocessor boundaries after repartitioning should be minimized, so that heavy communication overhead is avoided.

Two types of repartitioning methods are explained and compared in Section 7.1. A space-filling curve (SFC) based repartitioning approach is implemented in our work. We first introduce SFC in Section 7.2. Then its generation algorithm is presented in Section 7.3. The repartitioning

algorithm is explained in detail in Section 7.4. Next, the interprocessor boundaries update method is given in Section 7.5. The element transfer method can be found in Section 7.6. We summarize the load balancing algorithm in Section 7.7. Finally, we showcase of the results of dynamic load balancing with our wave equation solver (Section 7.8).

7.1 Repartitioning Methods

The workload redistribution problem is an optimization problem with two main goals: the workload should be distributed evenly among the processors with small memory overhead and the interfacing boundaries between partitions should be as small as possible [31]. The optimization problem has been proven to be NP-hard [14]. Two popular approaches have been developed: *graph-based* repartitioning algorithms and *space-filling curve based* repartitioning algorithms. We briefly introduce both in Section 7.1.1 and Section 7.1.2, respectively.

7.1.1 Graph-based Repartitioning Algorithms

A mesh based problem can be comprehended as a graph, with each element as a node/vertex in the graph and the connections between elements as the edges in the graph. Then the nodes represent the computational cost and the edges indicate the communication cost. Therefore, repartitioning becomes a problem of dividing the nodes in groups of equal size, while minimizing the number of edges connecting vertices of different groups [45].

A graph $G = (V, E)$ is a combination of vertices (nodes) V and edges E, where each edge $e = \{v, u\}$ contains a pair of vertices $(v, u \in V)$. One can assign a positive weight to a vertex and/or an edge. The weight of a vertex v and an edge e is denoted by $\eta(v)$ and $\theta(e)$, respectively. The default value of the weight is one. A *neighbourhood* of vertex $v \in V$ is the set of adjacent vertices $\Gamma(v)$. A partition of V into k non-empty and disjoint subsets $V_1, V_2, V_3, ..., V_k$ is the so-called *k-way partitioning* of G. We call each V_i a partition.

The degree of balance of the k partition is defined as the maximum ratio of

$$max\left(\frac{\eta(V_i)}{\overline{\eta}(V)}\right), \quad i = 1, ..., k, \tag{7.1}$$

where $\eta(A)$ is the sum of the vertex weights of the vertices in set A. $\overline{\eta}(V)$ denotes the average weights. Therefore, a well-balanced partition would have a balance ratio of one. Otherwise, the balance ratio is larger than one. The edges that connect the vertices are being *cut* when partitioning. The sum of the weight of the cut edges is called the *edgecut* in k-way partitioning.

With a user defined parameter ϵ, the k-way partitioning problem of G is an optimization problem of finding a way to partition G with minimum edgecut, while keeping the balance ratio below $\epsilon + 1$.

Graph-based partitioning algorithms have been studied for many years and are now a well-developed field. Thus, there are many open source partitioning libraries implemented based on graph-based theories: for example, *ParMetis* [71], *Scotch* [60] and *Party* [56]. To obtain a high-quality and fast-generated partition, most of these use the *multilevel method* [13] and the *spectral method* [13, 33].

Although graph-based partitioning algorithms have solved the problem very well for a long time, graph-based partitioning algorithms seem to be reaching their scalability limits [31]. Since graph-based algorithms require complete graph information, their memory consumption grows linearly with graph size. Thus, this feature hinders their efficiency on distributed-memory machines. Another partitioning method is becoming more prominent: *space-filling curves based* repartitioning algorithms. This alternative provides a memory-efficient way to perform the partition but also maintains a high-quality repartitioning result. This approach will be introduced in the next section.

7.1.2 Space-Filling Curves based Repartitioning Algorithms

Space-Filling Curve (SFC) based repartitioning algorithms have gained more attention in recent years due to their fast runtime, good balancing results and low memory overheads. SFCs map elements in multidimensional geometric arrays in to a one-dimensional array, so one can perform partition of the mesh in one dimension.

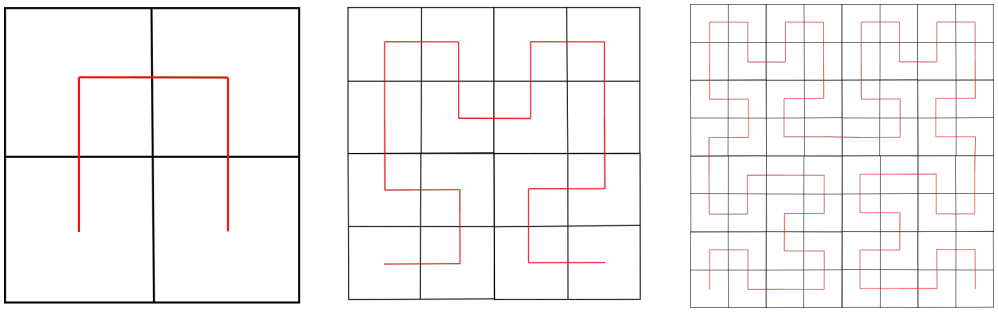
Space-filling curves not only simplify a multidimensional partitioning problem, they also offer desirable properties with respect to the linear mapping: they preserve good *locality* between objects in the multidimensional space. This means that the points that are near each other in the linear space stay close to each other in the target multidimensional space. Many researchers use SFCs to partition multidimensional graphs. Ou and Ranka [59], Pilkington and Baden [61] have proven that SFC-based partitioning techniques offer considerably good quality partitions with low cost. Additionally experimental and analytical studies have been done to show SFC-based repartitioning algorithms provide a reasonable communication cost [1, 3, 57]. Moreover, algorithms have been developed to generate different kinds of SFCs efficiently in parallel [57, 17].

SFC-based partitioning algorithms are able to render a high-quality balance with a low-memory requirement. The diversity and flexibility of space-filling curves give researches more possibilities to tailor them to their applications. Therefore, in this work, we embrace SFC-based partitioning algorithms and show how they are suitable for our application.

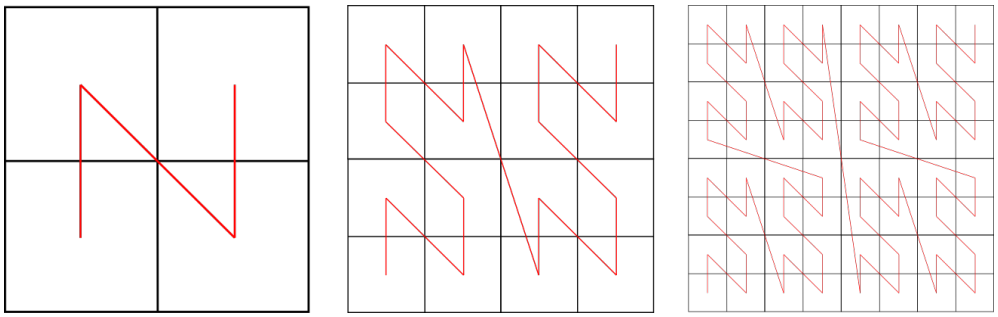
7.2 Space-Filling Curves

7.2.1 Introduction to Space-Filling Curves

In 1890, G. Peano constructed the first curve that passes through every point of a two-dimensional closed square, *i.e.*, $[0,1]^2$. The curves with such property are now called *space-filling curves*, or *Peano curves* [68]. D. Hilbert (1891), E.H. Moore (1900), H. Lebesgue (1904) *et al.* followed, discovering many different space-filling curves. Although these curves were initially designed to fill a two-dimensional closed square region, they are now capable of filling three-dimensional spaces or even higher dimensions [57, 68]. Moreover, they are adjusted to fit with arbitrary and dynamic multidimensional data [3].



(a) Hilbert curve (first three levels).



(b) Morton curve (first three levels).

Figure 7.1: Two popular SFCs: Hilbert curve and Morton curve

Two popular SFCs are the *Hilbert curve* and the *Morton curve* (or *z-ordering curve*) (Fig. 7.1). In the following section we will discuss these two curves and explain why we choose the Hilbert curve over the Morton curve.

7.2.2 Morton Curve

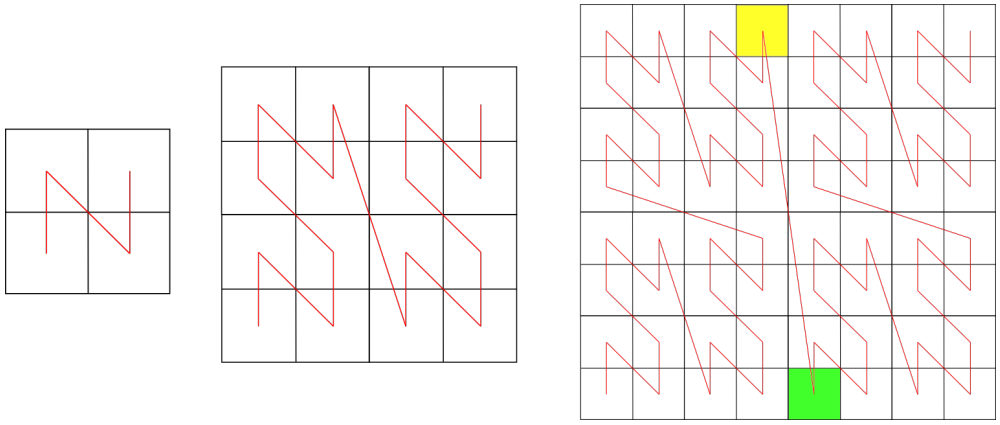


Figure 7.2: Morton curves for different grid sizes/levels. The Morton curve can be generated fast but "jumps" are inevitable in its pattern, for example, the jump from the yellow cell to the green cell.

In Fig. 7.2, we show the Morton curve in 2×2, 4×4, and 8×8 grid sizes. The curve of a $2^k \times 2^k$ grid can be constructed with four $2^{k-1} \times 2^{k-1}$ curves on the four quadrants. This rule applies for all space-filling curves.

One of the advantages of the Morton curve is that the position of each element in the ordering can be determined by interleaving the bits of the x and y coordinates of the elements, this is not easy for the Hilbert curve [69]. Another advantage of the Morton curve is that the recursion necessary for its generation is easy to specify. As we can see in Fig. 7.2, the curve's pattern at each refinement is identical to its ancestor, *i.e.*, there is no rotation or reflection performed. Therefore, the Morton curve can be generated very fast, which makes it competitive [17]. However, because of the simplicity of its generation pattern, there can be big "jumps" in its linearization: for example, in Fig. 7.2, the curve jump from the upper yellow cell to the lower green cell in the 8×8 mesh. This character degrades the locality of Morton curve.

7.2.3 Hilbert Curve

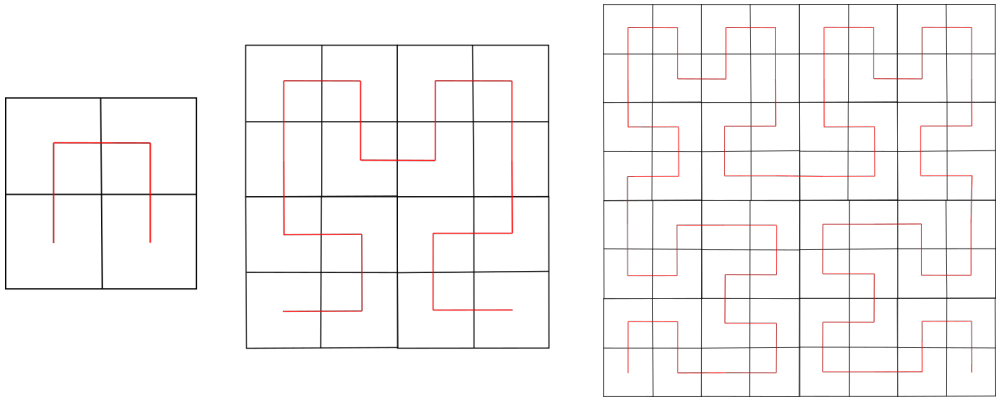


Figure 7.3: Hilbert curves for different grid sizes/levels.

The first three levels of the Hilbert curve are shown in Fig. 7.3. Higher levels are formed by combining four ancestors of the lower level. However, unlike the Morton curve, to go to the next level, extra rotations and inversions are needed, which contributes to the complexity of the Hilbert curve generation algorithm. However, the rotations and inversions make the points on the curve close to their neighbours in the target space (avoiding sudden jumps). The Hilbert curve outperforms the other space-filling curves in preserving locality [36]. Experiments and analytical studies have shown that the Hilbert curve's good locality is beneficial to the interprocessor communication patterns [1, 57].

Fast execution is essential for a successful load balancing algorithm, therefore, simple Morton ordering is by far the most frequently used space-filling curve comparing to other curves [68]. However, Campbell *et al.*, have proven that by using their algorithm, there would be no appreciable difference in the generation times of Morton and Hilbert curves [17]. Similar techniques have also been offered by [34].

Since we have a competitive Hilbert curve generation algorithm, we choose to adopt the Hilbert curve as our space-filling curve. Now we can be benefit from the compactness of the Hilbert curve and without sacrificing the execution time. In the next section, we explain the fast-generated Hilbert curve algorithm.

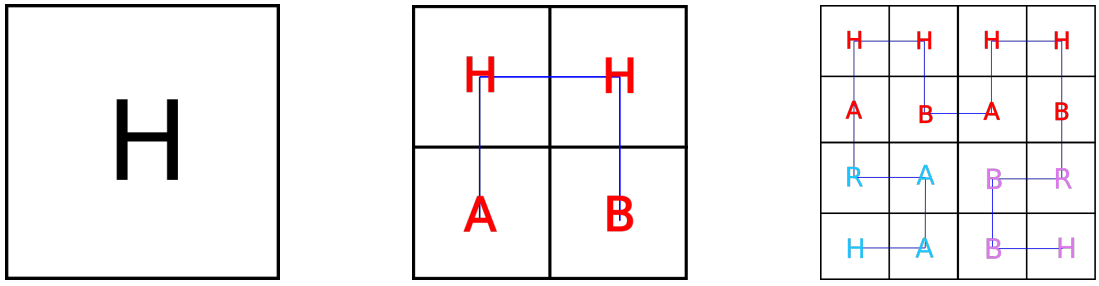
7.3 Fast-generated Hilbert Curve Algorithm

In this section, we first survey the existing Hilbert curve generation algorithms, then we explain the efficient Hilbert curve generation algorithm we choose.

7.3.1 Classifications

The generation of Hilbert curves can be classified into two categories: recursive algorithms [27, 80] and iterative algorithms [17, 28, 34]. Iterative algorithms, especially table-driven algorithms are much faster than recursive algorithms [47]. In this section we introduce a table-driven algorithm implemented in our application.

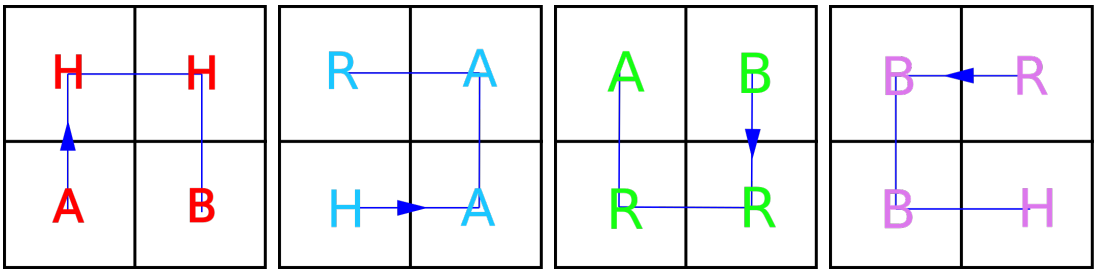
7.3.2 A Table-driven Algorithm



(a) Level 0 (b) Level 1 (c) Level 2

Figure 7.4: Construction of the Hilbert curve. Each cell has its own status s ($s \in \{H, A, B, R\}$). The status of the parent cell determines the geometrical arrangement of the children cells.

The generation of the Hilbert curve has a certain pattern that we can summarize into a table. Fig. 7.4 shows the first three levels of the Hilbert curve development. Every time the grid refines, each cell is divided into four equally-sized children cells with their corresponding statuses s. The four statuses are: $\{H, A, B, R\}$. The status $S(v)$ of a cell v determines its children's geometrical arrangement (Fig. 7.5). Here we use $C(v, j)$ to represent the jth child of cell v. For the 2D case, $j = 0, ..., 3$. The status of the jth child $S\left(C(v, j)\right) = S^C\left(S(v), j\right)$ can be derived from Table 7.1.



(a) Arrangement 0: A, H, H, B

(b) Arrangement 1: H, A, A, R

(c) Arrangement 2: B, R, R, A

(d) Arrangement 3: R, B, R, H

Figure 7.5: Four types of geometrical arrangements of the children cells. With the status $S(v)$ of a parent cell v, its children's status sequence can be derived by using Table 7.1. Then the children's geometrical arrangement can be found in the four cases shown. The colour scheme in this figure matches with the colour applied to the children sequence in Table 7.1.

$S^C(s,j)$	$j=0$	$j=1$	$j=2$	$j=3$
$s=H$	A	H	H	B
$s=A$	H	A	A	R
$s=R$	B	R	R	A
$s=B$	R	B	B	H

Table 7.1: Children cell statuses table: children status values $S^C(s,j)$ for 2D Hilbert-curve. s refers to the status of the parent element, j refers to the jth child. The colour scheme in the table corresponds to that of the geometrical arrangements in Fig. 7.5.

Given the parent cell's status, its children's statuses can be obtained by using Table 7.1. For instance, from the level 0 Hilbert curve (Fig. 7.4a) to the level 1 Hilbert curve (Fig. 7.4b), the curve evolves as follows: $S^C(H,j) \rightarrow A, H, H, B, \quad j = 0, ..., 3$. The geometrical arrangement of the children sequence is indicated by Fig. 7.5a.

To effectively describe the geometrical arrangement of the curve, *i.e.*, the shape of the curve, we number four sibling cells from 0 to 3 (Fig. 7.6) and use the number to indicate the position of a child cell. We treat four siblings as a unit on the curve. Thus, a cell at position 2 means it is at the upper right corner of the unit. By describing a unit by the element position numbers, one can determined the trace of the curve. For example, the arrangement of the curve in Fig. 7.5c can be described as 2301.

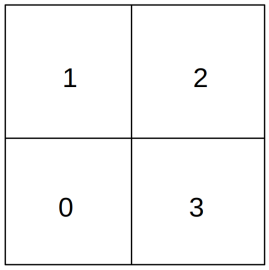


Figure 7.6: The position numbers of four sibling cells. The number indicates the geometrical position of a cell among four sibling cells.

From the description above, with the status of the parent cell, we can obtain an array of the children statuses, which has a specific geometrical arrangement. We summarize the children geometrical arrangements in Table 7.2 by using the position numbers we defined above. We use $P\left(C(v,j)\right) = P^C\left(S(v),j\right)$ to depict the jth child's position of parent v.

$P^C(s,j)$	$j=0$	$j=1$	$j=2$	$j=3$
$s=H$	0	1	2	3
$s=A$	0	3	2	1
$s=R$	2	3	0	1
$s=B$	2	1	0	3

Table 7.2: Children cell positions table: children position values $P^C(s,j)$ for 2D Hilbert-curve. s refers to the status of the parent cell, j refers to the jth child.

With the help of Table 7.1 and Table 7.2, we can build the Hilbert curve efficiently by using the status of the parent cell. For instance, if the lower left cell splits into four children cells in Fig. 7.4b, the geometrical arrangement of the children would be $A \rightarrow 0321$, and their statuses are $A \rightarrow HAAR$. Instead of using a recursive algorithm to form the Hilbert curve from the first level, we can quickly generate the $(i+1)st$ level Hilbert curve from the ith level Hilbert curve with the status of the parent cell. In the program, this table-driven algorithm can be achieved in parallel by using two lookup tables. Note that this algorithm also allows us to mix different levels of Hilbert curve in the same mesh, like the situation in Fig. 7.7. We now can control the mesh resolution by locally increasing the Hilbert curve level.

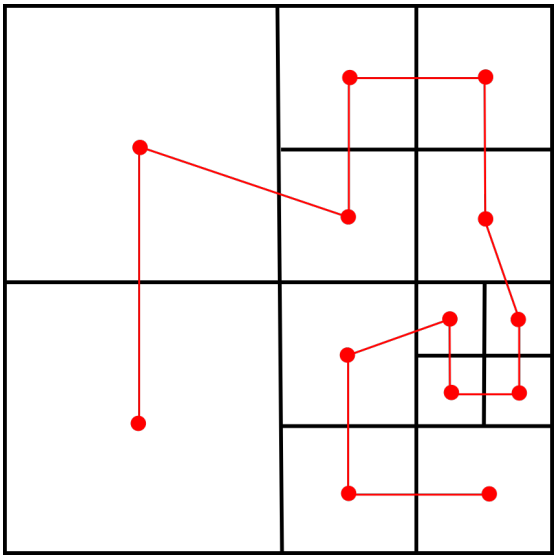


Figure 7.7: The Hilbert curve traverses the mesh with mixing levels.

7.4 Workload Redistribution Algorithms

7.4.1 Chains-on-chains partition

With the help of the Hilbert curve, we successfully reduce our multidimensional partitioning problem into a one-dimensional one. Balancing of a one-dimensional workload distribution is also known as *chains-on-chains partitioning* (CCP) [62]. The CCP problem is to divide a chain of N tasks with non-negative weights associated with them, into a sequence of P consecutive parts, so that the maximum load among all processors, *i.e.*, the *bottleneck*, is minimized [62]. An example will be given in Section 7.4.2.

Partitioning Strategy

With the task number and workload associated to the task changing during execution time, we need a strategy to repartition the workload dynamically. Miguet *et al.* [55] provide very promising 1D heuristics to accomplish the work. The splitting points on the curve can be determined almost entirely locally, with the exception of finding the average workload (globally). We explain the heuristics in the following section. The important symbols used in the heuristics are given in Table .7.3.

Symbols of load balancing variables	
N	Local element number.
P	Total processor number.
w_i	Weight of the ith element.
$prefix(i)$	Local prefix sum of ith element.
$W(i)$	Global prefix sum of ith element.
W_{global_sum}	Total element weight.
w_{ideal}	Ideal workload for each processor.
w_{opt}	Optimal workload for each processor.
L_s	Local sum of the workload of the partition s.
L_{max}	Maximum local workload among the processors, or the so-called bottleneck of a partition.
E	The efficiency of the load balancing.
E_{opt}	The optimal efficiency that can be achieved by the repartition.
$pmapping$	The rank number of the element's owner of the new partition.
$proc\_mapping\_table$	Global processor mapping table. Contains the first element's global ID in a specific rank.
$Send.pre$	A vector of element's keys, for those to be reallocated to the previous processor.
$Send.next$	A vector of element's keys, for those to be reallocated to the next processor.

Table 7.3: Summary of the important symbols in the load balancing algorithm.

We serialize an arbitrary mesh into a one-dimensional vector using a Hilbert curve. Each processor has N nodes on the local curve, where nodes represent the elements. Each node (element) has its own weight w_i: the computational load for each fluid element is considered to be $O(n^4)$, where n is the number of grid points in each direction [82]. Then we compute the local prefix sum (*i.e.* cumulative workload for that node) of node I as

$$prefix(I) = \sum_{i=0}^{I} w_i, \quad 0 \leq I < N. \tag{7.2}$$

In order to compute the global prefix sum (*i.e.* the global cumulative sum of the workload) among the processors, we need the weight offsets (*i.e.* the cumulative workload sum of the previous processor) of each segment of the curve. Luckily, standard MPI provides this function to calculate offsets as the exclusive prefix sum, MPI_Exscan, with the reduction operator MPI_SUM. Thus, we obtain the global prefix sum $W(i) = prefix(i) + offset$.

With the global prefix sum, we can easily get the ideal workload per partition

$$w_{ideal} = \frac{W_{global\_sum}}{P}, \tag{7.3}$$

where W_{global_sum} is the global total workload. P is the total processor number.

There are two ways to get the ideal workload w_{ideal} on each processor. The first is to first use $MPI\_Allreduce$ with MPI_SUM as the operator, then all processors share the global total workload's value (W_{global_sum}). Next each processor computes the ideal workload w_{ideal} locally.

The second is for only the last processor to compute the ideal workload since it has the global total workload locally, then it needs to use MPI_Bcast to broadcast the optimal workload to other processors. Both approaches result in similar run time and memory complexity [31]. In our work, we implement the latter one.

With the ideal workload, the splitting positions on the curve can be decided completely locally. Up to now, we only use global collective operations twice. No more communication is needed for this process. Then the elements will be transferred, if any, among the local processor and its two neighbour processors since they are arranged on a one-dimensional curve. Following this strategy, the partition decisions are largely simplified. Both global collective operations have the time complexity $O(logP)$ [64, 70], thus, this approach favours large scale computing.

For a partition s that contains N elements, the local load is

$$L_s = \sum_{i=0}^{N-1} w_i, \quad 1 \leq s \leq P. \tag{7.4}$$

Using this heuristic, the load efficiency [31, 46] is limited by

$$E = \frac{w_{ideal}}{L_{max}}, \tag{7.5}$$

where L_{max} is the maximum local workload among the processors. The efficiency of the distributed workload is defined by the slowest processor. Thus, L_{max} is the bottleneck of the existing partition. The goal of a partition is to minimize the bottleneck L_{max}. The efficiency is bounded by $\frac{1}{P} \leq E \leq 1$. In the real world scenario, one might find out that the load efficiency E cannot always be achieved as 1. This is due to the computational loads on the SFC being an array of discrete values (an element has to reside on one processor): there is therefore potentially a gap between optimal efficiency E_{opt} and ideal efficiency 100%. We define the minimum bottleneck as the optimal workload w_{opt}. The optimal load efficiency one can get is $E_{opt} = \frac{w_{ideal}}{w_{opt}}$.

7.4.2 Repartitioning Algorithm

With the knowledge of the local partition locations on each processor, the next step is to transfer the elements. Element transfer is one of the challenges in load balancing: the sender processors know which elements to send away and where they should be sent, however, the receivers do not know when to receive or from whom they should receive, nor do they know the size of the messages that are coming. In the MPI point-to-point communication scheme, the receiver needs to know the sources of the message prior to the communication. Different algorithms are developed to tackle this problem. We list some popular solutions and our choice in the following sections.

All-to-all Collective Method

The first solution is to use the MPI all-to-all operation to inform all the processors about their message exchange partners before the actual data transfer. One of the choices is to use $MPI\_Alltoallv$. The logic of this method is straightforward and it is also the method-of-choice of many applications. Since this a global collective operation, its memory usage has a linear relationship with processor number, *i.e.*, the memory complexity is $O(P)$, where P is the number of processor. Some architectures optimize this operation to make the process efficient enough [44].

Iterative Method

The second method is to transfer the elements iteratively. The elements are marked with the destination process and forwarded in a virtual ring topology until they reach their destinations [31]. In this way, we avoid all-to-all communication, therefore, making it more memory-friendly. This method can maintain its efficiency if the new partition is only slightly changed from the current element distribution. If the weights vary slowly and the repartition is done frequently, there are a few message exchanges. However, in the worst case scenario, this could need $O(P)$ forward messages. Therefore, for large numbers of processes, this algorithm can be very inefficient.

Centralized Method

The third method is to use a root processor, typically the first processor p_0, to decide the partition locations on the curve then broadcast the partitioning result to other processors so that the new partition is known globally [82]. The algorithm works as follows: all processors send their local computational load to the root processor p_0; on p_0, the prefix sum of each element is computed then the splitting point on the curve is decided; finally, the new partition is broadcast to all the processors. This approach is also very intuitive: we give p_0 the global information and spread the result to the rest of the processors. So this method is similar to the first method mentioned above: processors are informed with global information of the new partition. The disadvantage of this method is that the memory usage grows linearly with the problem size. Additionally, the algorithm is limited by the memory size of the root processor since it needs to hold all the elements' weights.

Distributed Method

The distributed method, proposed by Zhai *et al.* [82], is similar to the first all-to-all collective method. After deciding the splitting position on the curve locally, the processors communicate with their neighbours and then use a collective operation $MPI\_Allgatherv$ to inform all the

processors about the data transfer pattern. We implement this method in our application. In the following part, we explain this method with a concrete example.

Step-by-step Explanation with a Four Processors Example

The distributed method has the following steps:

1. Compute the local and global prefix sums.

   In Fig. 7.8, we show an initial partition among four processors. Processors are discriminated by their colours. Each element inside the figure can be seen as a node on the curve as in Fig. 7.7.

   The fluid element computational workload is indifferent to the size of the element. It is the polynomial order of an element that counts. The load is treated as the fourth power of the number grid points n along each direction ($O(n^4)$). Therefore, the element workload is

   $$load = (porder + 1)^4, \tag{7.6}$$

   where *porder* is the polynomial order in one direction (in our program we use the same polynomial order in both directions.). Since the value of the element workload will grow with the fourth order, we normalize the load value between 1 and 2, *i.e.*, $load \in [1, 2]$. The normalized element workload can be calculated as

   $$load_{normalized} = \frac{load - load_{min}}{load_{max} - load_{min}} + 1.0, \tag{7.7}$$

   where $load_{min}$ and $load_{max}$ are the minimum workload and maximum workload respectively. They can be pre-calculated based on the user-defined polynomial order range. We present this algorithm as Algorithm 7.2 *Elem_load* in the next section.

Proc num	1				2				3				4			
Global ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight	1	1	1	2	2	3	3	3	2	1	1	1	1	2	2	1
Local prefix sum	1	2	3	5	2	5	8	11	2	3	4	5	1	3	5	6
Local sum	5				11				5				6			

Figure 7.8: Four processors example: initial element partition. The colour scheme in the figure represents different processors.

An example is given in Fig. 7.8. With the local prefix sums calculated locally, we then use a standard MPI function to calculate the global prefix sum (Fig. 7.9).

Proc num	1	2	3	4
Exclusive prefix sum	0	5	16	21

Figure 7.9: Four processors example: exclusive prefix sum.

Calling MPI_Exscan, we get the exclusive prefix sums of the workload offsets for each processor. Adding this value to the local prefix sum, we obtain the global prefix sum (Fig. 7.10).

Proc num	1				2				3				4			
Global ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight	1	1	1	2	2	3	3	3	2	1	1	1	1	2	2	1
Local prefix sum	1	2	3	5	2	5	8	11	2	3	4	5	1	3	5	6
Global prefix sum	1	2	3	5	7	10	13	16	18	19	20	21	22	24	26	27
Local sum	5				11				5				6			

Figure 7.10: Four processors example: adding the exclusive prefix sum value to the local prefix sum, the global prefix sum is obtained.

2. Calculate the ideal workload and decide the partition locations on the curve.

   On the last processor, we have the total workload value as the global prefix sum of the last element. Therefore, the ideal workload is readily computed as $w_{ideal} = \frac{W_{global\_sum}}{P} = \frac{27}{4} = 6.75$.

   Then, we broadcast the ideal workload to each processor by using MPI_Bcast. Here, we define a variable called processor mapping number $pmapping$, which is the future processor's rank number:

$$pmapping(i) = int\left(\frac{W_i - 0.01}{w_{ideal}}\right). \tag{7.8}$$

   The future processor rank number is the integer part of the ratio of the element global prefix sum, minus 0.01, to the ideal workload. Since the rank value starts with zero, we subtract the element global prefix sum by 0.01. The results are illustrated in Fig. 7.11.

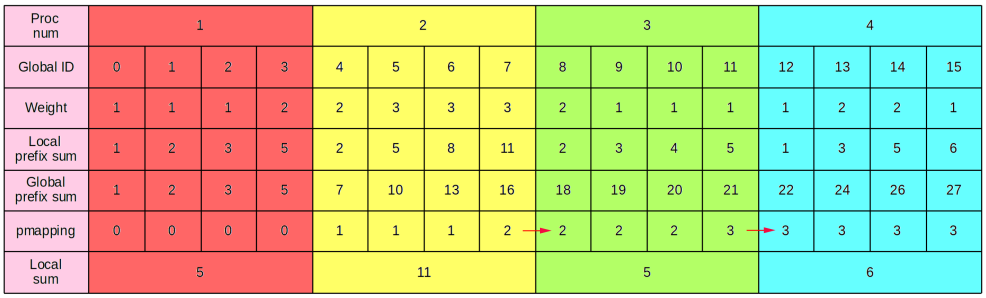


Figure 7.11: Four processors example: compute the processor mapping values. The value of *pmapping* is the element's future rank number. Note that Processor 2 and 3 will need to send their last elements to the next processor (indicated by the arrows).

At this point, we obtain the new partition result. As indicated by the *pmapping* values, there is no element movement for Processor 1, while Processor 2 needs to send its last element to Processor 3, and Processor 3 needs to send its last element to Processor 4. We successfully reduce the multidimensional repartitioning problem into a one-dimensional one. The element geometrical arrangement on each processor is decided by the Hilbert curve, which guarantees a good locality. Thus, we obtain the new partitions and good partition boundaries at the same time.

3. Form the global processor mapping table.

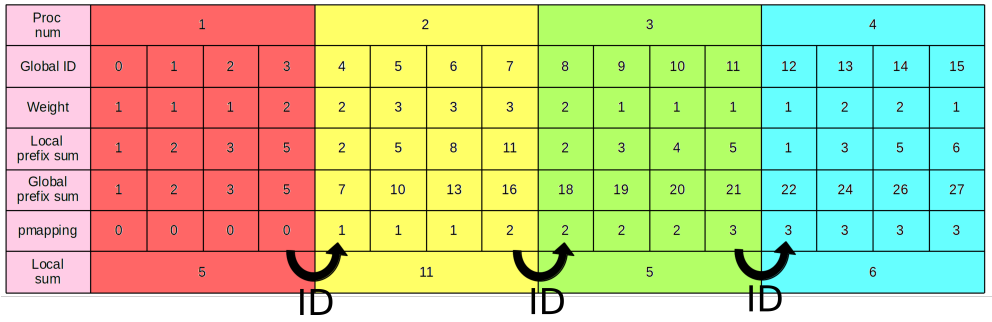


Figure 7.12: Four processors example: each processor sends their last element's global ID with the *pmapping* value to the next processor, if any.

Since each processor has decided the new partition locally, next, we need to inform the processors about the global element exchange pattern. We form a processor mapping table *proc_mapping_table*, which is the collection of the first element's global ID with the rank number in the new partition. We first communicate between the neighbour processors: starting from the first processor, each processor sends their last element's *pmapping* value to the next processor, if any (see Fig. 7.12).

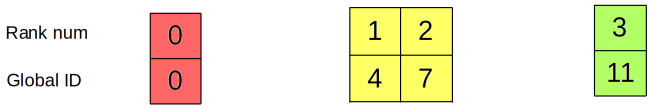


Figure 7.13: Four processors example: local *proc_mapping_table*.

Next, each processor forms the *proc_mapping_table* locally. Each records its starting element's global number of a specific *pmapping* value and compares their first column's *pmapping* value ($pmapping = rank$) in the local *proc_mapping_table* with what they receive from the previous processor: if the *pmapping* values coincide, then they erase the objects in the first column of their table. For example, Processor 2 forms the yellow table shown in Fig. 7.13. It owns two different *pmapping* values: 1 and 2. The element global IDs of the first occurrence of these two *pmapping* values are 4 and 7 respectively. Thus, Processor 2 initially forms the local *proc_mapping_table* as Fig. 7.13 shows. Then it receives from Processor 1 with the message $pmapping = 0$, which does not coincide with the *pmapping* value in the first column of Processor 2's table. Thus, the first column of its *proc_mapping_table* is retained. On the contrary, Processor 4 receives $pmapping = 3$ from Processor 3, which equals the first and only column's *pmapping* value, therefore, the new partition information has already been recorded by the previous processor, so we erase it from the local table. Note that Processor 1 always has at least one column in its *proc_mapping_table* since it owns the first element.

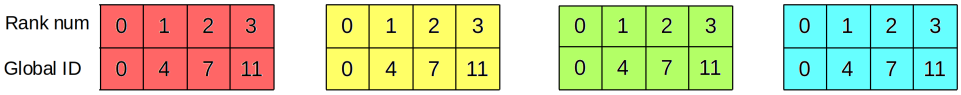


Figure 7.14: Example: global *proc_mapping_table*

Next, we use an all-to-all collective operation ($MPI\_Allgatherv$) to form the same global processor mapping table on each processor, so that everybody can learn their and their neighbours' communication pattern from this table. The processor mapping table is indicated in Fig. 7.14.

4. Element transfer.

   When calculating the *pmapping* values, two vectors are created locally to store the key of the elements that will be transferred to the previous neighbour processor (*Send.pre*) and the next neighbour processor (*Send.next*). These two vectors can be formed before the creation of the global *proc_mapping_table*. We utilize the newly calculated *pmapping* value: if $pmapping < local\ rank\ number$, then we put this element's key in the *Send.pre* vector, else if $pmapping > local\ rank\ number$, this element's key belongs to *Send.next* vector. Note that the data transfer is restricted between neighbour processors on the

curve, which means one processor can have at most two neighbours to whom it can transfer workload. This could potentially hinder the repartition technique to reach the optimal efficiency if the dynamicity of the application is significant or the load balancing frequency is low. If such a situation is encountered, we apply a recursive repartitioning algorithm to meet the optimal efficiency. We will explain the recursive repartitioning algorithm in the next section (Section 7.4.2).

After the global *proc_mapping_table* is formed, we can start to reallocate elements. The elements are sent away to the neighbour processors based on the *Send.pre* and *Send.next* vectors. The receiver side should call receive based on the *proc_mapping_table*. In this way we accomplish an efficient element redistribution.

Proc num	1				2			3				4				
Global ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight	1	1	1	2	2	3	3	3	2	1	1	1	1	2	2	1
Global prefix sum	1	2	3	5	7	10	13	16	18	19	20	21	22	24	26	27
Local sum	5				8			7				7				

Figure 7.15: Four processors example: load balancing result.

The new partition result is presented in Fig. 7.15. Here we evaluate the efficiency of our load balancing algorithm. Before load balancing, the workload efficiency based on Equation 7.5 is $E_{before} = \frac{w_{ideal}}{L_{max}} = \frac{6.75}{11} \approx 61\%$. After the repartitioning, the new load efficiency is $E_{after} = \frac{w_{ideal}}{L_{max}} = \frac{6.75}{8} \approx 84\%$. We manage to largely increase the load balance by using the distributed load balancing algorithm.

In summary, the idea behind chains-on-chains repartitioning algorithm is easy to understand and implement. SFC-based repartitioning algorithm largely simplifies the workload redistribution problem. It not only provides well-balanced results, but also offers acceptable partition boundaries based on the natural locality of SFC. Only a few collective operations are required and the load balancing algorithm is highly independent, both of which are beneficial to the parallel efficiency of the program.

Whole Algorithm

In this section, we present the pseudocodes for the repartitioning algorithms. Note that after we finish building the global processor mapping table, we update the neighbour information of the local elements. The interprocessor interfaces update method will be explained in the next section (Section 7.5).

We present Algorithm 7.1 *Build_mapping_table* to generate the repartitioning pattern. It executes as follows:

1. Calculate the global prefix sum on each element.
2. Last processor computes the average workload and broadcasts the value globally.
3. Each processor decides the partition locally and use MPI collective operation to form the global processor mapping table, *proc_mapping_table*.
4. Each processor updates the elements' local neighbour information independently (use Algorithm 7.3 *Update_neighbours*).

The meaning of the symbols used in load balancing algorithms can be found in Table 7.3.

Algorithm 7.1 Build_mapping_table: build the processor mapping table. Processors transfer elements based on such table.

Use Algorithms: **Algorithm 5.2 Get_key_fun**
Algorithm 7.2 Elem_load
Algorithm 7.3 Update_neighbours

1: $temp = head$ ▷ A temporary pointer points to the head of the local linked list.
2: $lprefix\_load[local\_elem\_num]$ ▷ Allocate space to compute load prefix sum.
3: $lprefix\_load[0] = Elem\_load(temp \to n)$ ▷ Compute the element workload.
4: $temp = temp \to next$
5: **for** $k = 1 \to local\_elem\_num$ **do** ▷ Get the local prefix sums.
6: $lprefix\_load[k] = Elem\_load(temp \to n) + lprefix\_load[k-1]$
7: $temp = temp \to next$
8: **end for**
9: $local\_load\_sum = lprefix\_load.back$ ▷ The sum of the local load is the last prefix sum of the last element.
10: $exscan\_sum = 0$ ▷ The load of the previous processor.
11: $MPI\_Exscan(sendbuf : local\_load\_sum, recvbuf : exscan\_sum, count : 1, MPI\_Datatype : MPI\_DOUBLE, MPI\_OP : MPI\_SUM)$
12: $load\_avg = 0$ ▷ Initialize ideal workload.
13: **if** last processor **then** ▷ Last processor compute the ideal workload.
14: $load\_tol = exscan\_sum + local\_load\_sum$
15: $load\_avg = load\_tol/$ (number of proc)
16: **end if**
17: $MPI\_Bcast(buffer : load\_avg,\ count : 1, MPI\_Datatype : MPI\_DOUBLE, root : last\ proc)$
18: $temp = head$ ▷ Move the temporary pointer to the head of the linked list.
19: $proc\_pre = -1$
20: **for** $k = 0 \to local\_elem\_num$ **do**
21: $lprefix\_load[k]$ += $exscan\_sum$ ▷ Get the global prefix sum.
22: $pmapping = int\left(\frac{lprefix\_load[k]-0.01}{load\_avg}\right)$
23: **if** $pmapping \neq proc\_pre$ **then**
24: **put** $\{pmapping, element\ global\ ID\}$ **in** $proc\_mapping\_table$
25: $proc\_pre = pmapping$
26: **end if**
27: **if** $pmapping < rank$ **then**
28: $key = Get\_key\_fun(temp \to index[0], temp \to index[1], temp \to index[2])$
29: $Send.pre \leftarrow key$ ▷ Put this key in the sending list targeted the previous processor.
30: $my\_rank\_first = temp \to next$ ▷ Record the new head of the linked list.

31: **else if** $pmapping > rank$ **then**

32: $key = Get\_key\_fun(temp \rightarrow index[0], temp \rightarrow index[1], temp \rightarrow index[2])$

33: $Send.next \leftarrow key$ ▷ Put this key in the sending list targeted the next processor.

34: **else**

35: $my\_rank\_last = temp$ ▷ Record the last element of the new linked list.

36: **end if**

37: $temp = temp \rightarrow next$

38: **end for**

39: **if** $rank \neq (num\_proc - 1)$ **then** ▷ Processors send the last *pmapping* value to the next processor, except the last processor.

40: $last\_rank =$ **last** *pmapping* **in** *proc_mapping_table*

41: $MPI\_Send(buffer : last\_rank,\ count : 1,\ MPI\_Datatype : MPI\_INT,\ destination : rank + 1,\ tag : rank + 1)$

42: **end if**

43: **if** $rank \neq 0$ **then** ▷ Processors receive from the previous neighbour, except the first processor.

44: $MPI\_Recv(buffer : pre\_rank,\ count : 1,\ MPI\_Datatype : MPI\_INT,\ source : rank - 1,\ tag : rank)$

45: $first\_rank =$ **first** *pmapping* **in** *proc_mapping_table*

46: **if** $first\_rank == pre\_rank$ **then**

47: *proc_mapping_table* **erase** the first object

48: **end if**

49: **end if**

50: $sizet =$ **size of** *proc_mapping_table*

51: **allocate** $sizea[num\_proc]$

52: $MPI\_Allgather(sendbuf : sizet,\ sendcount : 1,\ MPI\_Datatype : MPI\_INT,\ recvbuf : sizea,\ recvcount : 1,\ MPI\_Datatype : MPI\_INT)$ ▷ Gather the size of the partial *proc_mapping_table* on each processor. Preparing to form the global *proc_mapping_table*.

53: **allocate** $recv\_buff[num\_proc * 2]$ ▷ Buffer for the global *proc_mapping_table*

54: $MPI\_Allgatherv(sendbuf : local\ partial\ proc\_mapping\_table,\ recvbuf : recv\_buff)$ ▷ Collective operation to form the complete *proc_mapping_table*.

55: **complete building** *proc_mapping_table*

56: $Update\_neighbours()$ ▷ Update element's local neighbours in $facen$ before element transfer.

Algorithm 7.2 Elem_load: element computation workload.

Input: $porder$, $nmin$, $nmax$

Output: $load$

1: $load\_min = (nmin + 1)^4$

2: $load\_max = (nmax + 1)^4$

3: $load = \frac{(porder+1)^4 - load\_min}{load\_max - load\_min} + 1.0$

Next we present the local neighbours update algorithm: Algorithm 7.3 *Update_neighbours*. Since an element's neighbour information is stored in the $facen$ variable, we need to update its content for the new partition. Note that this algorithm only updates elements' local neighbour information, which does not include the element's interprocessor interfaces. In the next section, we will disclose the interprocessor boundaries update algorithms. Here the update process is limited to the local interfaces.

Since in Algorithm 7.1 *Build_mapping_table* processors create two vectors containing the keys of the elements to be moved in the future, we can make use of these two vectors to achieve our goal. The first vector *Send.pre* contains the keys of the elements that need to be moved to the previous processor. For each element in this vector, we loop through its four faces and locate its local neighbour ($face\_type = 'L'$). There are three circumstances to deal with:

1. The neighbour's key is found in *Send.next*.

   Since we are in the *Send.pre* list, then this result suggests that these two elements will be on two different processors. We again change the $face\_type$ to be $'M'$.

2. The neighbour's key appears in *Send.pre*.

   The element's neighbour will be allocated to the same (previous) processor, thus, they stay as local neighbours. The $face\_type$ stays intact, we only need to update the $rank$ number to be the future rank.

3. Its neighbour's key is neither in *Send.pre*, nor in *Send.next*.

   This indicates that its neighbour remains local while the element itself is moved to the previous processor, *i.e.*, after load balancing, they will be located on different processors. We change both of their $face\_type$ to be MPI boundary $'M'$.

A similar process also applies to the *Send.next* vector. Although this time, we do not need to check whether the element's neighbour is in the *Send.pre* list or not, since if it is, information has been updated in the previous process. When we update the element $facen$ in the two sending lists, in the meantime, we need to update the corresponding $facen$ of the neighbour by using Algorithm 7.4 *Neighbour_change*.

Algorithm 7.3 Update_neighbours: update the element's local neighbours information in the $facen$ variable before element transfer.

Use Algorithms: Algorithm 7.4 Neighbour_change

```
1: for key in Send.pre do
2:     for facei = 0 → 3 do
3:         for it = facen[facei].begin → facen[facei].end do
4:             if it → face_type ==′L′ then
5:                 n_key = it → key
6:                 if not find n_key in Send.pre then          ▷ If neighbour's key is
                                                                 not inside Send.pre.
7:                     if not find n_key in Send.next then     ▷ Neighbour's key is
                                                                 not in Send.next.
8:                         it → face_type =′M′                 ▷ Its neighbour stays local.
9:                         Neighbour_change(facei, n_key, key, rank − 1) ▷ Updates neighbour's
                                                                 facen.
10:                    else                                    ▷ Find neighbour's key in Send.next.
11:                        it → face_type =′M′
12:                        it → rank = rank + 1
13:                        Neighbour_change(facei, n_key, key, rank − 1)
14:                    end if
15:                else                                        ▷ Neighbour's key is in Send.pre.
16:                    it → rank = rank − 1
17:                end if
18:            end if
19:        end for
20:    end for
21: end for
22: for key in Send.next do
23:    for facei = 0 → 3  do
24:        for it = facen[facei].begin → facen[facei].end do
25:            if it →==′L′ then
26:                n_key = it → key
27:                if not find n_key in Send.next then
28:                    it → face_type =′M′
29:                    it → rank = rank
30:                    Neighbour_change(facei, n_key, key, rank + 1)
31:                else
```

```
32:                     it → rank = rank + 1
33:                 end if
34:             end if
35:         end for
36:     end for
37: end for
```

The elements' $facen$ contents are interactive: once we modify a specific face information, we have to make a corresponding adjustment to the neighbour's $facen$. Therefore, the Algorithm 7.4 $Neighbour\_change$ is used to make corresponding adjustments to the neighbour element's $facen$ variable. With the input of the current element's face number, $facei$, its neighbour's key, n_key, the key of current element, my_key, and the rank number, $rank$, that the current element will reside on. The algorithm only changes the $face\_type$ and $rank$ variables, since these two are the only data that change. Inside this algorithm, another Algorithm 7.5 $Opposite\_dir$ is invoked to get the opposite face number of the input $facei$.

Algorithm 7.4 Neighbour_change: update the corresponding neighbour $facen$. Note that the neighbour element will remain local.

Input: $facei$, n_key, my_key, $rank$

Use Algorithms: Algorithm 7.5 Opposite_dir

```
1: oface = Opposite_dir(facei)                    ▷ The face number of the neighbour.
2: for it = local_hash[n_key] → facen[oface].begin
                → local_hash[n_key] → facen[ofacen].end do
3:     if it → my_key then
4:         it → face_type ='M'
5:         it → rank = rank
6:         break
7:     end if
8: end for
```

The following Algorithm 7.5 $Opposite\_dir$ returns the opposite face number of the input face number dir. The element face configuration is given in Fig. 6.1. Although the logic of this algorithm is simple, we suggest to implement this algorithm as a static lookup table, which can be built and stored in the memory during compile time.

Algorithm 7.5 Opposite_dir: input the current face number, return the opposite face number.

Input: *dir*

```
1: if dir == 0 then
2:     return 1
3: else if dir == 1 then
4:     return 0
5: else if dir == 2 then
6:     return 3
7: else if dir == 3 then
8:     return 2
9: end if
```

7.5 Interprocessor Boundaries Update Method

To update the MPI boundaries for the new partition, we utilize the MPI tables processors formed after applying refinement. Note that we update the MPI boundary before the actual movement of the workload, therefore, the interprocessor elements have not changed, which makes the data in the MPI tables valid.

Recall the data structure of the MPI table (Fig. 6.3): we have saved one variable *owners_rank* unused. This variable stores the future rank of the MPI boundary element. We present Algorithm 7.6 *Ownership_one_dir* to fill the existing MPI table with the future rank numbers of the boundary elements. We check each element in the MPI table to see if it is in the *Send.pre* list and *Send.next* list: if found, we record its future rank accordingly, if not, this element remains local.

Algorithm 7.6 Ownership_one_dir: fill the ownership of the elements in the MPI table in one direction.

Input: *mtable*, *Send.pre*, *Send.next*

```
1: for v1 in neighbour processors' rank in MPI table do
2:     for v2 in the array of key corresponding to the current neighbour processor do
3:         if find v2 → key in Send.pre then          ▷ This element will move to the previous processor.
4:             v2 → owners_rank = rank − 1
5:         else if find v2 → key in Send.next then    ▷ This element will move to the next processor.
6:             v2 → owners_rank = rank + 1
7:         else
8:             v2 → owners_rank = rank
9:         end if
10:    end for
11: end for
```

After we record the future ranks of the boundary elements, we are ready to pack up the MPI boundary information and exchange it. Similar to the process of updating MPI boundaries after refinement, in each direction we exchange messages twice. With the sender's table, *sendo*, the receiver's table, *recvo*, and the face direction to be updated, $facei$, we pack the *key* values of the elements in *sendo* and their ownership ranks, $owners\_rank$, corresponding to the neighbour processor and send the message out. To assist the communication, we define our customized MPI data type, *Owner_type*, to hold the *key* value and $owners\_rank$. Here we still use a dynamic receiving approach. With the help of MPI_Probe, the message size is detected, then we apply the actual receive. Once the message is received correctly, we call Algorithm 7.8 $Update\_mpib$ to renew the MPI boundaries.

Algorithm 7.7 Send_recv_ownership: exchange MPI boundary tables to updates the MPI boundaries.

Input: *sendo*, *recvo*, *facei*

Use Algorithms: Algorithm 7.8 Update_mpib

1: $size\_s = sendo.size$ ▷ Size of the sender's table.
2: $size\_r = recvo.size$ ▷ Size of the receiver's table.
3: hash map: $send\_info < int, vector < owner\_struct >>$ ▷ Use hash table data structure to hold the sending information. The key is the target rank number, the value is a vector of type *owner_struct*.
4: $isend = 0$
5: **if** $size\_s > 0$ **then**
6: **for** $v \in sendo$ **do**
7: $target\_rank = v.first$
8: $num\_elem = v.second.size$
9: **allocate** $send\_info[target\_rank] = vector < owner\_struct > (num\_elem)$
10: $it =$ **beginning of** $v.second$
11: **for** $k = 0 \rightarrow num\_elem$ **do**
12: $send\_info[target\_rank][k] = it \rightarrow local\_key$
13: $send\_info[target\_rank][k] = it \rightarrow owners\_rank$
14: $++it$
15: **end for**
16: $MPI\_Isend(sendbuf : send\_info, count : num\_elem, MPI\_Datatype : Owner\_type, destination : target\_rank, tag : rank)$
17: $isend \mathrel{+}= 1$
18: **end for**
19: **end if**
20: **if** $size\_r > 0$ **then**
21: **for** $v \in recvo$ **do**
22: $num = 0$
23: $MPI\_Probe(source : v \rightarrow target\_rank, tag : v \rightarrow target\_rank)$
24: $MPI\_Get\_count(count : num, MPI\_Datatype : Owner\_type)$
25: **allocate** $recv\_info[num]$
26: $MPI\_Recv(recvbuf : recv\_info, count : num, MPI\_Datatype : Owner\_type, source : v \rightarrow target\_rank, tag : v \rightarrow target\_rank)$
27: $Update\_mpib(recv\_info, recvo, facei, num, v \rightarrow target\_rank)$
28: **end for**
29: **end if**
30: **if** $size\_s > 0$ **then**
31: $MPI\_Waitall()$
32: **end if**

The algorithm to update MPI boundaries after receiving the message from neighbour processors is divided into two parts: (1) Algorithm 7.8 $Update\_mpib$ locates the target object in the $facen$ variable. (2) Algorithm 7.9 $Change\_face$ compares the future rank of the target element with the old information: if the ranks differ, then we update the information accordingly. Since the mesh stays still, the related variables we need to update are $face\_type$ and $rank$.

Algorithm 7.8 Update_mpib: update the MPI boundaries based on the element ownership received.

Input: $recv\_info$, $otable$, $facei$, $num1$, $target\_rank$

Use Algorithms: Algorithm 7.9 Change_face

```
1: num = num1/2
2: for ito = otable.begin → otable.end do
3:     for it_face = local_hash[ito → local_key] → facen[facei].begin →
               local_hash[ito → local_key] → facen[facei].end do
4:         if it_face → face_type ==′M′ and it_face → rank == target_rank then
5:             Change_face(num, recv_info, ito, it_face)
6:         end if
7:     end for
8: end for
```

Algorithm 7.9 Change_face: compare the neighbour's new rank with the old one and update the $facen$ accordingly.

Input: num, $recv\_info$, ito, it_face

```
1: for k = 0 → num do
2:     if recv_info[k].owners_rank == it_face → owners_rank then      ▷ The neighbour
                                                  element will be moved to current rank.
3:         it_face → face_type =′L′
4:         it_face → rank = ito → owners_rank
5:     else                              ▷ The neighbour element will not be in this rank.
6:         rank_old = it_face → rank
7:         if rank_old ≠ recv_info[k].owners_rank then          ▷ This element will be
                                                                  sent to a new rank.
8:             it_face → rank = recv_info[k].owners_rank
9:         end if
10:    end if
11: end for
```

At this point, all the pieces to update MPI boundaries before we reallocate elements have been explained. Here we combine them together and present the full algorithm to update MPI boundaries: Algorithm 7.10 $Update\_mpi\_boundaries\_LB$. Firstly, we record the future ranks

of the elements in the MPI tables. Then we exchange messages and update the boundaries. After executing this routine, elements are equipped with the neighbour information for the new partition.

Algorithm 7.10 Update_mpi_boundaries_LB: update the MPI boundaries before repartitioning.

Use Algorithms: **Algorithm 7.6 Ownership_one_dir**
Algorithm 7.7 Send_recv_ownership

1: $Ownership\_one\_dir(north)$ ▷ Fill the MPI tables with the future ranks of the elements.
2: $Ownership\_one\_dir(south)$
3: $Ownership\_one\_dir(east)$
4: $Ownership\_one\_dir(west)$
5: $Send\_recv\_ownership(north, south, 0)$ ▷ Update boundaries in x direction.
6: $Send\_recv\_ownership(south, north, 1)$
7: $Send\_recv\_ownership(west, east, 3)$ ▷ Update boundaries in y direction.
8: $Send\_recv\_ownership(east, west, 2)$

7.6 Reallocate Elements

In the former sections, we explained the repartition strategy and the update of the MPI boundaries. Now we are ready to reallocate the workload. To redistribute elements, we pack the crucial information of each element in three buffers: the element's character buffer, the element's neighbour buffer and the element's solution buffer. Since the first two buffers have a mixture of data types, we define two MPI derived data types for a more efficient data transfer pattern. Then we apply point-to-point communication between processors.

The symbols for the variables used in the reallocation process are given in Table 7.4.

Symbols for element reallocation process	
$info\_pack$	Data structure for element characters.
$Elem\_type$	Element character data type.
$face\_pack$	Data structure for element neighbours' information.
$Face\_type$	Element neighbour data type.
$head$	Pointer to the first element (before repartition) in the linked list.
my_rank_first	Pointer to the first element (after repartition) in the linked list.
my_rank_last	Pointer to the last element in the linked list.

Table 7.4: Symbols of variables in element reallocation

7.6.1 Derived Data Types

In this section, we showcase the two MPI derived data types we defined to support the element transfer.

The first derived data type is for element character information, for instance, the element integer coordinates, element status, and element reference space. The full data structure is given in code Listing 7.1.

```
struct info_pack{

        int n;  // element polynomial order.

        int index[3]; // element integer coordinates.

        char status; // element Hilbert status.

        int child_position; // element relative position.

        double xcoords[2]; // physical coordinate in x
            direction.
        double ycoords[2]; // physical coordinate in y
            direction.

        double ref_x[2]; // reference space in x direction.
        double ref_y[2]; // reference space in y direction.

};
```

Listing 7.1: Data structure for element characters: *info_pack*

We named the customized MPI data type *Elem_type*. The method to create the MPI derived data type *Elem_type* is similar to what was shown in Section 6.3.1. One can easily modify Listing 6.3 to create other desired data structures.

The second derived data type is named *face_pack*: it manages all the neighbour information. The MPI derived data type is called *Face_type*.

```
struct face_pack{

        long long int owners_key; // the key of the owner
```

```
    int facei; // face direction.

    char face_type; // neighbour's type.

    int hlevel; // neighbour's refinement level.

    int porderx; // neighbours' polynomial order in x
        direction.

    int pordery; // neighbours' polynomial order in x
        direction.

    long long int key; // neighbour's key.

    int rank;        // neighbour's rank.

    double ref_x[2]; // neighbour's reference space in x
         direction.
    double ref_y[2]; // neighbour's reference space in y
         direction.

};
```

Listing 7.2: Data structure for element neighbours' info: *face_pack*

7.6.2 Element Transfer Algorithms

Next, we present the Algorithm 7.11 *Reallocate_elem* to redistribute elements using the MPI data types we just derived. Since each processor has its element sending lists, *Send.pre* and *Send.next*, the sending phase of the communication is very intuitive. On the other hand, the receiver can make its decision of when to receive and from whom to receive based on the processor mapping table.

The reallocation algorithm starts with the sending process. If there is an object in the *Send.pre* list, we pack the element character information, element solutions, and element neighbours situations in *send_elem_p*, *solu_packed_p* and $face\_info\_p$ with the corresponding data types. For elements in the *Send.next* list we apply the similar operations.

Then the receiver will decide whether to call receive or not. If a receive operation is required, then it needs to know from whom it should expect to receive. Except for the first and the last processor, each processor has two neighbour processors in the 1D partition. Therefore, we separate the two special occasions from the general case.

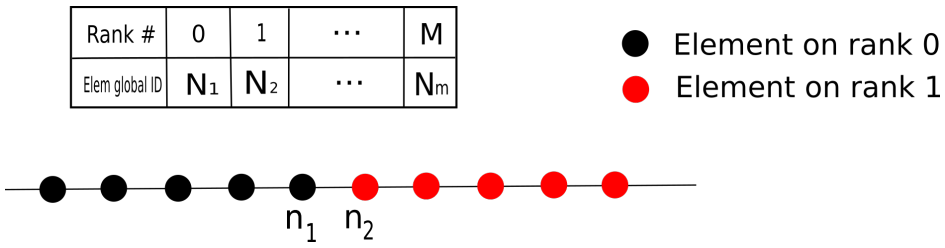


Figure 7.16: Receiver decision-making for rank 0. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours.

We use Fig. 7.16 to illustrate how Rank 0/Processor 1 (rank number = processor number - 1) makes its receiving decision. The processor mapping table is above the line which represents the workload distribution. In this case, we assume there are M ranks and in total m elements. The elements on two processors are discriminated by their colours. The last element's global ID in Processor 1 is n_1 and the first element's global ID in Processor 2 is n_2 with the relationship $n_1 + 1 = n_2$. The processor mapping table provides the first element's global ID of the new partition. Therefore, if $N_2 - 1 > n_1$, then Processor 1 should receive elements from its neighbour, Processor 2.

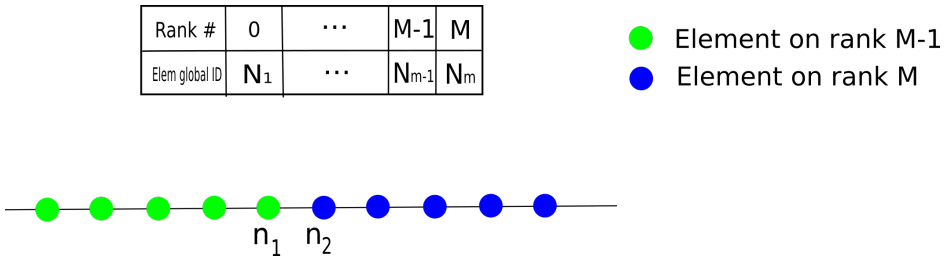


Figure 7.17: Receiver decision-making for the last rank. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours.

Similar logic applies to the last rank, which also has one neighbour processor on the curve. We still use n_1 and n_2 to represent the last element's ID on rank $M-1$ and the first element's ID on the rank M. This time, if $N_m < n_2$, then it should receive elements from the previous rank.

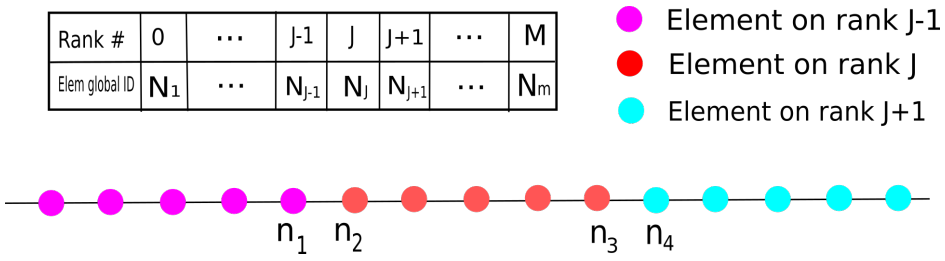


Figure 7.18: Receiver decision-making for ranks in between. The processor mapping table indicates the future workload distribution. The elements on two processors are discriminated by their colours.

The general case applies to the processors between the first and the last ones. They all have two neighbour processors on the curve. We discuss the receiving pattern between Rank J and its previous rank, $J-1$, and next rank, $J+1$. The involved elements' IDs are indicated in Fig. 7.18. If $N_J < n_2$, then Rank J needs to receive from the previous rank, $J-1$. If $N_{J+1}-1 > n_3$, then Rank J needs to receive from the next rank, $J+1$. The receiving strategies for receivers are easy to deduce with the help of the processor mapping tables.

These strategies are translated into Algorithm 7.11 *Reallocate_elem*.

Algorithm 7.11 Reallocate_elem: redistribute the elements.

Use Algorithms: **Algorithm 7.12 Send_pack**
Algorithm 7.13 Solution_pack
Algorithm 7.14 Face_pack
Algorithm 7.15 Erase_elem_old
Algorithm 7.16 Recv_elem
Algorithm 7.17 Recv_solu
Algorithm 7.18 Recv_face
Algorithm 7.19 Enlarge_hash
Algorithm 7.20 Fill_facen

1: *start* = global ID of the first local element
2: *last* = global ID of the last local element
3: *num_pre* = **size of** *Send.pre*
4: *num_next* = **size of** *Send.next*
5: **allocate** *send_elem_p*[*num_pre*] ▷ Buffer for element character info.
Allocating buffers for the info to send to the previous processor.
The following operations share the same purpose.
6: **init** *solution_pack_p* ▷ Buffer for element solutions.
7: **init** *face_info_p* ▷ Buffer for element face info.

8: **allocate** $send\_elem\_n[num\_next]$ ▷ Allocating buffers for the info to send to the next processor. The following operations share the same purpose.

9: **init** $solution\_pack\_n$

10: **init** $face\_info\_n$

11: **if** $num\_pre > 0$ **then** ▷ Need to transfer elements to the previous processor.

12: $it = Send.pre.begin$

13: $solu\_num = 0$

14: $Send\_pack(send\_elem\_p, it, solu\_num)$ ▷ Pack element character info.

15: $solu\_packed(Send.pre, solu\_num, solu\_pack\_p)$ ▷ Pack element solutions.

16: num_n

17: $Face\_pack(face\_info\_p, Send.pre, num\_n)$ ▷ Pack element face info.

18: $MPI\_Isend(buf : send\_elem\_p, count : num\_pre, MPI\_Datatype : Elem\_type,$
$destination : rank - 1, tag : rank)$

19: $MPI\_Isend(buf : solu\_packed\_p, count : solu\_num,$
$MPI\_Datatype : MPI\_DOUBLE, destination : rank - 1, tag : rank + 3)$

20: $MPI\_Isend(buf : face\_info\_p, count : num\_n, MPI\_Datatype : Face\_type,$
$destination : rank - 1, tag : rank + 1)$

21: $Erase\_elem\_old(Send.pre, 'p', num\_pre)$

22: **end if**

23: **if** $num\_next > 0$ **then** ▷ Need to transfer element to the next processor.

24: $it = Send.next.begin$

25: $solu\_num = 0$

26: $Send\_pack(send\_elem\_n, it, solu\_num)$

27: $Solution\_pack(Send.next, solu\_num, solu\_packed\_n)$

28: $num\_n = 0$

29: $Face\_pack(face\_info\_n, Send.next, num\_n)$

30: $MPI\_Isend(buf : send\_elem\_n, count : num\_next, MPI\_Datatype : Elem\_type,$
$destination : rank + 1, tag : rank)$

31: $MPI\_Isend(buf : solu\_packed\_n, count : solu\_num,$
$MPI\_Datatype : MPI\_DOUBLE, destination : rank + 1, tag : rank + 4)$

32: $MPI\_Isend(buf : face\_info\_n, count : num\_n, MPI\_Datatype : Face\_type,$
$destination : rank + 1, tag : rank + 1)$

33: $Erase\_elem\_old(Send.next, 'n', num\_next)$

34: **end if**

35: **if** $rank == 0$ **then** ▷ Apply receiving strategy to the first rank.
36: **if** $proc\_mapping\_table[1] - 1 > last$ **then**
37: **init** $recv\_info$ ▷ Element character info. Use data structure $info\_pack$.
38: **init** $recv\_face$ ▷ *facen* info. Use data structure $face\_pack$.
39: **init** $solu$ ▷ Element solutions.
40: $recv\_num = 0$
41: $Recv\_elem(rank + 1, rank + 1, recv\_info, recv\_num)$
42: $Recv\_solu(rank + 1, rank + 4, solu)$
43: $Recv\_face(rank + 1, rank + 2, recv\_face)$
44: $Enlarge\_hash(recv\_info, 'n', recv\_num, solu)$
45: $Fill\_facen(recv\_face)$
46: **end if**
47: **else if** $rank == num\_proc - 1$ **then** ▷ Apply receiving strategy to the last rank.
48: **if** $proc\_mapping\_table[rank] < start$ **then**
49: **init** $recv\_info$ ▷ Element character info. Use data structure $info\_pack$.
50: **init** $recv\_face$ ▷ *facen* info. Use data structure $face\_pack$.
51: **init** $solu$ ▷ Element solutions.
52: $recv\_num = 0$
53: $Recv\_elem(rank - 1, rank - 1, recv\_info, recv\_num)$
54: $Recv\_solu(rank - 1, rank + 3, solu)$
55: $Recv\_face(rank - 1, rank, recv\_face)$
56: $Enlarge\_hash(recv\_info, 'p', recv\_num, solu)$
57: $Fill\_facen(recv\_face)$
58: **end if**
59: **else** ▷ Rank in between, general strategy.
60: **if** $proc\_mapping\_table[rank + 1] - 1 > last$ **then** ▷ Receive from the latter processor.
61: **init** $recv\_info$ ▷ Element character info. Use data structure $info\_pack$.
62: **init** $recv\_face$ ▷ *facen* info. Use data structure $face\_pack$.
63: **init** $solu$ ▷ Element solutions.
64: $recv\_num = 0$
65: $Recv\_elem(rank + 1, rank + 1, recv\_info, recv\_num)$
66: $Recv\_solu(rank + 1, rank + 4, solu)$
67: $Recv\_face(rank + 1, rank + 2, recv\_face)$
68: $Enlarge\_hash(recv\_info, 'n', recv\_num, solu)$
69: $Fill\_facen(recv\_face)$
70: **end if**

```
71:    if proc_mapping_table[rank] < start then        ▷ Receive from the previous processor.
72:        init recv_info              ▷ Element character info. Use data structure info_pack.
73:        init recv_face                          ▷ facen info. Use data structure face_pack.
74:        init solu                                                  ▷ Element solutions.
75:        recv_num = 0
76:        Recv_elem(rank − 1, rank − 1, recv_info, recv_num)
77:        Recv_solu(rank − 1, rank + 3, solu)
78:        Recv_face(rank − 1, rank, recv_face)
79:        Enlarge_hash(recv_info,′ p′, recv_num, solu)
80:        Fill_facen(recv_face)
81:    end if
82: end if
83: if num_pre > 0 then
84:    MPI_Wait()                    ▷ Ensure the previous processor receives the message.
85: end if
86: if num_next > 0 then
87:    MPI_Wait()                        ▷ Ensure the next processor receives the message.
88: end if
```

We need three separate algorithms to pack the element information: they are Algorithm 7.12 *Send_pack*, Algorithm 7.13 *Solution_pack* and Algorithm 7.14 *Face_pack*. The key idea of these three algorithms is to extract the element information required for the solver. They are presented below.

In Algorithm 7.12 *Send_pack*, we input the buffer of the element character information, *send_info*, an iterator, *it* (initially placed at the beginning of the element sending list), and the number of equations in the system. The algorithm returns the number of solutions, *solu_num*, of the current element. Then this number is used in the solution-packing algorithm (Algorithm 7.13).

Algorithm 7.12 Send_pack: pack element character information.

Input: $send\_info$, it, $num\_of\_equation$

Output: $solu\_num$

```
for v ∈ send_info do
    v → n = local_hash[*it] → n
    solu_num += (v.n + 1)^2          ▷ in our program we use same polynomial order
                                       in x and y direction.
    for i = 0 → 3 do
        v.index[i] = local_hash[*it] → index[i]
    end for
    v.status = local_hash[*it] → status
    v.child_position = local_hash[*it] → child_position
    for i = 0 → 2 do
        v.xcoords[i] = local_hash[*it] → xcoords[i]
        v.ycoords[i] = local_hash[*it] → ycoords[i]
        v.ref_x[i] = local_hash[*it] → ref_x[i]
        v.ref_y[i] = local_hash[*it] → ref_y[i]
    end for
    ++it
end for
solu_num = solu_num * num_of_equation
```

Algorithm 7.13 *Solution_pack* packs up all the element solutions together. It iterates all the elements in sending list *send_list*. The variable *solu_num* is the total number of solutions that is generates by Algorithm 7.12 *Send_pack*. Since the element solutions form a single data type, we pack them together and reallocate them to the new processors.

Algorithm 7.13 Solution_pack: pack up element solutions.

Input: $send\_list$, $solu\_num$

Output: $solu\_packed$

```
1:  allocate solu_packed[solu_num]
2:  i = 0
3:  for key ∈ send_list do
4:      for equ = 0 → num_of_equation do
5:          for v = local_hash[key] → solution[equ] do
6:              solu_packed[i] = v
7:              ++i
8:          end for
9:      end for
10: end for
```

In Algorithm 7.14 $Face\_pack$, we store all the element $facen$ information in a vector. In order to discriminate the ownership of the neighbour information (elements could have different numbers of neighbours on different faces.), we explicitly record the owner's key of a specific item in the $face\_info$ vector, and also the number of the element face.

Algorithm 7.14 Face_pack: store all the element neighbours in a vector.

Input: *send*

Output: $face\_info$, num

```
1:  for v ∈ send do
2:      for i = 0 → 4 do ▷ Record neighbours' information on four sides of element interface.
3:          for it = local_hash[v] → facen[i].begin → local_hash[v] → facen[i].end do
4:              face_info allocate one more face_pack struct
5:              ++num
6:              face_info.back.owners_key = v        ▷ Operate on the last object in face_info.
7:              face_info.back.facei = i
8:              face_info.back.face_type = it → face_type
9:              face_info.back.hlevel = it → hlevel
10:             face_info.back.porderx = it → porderx
11:             face_info.back.pordery = it → pordery
12:             face_info.back.key = it → key
13:             face_info.back.rank = it → rank
14:             face_info.back.ref_x/y = it → ref_x/y
15:         end for
16:     end for
17: end for
```

After the element transfer, we can discard the storage of the reallocated elements. Algorithm 7.15 $Erase\_elem\_old$ performs this job. First, we update the variable $local\_elem\_num$ which traces the local element number. The algorithm deals with the most extreme case: empty rank. If an empty rank is obtained, it sets the head pointer (*head*), which points to the beginning of the linked list, to be the null pointer to avoid *dangling pointers* [1]. The *dir* variable is a character: we use $'p'$ and $'n'$ to discriminate the sending list *send* between *Send.pre* and *Send.next*. The purpose of this process is to adjust the pointers of the linked list. If the sending list is *Send.pre* then the *head* pointer becomes invalid; we trace to the new first element on the curve and reassign its address to the *head* pointer. Note that instead of traversing the linked list to the new head element of the curve, which requires a linear execution time proportional to the length of the *Send.pre* list, we can directly use the end element's key to access the new head element. This is one of the advantages of using a hash table data structure: search operations require

[1] Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type.

a constant time with the key value. If $dir == 'n'$, means we need to adjust the pointer of the element at the end of the local curve. In the end, we erase the element in the sending list.

Algorithm 7.15 Erase_elem_old: erase the elements in the sending list from the local element hash table.

Input: *send*, *dir*, *num*, *local_elem_num*, *head*

```
1:  local_elem_num −= num
2:  if local_elem_num == 0 then
3:      head = nullptr                ▷ If there is no local element, set the head pointer to be null.
4:  else
5:      if dir == 'p' then
6:          last = send.back
7:          head = local_hash[last] → next
8:      else                                                              ▷ dir == 'n'
9:          my_rank_last → next = nullptr
10:     end if
11:     for v ∈ send do
12:         local_hash erase v
13:     end for
14: end if
```

Corresponding to the three element packing algorithms, we offer three receiving algorithms to deal with the three different MPI data types. The first one is Algorithm 7.16 *Recv_elem*, which use *Elem_type* to receive the expecting data. We still apply a dynamic receiving approach since receivers lack knowledge of the size of the expected data. The same strategy is adopted by Algorithm 7.17 *Recv_solu* and Algorithm 7.18 *Recv_face* to receive the element solutions and element neighbour information, respectively.

Algorithm 7.16 Recv_elem: receive element information from the sender.

Input: *source*, *tag*, *count*

Output: *recv_info*

```
1:  MPI_Probe(source : source, tag : tag)
2:  MPI_Get_count(MPI_Datatype : Elem_type, count : count)
3:  allocate recv_info[count]
4:  MPI_Recv(buffer : recv_info, count : count, MPI_Datatype : Elem_type,
            source : source, tag : tag)
```

Algorithm 7.17 Recv_solu: receive element solution from the sender.

Input: *source*, *tag*, *count*

Output: *solu*

1: $MPI\_Probe(source : source, tag : tag)$
2: $MPI\_Get\_count(MPI\_Datatype : MPI\_DOUBLE, count : count)$
3: **allocate** $solu[count]$
4: $MPI\_Recv(buffer : solu, count : count, MPI\_Datatype : MPI\_DOUBLE,$
$source : source, tag : tag)$

Algorithm 7.18 Recv_face: receive element's neighbour information from the sender.

Input: *source*, *tag*, *count*

Output: $recv\_face$

1: $MPI\_Probe(source : source, tag : tag)$
2: $MPI\_Get\_count(MPI\_Datatype : Face\_type, count : count)$
3: **allocate** $solu[count]$
4: $MPI\_Recv(buffer : recv\_face, count : count, MPI\_Datatype : MPI\_Face\_type,$
$source : source, tag : tag)$

After receiving all the element information, we need to transfer it to the element hash table and adjust the linked list among elements. The Algorithm 7.19 $Enlarge\_hash$ works as follows: first, we put the received element in to the hash table with its *key* value. In the meantime, we connect the received element in order by using pointers. Since the received elements are stored in $recv\_info$ buffer in the Hilbert curve order, we follow this order and build a new linked list among them. Finally, we connect the new linked list to the local linked list in the element hash table, then we have the full Hilbert curve among all the local elements. A temporary pointer *temp_head*, that points to the head element of the received element list is created to facilitate the process.

We first sort the element characteristic information in $recv\_info$, then the element solutions. In the end, we connect the newly formed linked list among the received elements to the old local linked list. If the elements have come from the previous rank ($dir = 'p'$), then the new linked list is placed at the front of the old linked list, with the *head* pointer pointing to the new first element. Otherwise, the elements have come from the next rank ($dir = 'n'$), and the new linked list is appended to the end of the old one. The extreme case is that the elements land on an empty rank, then we only need to reassign the *head* pointer.

Algorithm 7.19 Enlarge_hash: after receiving element information, we create new units in hash table and store them.

Input: $recv\_info$, dir, num_recv, $solu$

Use Algorithms: Algorithm 5.2 Get_key_fun

1: $temp\_head = nullptr$ ▷ Temporary pointer points to the head of the received list.
2: $pre\_key = 0$ ▷ Previous element's key.
3: $nodei = 0$
4: **for** $it = recv\_info.begin \rightarrow recv\_info.end$ **do**
5: $key = Get\_key\_fun((*it).index[0], (*it).index[1], (*it).index[2])$
6: **allocate new item in** $local\_hash[key]$ ▷ Allocate a new item in local element hash table with key value.
7: **if** $it == recv\_info.begin$ **then**
8: $temp\_head = local\_hash[key]$ ▷ $temp\_hash$ points to the first element of the received list.
9: **end if**
10: $local\_hash[key] \rightarrow n = (*it) \rightarrow n$
11: $local\_hash[key] \rightarrow m = (*it) \rightarrow n$ ▷ Uniform polynomial order.
12: **for** $i = 0 \rightarrow 3$ **do**
13: $local\_hash[key] \rightarrow index[i] = (*it) \rightarrow index[i]$
14: **end for**
15: $local\_hash[key] \rightarrow status = (*it) \rightarrow status$
16: $local\_hash[key] \rightarrow child\_position = (*it) \rightarrow child\_position$
17: **for** $i = 0 \rightarrow 2$ **do**
18: $local\_hash[key] \rightarrow xcoords[i] = (*it) \rightarrow xcoords[i]$
19: $local\_hash[key] \rightarrow ycoords[i] = (*it) \rightarrow ycoords[i]$
20: $local\_hash[key] \rightarrow ref\_x[i] = (*it) \rightarrow ref\_x[i]$
21: $local\_hash[key] \rightarrow ref\_y[i] = (*it) \rightarrow ref\_y[i]$
22: **end for**
23: $num\_solu = ((*it) \rightarrow n + 1)^2$ ▷ Current element solution number.
24: **for** $equ = 0 \rightarrow num\_of\_equation$ **do**
25: **allocate** $local\_hash[key] \rightarrow solution[equ]$ = **size** num_solu
26: **for** $i = 0 \rightarrow num\_solu$ **do**
27: $local\_hash[key] \rightarrow solution[equ][i] = solu[nodei]$
28: $++nodei$
29: **end for**
30: **if** $it \neq recv\_info.begin$ **then**
31: $local\_hash[pre\_key] \rightarrow next = local\_hash[key]$ ▷ Build the linked list among the received elements.
32: **end if**

33: $pre\_key = key$

34: **if** $local\_elem\_num == 0$ **then**

35: $head = temp\_head$ ▷ If there is no local element in the hash table, then the *head* pointer should point to the head of received element list.

36: **else** ▷ If local hash table is not empty.

37: **if** $dir == 'p'$ **then** ▷ Element from previous processor.

38: $local\_hash[pre\_key] \rightarrow next = my\_rank\_first$ ▷ Connect the newly formed linked list to the beginning the old local linked list.

39: $head = temp\_head$ ▷ Renew *head* pointer.

40: **else** ▷ Element from next processor.

41: $my\_rank\_last \rightarrow next = temp\_head$ ▷ Connect the newly formed linked list to the end of the old local linked list.

42: **end if**

43: **end if**

44: **end for**

45: **end for**

46: $local\_elem\_num \mathrel{+}= num\_recv$ ▷ Update the local element number.

At this point, only the element face information remains to be settled. Algorithm 7.20 $Fill\_facen$ takes the package of neighbour information $face\_info$ and uses it to update the corresponding element's face storage. Although we pack all the $facen$ messages of all the elements in a big buffer, we can discriminate them with the explicitly stored owner's key, *owners_key*, and its face number, $facei$. Therefore, the process is intuitive. The *owners_key* and $facei$ variables are important since an element's neighbour number is not a constant in each direction.

Algorithm 7.20 Fill_facen: put the neighbour information inside the element hash table.

Input: $face\_info$

1: **for** $v \in face\_info$ **do**
2: $local\_hash[v.owners\_key] \rightarrow facen[v.facei]$ **allocate new unit** ▷ Access the target element and target face, then allocate one unit for the following variables.
3: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.face\_type = v.face\_type$ ▷ Store the information in the newly allocated unit. Hereafter the same.
4: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.hlevel = v.hlevel$
5: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.porderx = v.porderx$
6: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.pordery = v.pordery$
7: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.key = v.key$
8: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.rank = v.rank$
9: **for** $i = 0, 1$ **do**
10: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.ref\_x[i] = v.ref\_x[i]$
11: $local\_hash[v.owners\_key] \rightarrow facen[v.facei].back.ref\_y[i] = v.ref\_y[i]$
12: **end for**
13: **end for**

7.7 Whole Load Balancing Algorithm Summary

Now we have all the pieces in the load balancing algorithm. We assemble them together and present the whole algorithm: Algorithm 7.21 *Load_balancing*. The algorithm works as follows: first, it refines the MPI table with the elements' future ownership. Second, it updates the MPI boundaries. Third, it performs the element reallocation. Two new algorithms are needed. Algorithm 7.22 *Clear_mapping_tables* to clear up the processor mapping tables; and Algorithm 7.23 *MPI_table_rebuild* to build the new MPI tables, which will be used in the MPI boundary element message exchange procedure.

Algorithm 7.21 Load_balancing: complete load balancing algorithm.

Use Algorithms: **Algorithm 7.1 Build_mapping_table**
Algorithm 7.10 Update_mpi_boundaries_LB
Algorithm 7.11 Reallocate_elem
Algorithm 7.22 Clear_mapping_tables
Algorithm 7.23 MPI_Table_rebuild

1: *Build_mapping_table*()
2: *Update_mpi_boundary*()
3: *Reallocate_elem*()
4: *Clear_mapping_tables*()
5: *MPI_table_rebuild*()

Algorithm 7.22 *Clear_mapping_table* clears up the processor mapping tables and other variables that are related to the repartitioning algorithms.

Algorithm 7.22 Clear_mapping_tables: clears up the processor mapping tables and other variables that are related to the repartitioning algorithms

1: **clear** *proc_mapping_table*
2: **clear** element sending lists.
3: $my\_rank\_first = nullptr$ ▷ Set local pointer to be null pointer.
4: $my\_rank\_last = nullptr$

Algorithm 7.23 *MPI_Tables_rebuild* first deletes the old MPI tables since the elements on the MPI boundaries have changed, then we use the *Construct_mpi_tables* algorithms to rebuild the new MPI tables.

Algorithm 7.23 MPI_Table_rebuild: rebuild the MPI tables in a and y directions.

Input: *north*, *south*, *east*, *west*,
neighbours_north, *neighbours_south*, *neighbours_east*, *neighbours_west*

Use Algorithms: **Algorithm 7.24 Clear_tables**
Algorithm 6.7 Construct_mpi_table

1: *Clear_tables*() ▷ Erase the old MPI tables.
2: *Construct_mpi_tables*(*north*, 1, *neighbours_north*, *south*, 0, *neighbours_south*) ▷ x direction.
3: *Construct_mpi_tables*(*east*, 3, *neighbours_east*, *west*, 2, *neighbours_west*) ▷ y direction.

Algorithm 7.24 Clear_tables: clear up all the MPI tables and possible neighbours.

1: **clear** *north* ▷ Clear MPI four tables.
2: **clear** *south*
3: **clear** *west*
4: **clear** *east*
5: **clear** *neighbours_north* ▷ Clear element possible neighbours tables.
6: **clear** *neighbours_south*
7: **clear** *neighbours_east*
8: **clear** *neighbours_west*

Although our repartitioning algorithm is designed to balance the workload in one shot, in some extreme cases, *i.e.*, the program has very high dynamicity or the load balancing frequency is low, the optimal balance efficiency cannot be reached in one shot. This is because the workload transfer is restricted to neighbour processors: if the situation requires the workload to be distributed to processors beyond the neighbour ones, which cannot be fulfilled, we encounter a

sub-optimal result. To tackle the extreme cases, we evaluate the load balance efficiency after the repartition, if a sub-optimal result is obtained or the result does not fulfill the user defined efficiency threshold, the repartition will be performed recursively, until the desired efficiency is reached. In this way, the repartition efficiency can be guaranteed. The recursive load balancing algorithm is given in Algorithm 7.25 *Load_balancing_recursive*. The input ϵ is the user defined efficiency threshold.

Algorithm 7.25 Load_balancing_recursive: recursive load balancing algorithm. Partition the workload until the user defined load efficiency is reached.

Input: ϵ

Use Algorithms: Algorithm 7.21 Load_balancing

1: *Load_balancing*()
2: **compute the optimal efficiency:** E_{opt}
3: **evaluate the current balance efficiency:** E
4: **while** $E < \epsilon \cdot E_{opt}$ **do**
5: *Load_balancing*()
6: **evaluate the current balance efficiency:** E
7: **end while**

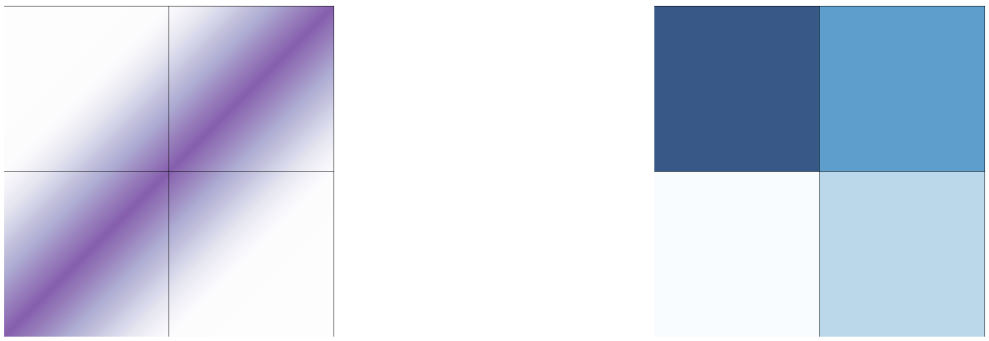
We have presented all the algorithms that contribute to the workload repartitioning process. In summary, the logic of the space-filling curve based repartitioning method is easy to understand and implement. Although unlike graph-based repartitioning algorithms which use global mesh information to optimize the partition so that one can minimize the communication surfaces to some degree, the SFCs-based repartitioning algorithms offer competitive interprocessor boundaries based on the good locality of SFCs. Additionally, since there is no boundary optimization process of the partition, no recursive communications are needed in SFC-based repartitioning algorithms, which contributes to a highly parallelized algorithm. The implementation based on MPI collective operations should deliver portable performance across a broad variety of architectures [31]. With little requirement of global mesh knowledge, the memory overhead incurred by the load balancing algorithm stays low, which is a beneficial feature to today's distributed memory systems since memory consumption could become the bottleneck of an application.

7.8 Dynamic Load Balancing Performance

In this section, we showcase the dynamic load balancing algorithm's performance with the wave equation solver. We revisit the numerical experiment of Section 3.5 and apply the Hilbert curve repartitioning technique to it.

7.8.1 Load Balancing Performance of Experiment One

The load balancing performance of Experiment One of Section 3.5.2 is presented.

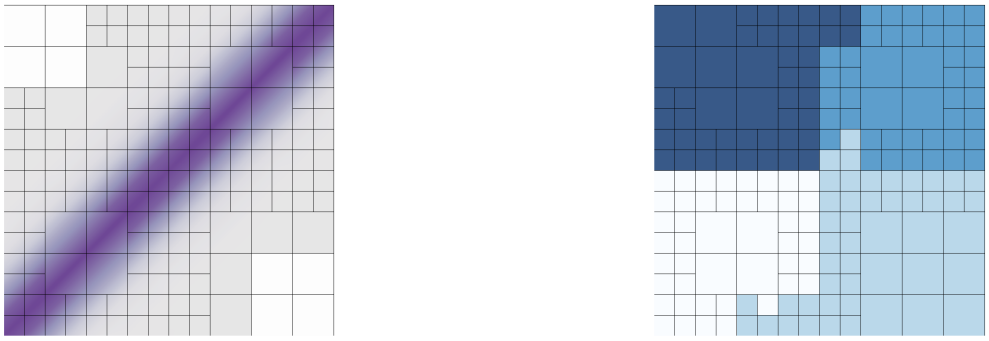


(a) Initial mesh. $K = 4$, $p = 6$.

(b) Initial partition. Four processors ($P = 4$), each one owns one element.

Figure 7.19: Load balancing Experiment One Scene One: Start the simulation with 4 elements ($K = 4$). Time $t = 0$. (a) Configuration of the wave in purple and the polynomial order (p) of each element is indicated by the degree of greyness. (b) Colour scheme represents different processors. Elements in the same colour reside on the same processor.

The simulation starts with four elements, with each element residing on different processors. In total, four processors are invoked in this experiment. The mesh distribution is shown in Fig. 7.19a and the workload distribution in Fig. 7.19b. Since the four elements are identical, we have a well-balanced workload distribution at the beginning of the test.

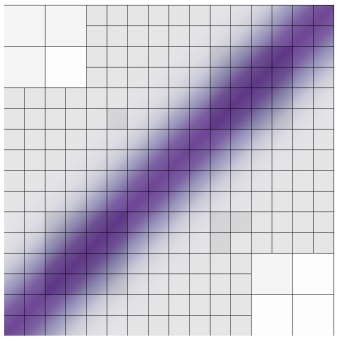


(a) hp-refined mesh at time $t = 6.0 \times 10^{-5}$. $K = 184$, $p = 6 \sim 10$.

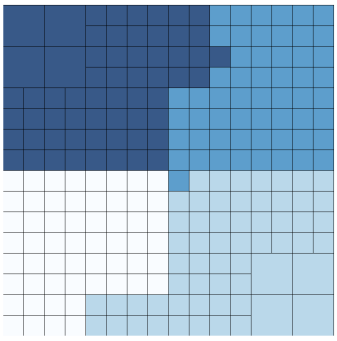
(b) Workload repartition ($P = 4$).

Figure 7.20: Load balancing Experiment One Scene Two: hp-refinements progresses and the workloads are balanced among processors. Time $t = 6.0 \times 10^{-5}$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.19.

The calculation proceeds with time step $\Delta t = 1.0 \times 10^{-8}$; hp-refinement is applied with a tolerance of $\epsilon = 10^{-6}$ (refinement criterion: $\epsilon_{est} \leq \epsilon$) for pressure. At time $t = 6.0 \times 10^{-5}$, hp-refinement is invoked to improve the mesh resolution. Load imbalance is introduced to the application due to the adaptivity. The repartitioning result given by our load balancing algorithm is shown in Fig. 7.20b. The repartition decisions are made based on the trajectory of the Hilbert curve.



(a) hp-refined mesh at time $t = 8.0 \times 10^{-5}$. $K = 232$, $p = 6 \sim 12$.



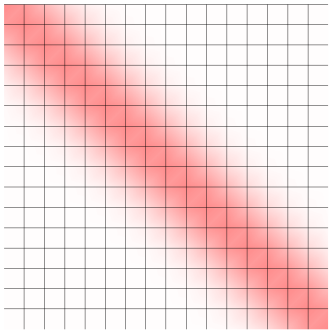
(b) Workloads are redistributed among the processors using Hilbert curve based repartitioning algorithm.

Figure 7.21: Load balancing Experiment One Scene Three: hp-refinement progresses and the workloads are balanced among processors. Time $t = 8.0 \times 10^{-5}$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.19.

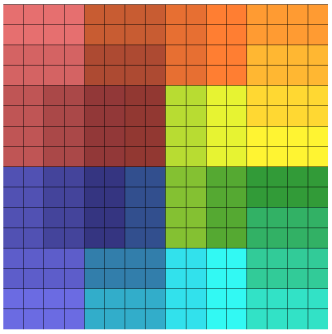
At time $t = 8.0 \times 10^{-5}$, elements with higher polynomial orders are prescribed in the wave region (Fig. 7.21a). Note that the workload has two contributions: element numbers and element polynomial orders. Thus, the processors that own "heavier" elements (elements with high polynomial orders) are assigned less elements (Fig. 7.21b: upper right (the second light blue) and lower left (white)).

7.8.2 Load Balancing Performance of Experiment Two

The load balancing performance of Experiment Two of Section 3.5.3 is presented.



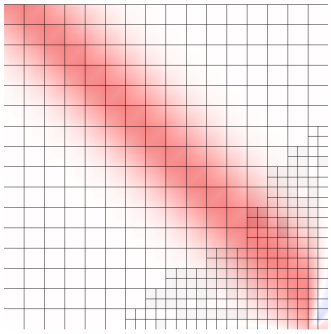
(a) Initial mesh at time $t = 0$. Well-resolved Wave 1 (in red) is initialized. uniform mesh with $K = 256$ elements. $p = 6$.



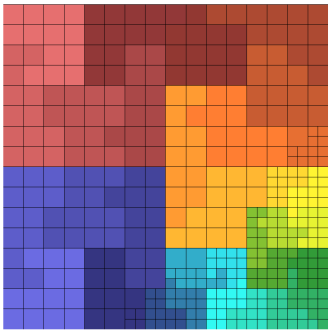
(b) Initial partition. Since elements are identical, each processor is assigned with same number of elements. The load is well-balanced among $P = 32$ processors.

Figure 7.22: Load balancing Experiment Two Scene One: mesh and the workloads among processors at time $t = 0$. (a) Configuration of the velocities in the x direction in red and blue, and the polynomial order p of each element is indicated by the degree of greyness. (b) The colour scheme represents different processors. Elements in the same colour reside on the same process.

The simulation starts with a uniform mesh with uniform polynomial order (Fig. 7.22a). The workload is perfectly partitioned among 32 processors, with each one owning eight elements. The initial partition is shown in Fig. 7.22b. The processors are distinguished by their colours. The partitions have the same workload and boundary sizes.



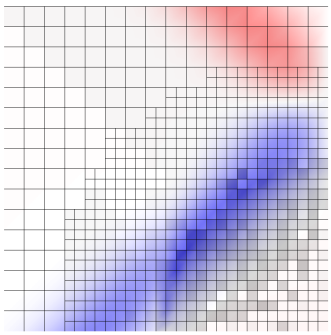
(a) hp-refined mesh at time $t = 0.038$. $K = 406$. $p = 6 \sim 10$.



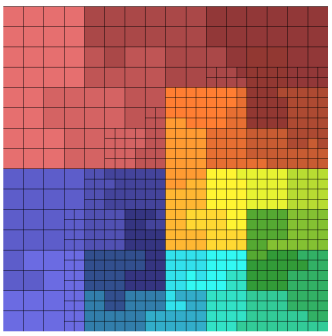
(b) Repartitioning result at time 0.042. $P = 32$.

Figure 7.23: Load balancing Experiment Two Scene Two: hp-refinement progresses and the workloads are balanced among processors. Time $t = 0.038$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.22.

At time 0.038, refinements congregate at the lower right corner of the domain, where the wave is reflected. The workload redistribution result is presented in Fig. 7.23b. One can observe that the processors containing the fine mesh regions own a relatively smaller domain than the processors in the coarse mesh regions since the load is heavier in the fine mesh regions (for example the lightest blue in Fig. 7.24b).



(a) hp-refined mesh at time $t = 0.5$. $K = 697$. $p = 6 \sim 14$.



(b) Repartitioning result at time 0.5. $P = 32$.

Figure 7.24: Load balancing Experiment Two Scene Three: hp-refinement progresses and the workloads are balanced among processors. Time $t = 0.5$. K : number of elements. p : polynomial order. P : number of processors. Same colour scheme as in Fig. 7.22.

At time 0.5, as reflective wave propagates towards the upper-right, high-order elements are placed in its trajectory. Since high-order elements are more computationally expensive, processors in the reflective wave regions are assigned with fewer elements than other processors (Fig. 7.24b), which alleviates the load imbalance incurred by the hp-adaptivity. The repartitioning boundaries are determined by the evolution of the Hilbert curve. The good locality of the Hilbert curve provides a compact repartitioning configuration to each processor.

From these load balancing examples, one can observe how the Hilbert curve facilitates the load balancing algorithm. By splitting the curve into equal workload segments, the new partition results are readily determined, which contributes to a quickly executed algorithm. The compactness of the Hilbert curve guarantees the neighbour elements stay close to each other in the target space. We further evaluate the repartitioning quality and the scalability of the Hilbert curve based load balancing algorithm in the results chapter (Chapter 8).

Chapter 8

Results

We perform large-scale computations to investigate the validity of the hash table AMR, the scalability of the load balancing algorithm and the overall performance of the application. Scalability is important because it indicates the parallel efficiency of the algorithm or program. Two types of scaling tests are introduced in Section 8.1. The TAU profiler (Section 8.2) is invoked to facilitate the evaluation of the time and memory consumption of the load balancing algorithm. The repartitioning qualities are presented in Section 8.6. Finally, the speedup with our load balancing technique is measured in Section 8.7.

8.1 Introduction to scalability

In parallel computing, the scalability of a program is important since it indicates the parallel efficiency of the software. A scalable program means its performance is proportional to the HPC resources that are invoked. Additionally, the scaling results are good references for choosing the amount of computational resources for a specific problem size.

For software, the parallel efficiency can be evaluated as the ratio between the actual speedup and the ideal speedup for a given number of processors. The speedup S in parallel programming is defined as

$$S = \frac{t_1}{t_P}, \tag{8.1}$$

where t_1 is the execution time using one processor, and t_P is the execution time using P processors. Ideally, it is a linear speedup equal to P, which means each processor contributes 100% of its computational power.

Two types of scaling are often used to measure the parallel efficiency of a program: *strong scaling* and *weak scaling*.

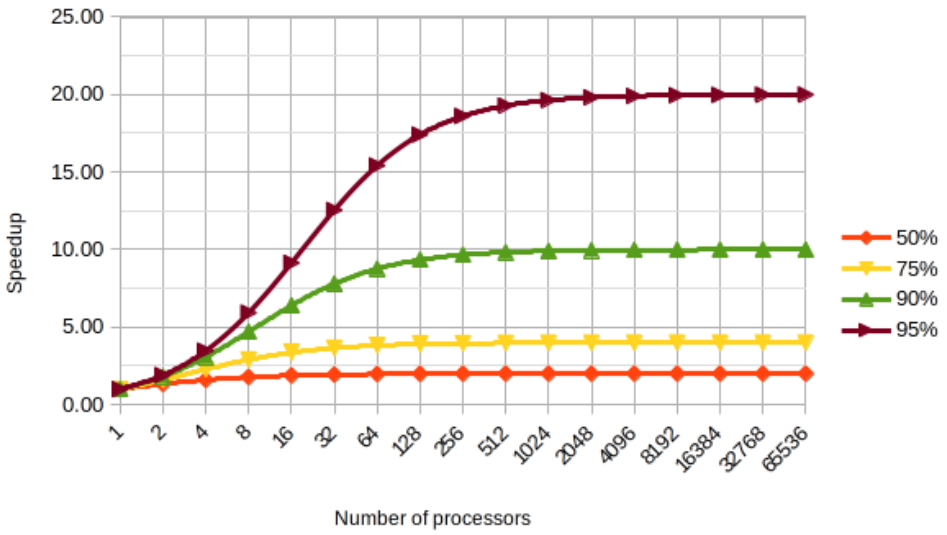


Figure 8.1: Example: theoretical speedup according to Amdahl's law. The upper limit of the parallel speedup is determined by the serial portion of the program. The colour schemes correspond to different parallel portions (parallel portion = 1 − serial portion) of a program.

Strong scaling uses *Amdahl's law*, which gives the theoretical speedup of a fixed size workload problem. Amdahl's law provides the speedup's upper limit as

$$speedup = \frac{1}{s + \frac{p}{P}}. \tag{8.2}$$

s is the serial part of the program, and p is the parallel part. P is the number of processors. The speedup is limited by the serial fraction of the program (Fig. 8.1).

Measuring the speedup with a fixed problem size and increasing the computational resources is called *strong scaling* measurement. The goal of such measurement is to find the optimal computational resources for the targeted problem.

On the other hand, *weak scaling* points out that the size of the problem scales with the amount of resources being utilized. This theory proposed by J. L. Gustafson *et al.* [30], is the so-called *Gustafson's law*. According to Gustafson's law, the suitable amount of computational resources for a specific problem is related to the size of the problem. The speedup of the weak scaling can be computed as

$$scaled\ \ speedup = s + p \times P. \tag{8.3}$$

The variables s, p, P have the same meaning as before. With Gustafson's law, the speedup is proportional (*i.e.*, linear) to the computational resources. There is no limit to the speedup, but the serial portion of the program will affect the speedup's slope (Fig. 8.2).

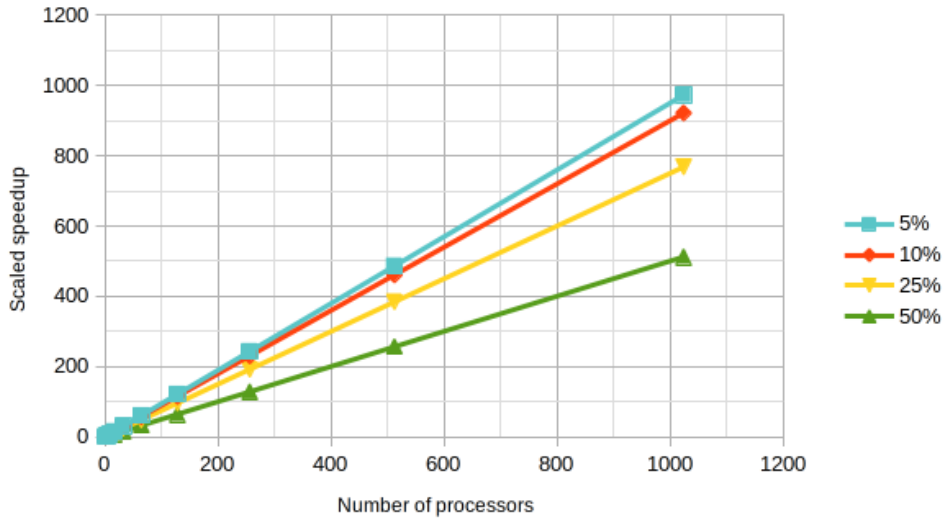


Figure 8.2: Example: theoretical speedup according to Gustafson's law. The speedup has no upper limit, but its slope is affected by the serial portion of the program. The colour schemes correspond to the different serial portions.

Therefore, it is worthwhile to analyze the program's scalability by performing strong scaling and weak scaling tests on the execution time and memory usage of our load balancing algorithm. These are presented in the following sections.

8.2 TAU Profiler

In order to fully evaluate the parallel performance of our application, we seek help from a well-developed parallel profiling library: *TAU*.

TAU[1] (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. TAU is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, Python. One intriguing advantage of TAU is that it provides its own visualization tool, paraprof, which offers us graphical display of all the performance analysis results. This feature can help us to locate any bottleneck of the program efficiently. Due to TAU's versatility, we use TAU to assist our scalability tests.

[1] https://www.cs.uoregon.edu/research/tau/home.php

8.3 Experiment Platforms

The clusters we use to perform our tests are offered by *Compute Canada*[2]. Compute Canada, in partnership with regional organizations ACENET, Calcul Québec, Compute Ontario and WestGrid, leads the acceleration of research and innovation by deploying state-of-the-art advanced research computing (ARC) systems, storage and software solutions. It collaborates with many different partners from across Canada. Our experiments are done on two clusters it offers: Graham and Cedar.

8.3.1 Graham

Graham is a heterogeneous cluster, suitable for a variety of workloads, located at the University of Waterloo. Graham owns 41,548 cores spread across 1,185 nodes of different types. We utilize the node that owns 32 cores with 125Gb available memory per node. Each node is a dual socket Intel E5-2683 v4 Broadwell processor operating at 2.1 GHz clock speed. Graham uses Mellanox FDR (56 Gb/s) and EDR (100 Gb/s) InfiniBand interconnection.

8.3.2 Cedar

Cedar is a heterogeneous cluster suitable for a variety of workloads. It is located at Simon Fraser University. Cedar has 94,528 CPU cores spread among different kinds of nodes. We utilize the node that has 32 cores with 125Gb available memory per node. Each node is a dual socket Intel E5-2683 v4 Broadwell processor operating at 2.1 GHz clock speed. Cedar uses Intel OmniPath (version 1) interconnection (100 Gbit/s bandwidth).

8.4 Execution Time

An efficient load balancing algorithm is essential for a highly dynamic simulation which requires frequent workload balancing. In this section, we measure scaling of the load balancing algorithm's execution time.

8.4.1 Experiment Three

We use the benchmark solution of a Gaussian wave described in Section 2.3.3 for our test cases. The wavevector is chosen as $\mathbf{k} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$. Sound speed $c = 1$. Initial location of the wave: $x_0 = y_0 = 0$.

[2]https://www.computecanada.ca/

Strong Scaling

For the strong scaling test, the Gaussian plane wave propagates from the lower left corner of the square domain of size $[0, 16] \times [0, 16]$ to the upper right corner. The fixed size problem starts with $K = 16,384$ elements. The hp-refinement is guided by the error estimator, with h-refinement level ranging from 0 to 3, and element polynomial order ranging from 6 to 16. For the purpose of this test, we want the mesh to be largely unbalanced, therefore, we apply refinement to the mesh every time step and balance the workload after each refinement. We evaluate the execution time of the load balancing algorithm for 20 time steps and investigate its scalability by its mean performance. $47,782$ elements are used in the refined mesh. The measurement is done on Graham with OpenMPI 3.1.4 and TAU 2.29. The number of processors (P) ranges from 16 to $4,096$.

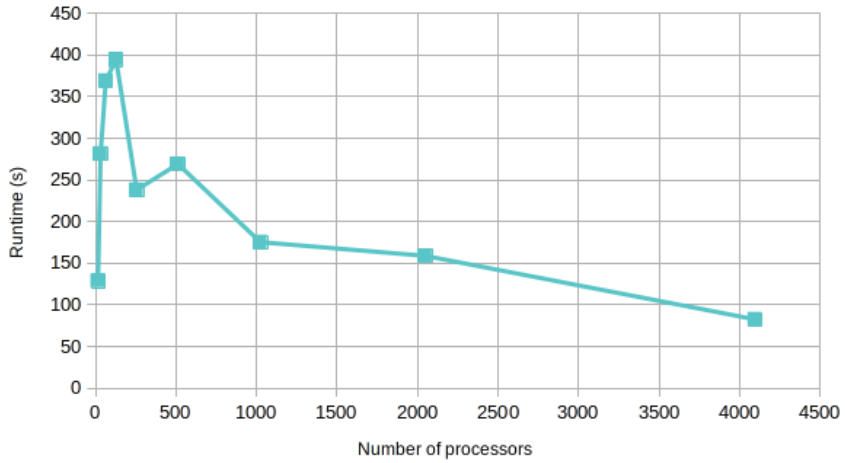


Figure 8.3: Execution time strong scaling of the load balancing algorithm on Graham with $K = 16,384$ elements, initially.

The load balancing algorithm introduces extra overhead to the program, which includes: (1) repartitioning decision-making, (2) updating element interfaces information, including interprocessor interfaces, and (3) re-distribution of workload. Therefore, the efficiency of the algorithm is essential, especially for large scale parallel application. Fig. 8.3 shows the execution time strong scaling of the load balancing algorithm on Graham.

The noticeable trend of the curve in the strong scaling figure is that the execution time increases for a relatively small number of processors before it scales. This is due to the global collective communication operation MPI_Exscan in MPI, which does not scale up linearly, influencing the overall performance of the load balancing algorithm. In Fig. 8.4 we present the scaling comparison for the main time-consuming routines. As one can observe, except for MPI_Exscan and $MPI\_Barrier$ (used in I/O routines) which do not scale linearly, other routines have a good scaling performance. Therefore, we can draw the conclusion that the collective communication

operation in MPI is the bottleneck of the load balancing algorithm.

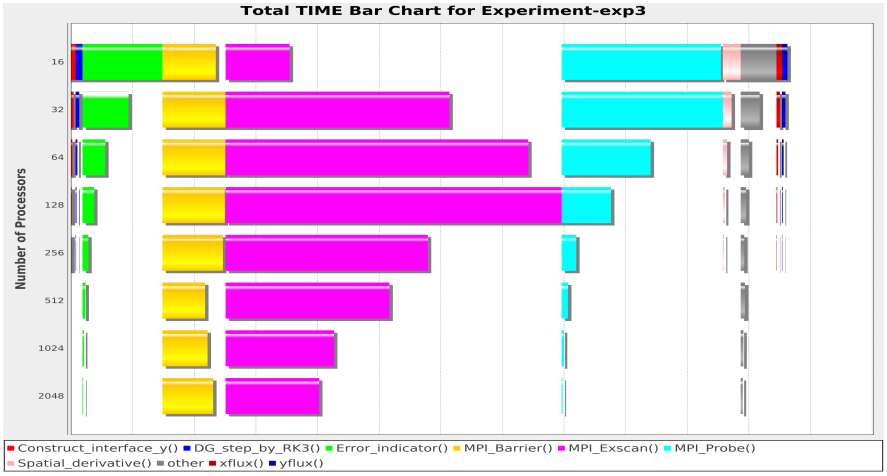


Figure 8.4: Main routines' execution time strong scaling profile. Except for MPI_Exsan (rose bars) and $MPI\_Barrier$ (yellow bars), other routines scale up well, which indicates that MPI_Exscan is the bottleneck of the load balancing algorithm.

Fortunately, the collective communication operation has been proven to have a time complexity of $O(logP)$, where P is the number of processors [64, 70]. Moreover, Fig. 8.4 indicates that MPI_Exscan starts to scale up after the number of processors exceeds 128. This explains the decrease of the execution time of load balancing algorithm in Fig. 8.3. Therefore, our load balancing algorithm can benefit large computing.

Weak Scaling

Since the weak scaling measurement requires the problem size to grow proportionally with the computational resources, we use a manufactured test to investigate the weak scaling of execution time. A static mesh is uniformly assigned random weights to each element. The repartitioning process is performed based on the weights assigned to the elements. In this way, we can scale up the problem size with the computational resources by keeping $K = 128$ elements per core.

The problem sizes and the corresponding computational resources are shown in Table 8.1. The investigation is done on Cedar with OpenMPI 3.1.4 and TAU 2.29.

Domain size	Number of elements	processors	nodes
$[0, 8] \times [0, 8]$	4,096	32	1
$[0, 16] \times [0, 16]$	16,384	128	4
$[0, 32] \times [0, 32]$	65,536	512	16
$[0, 64] \times [0, 64]$	262,144	2048	64
$[0, 128] \times [0, 128]$	1,048,576	8196	256

Table 8.1: Weak scaling problem set up with $K = 128$ elements per core.

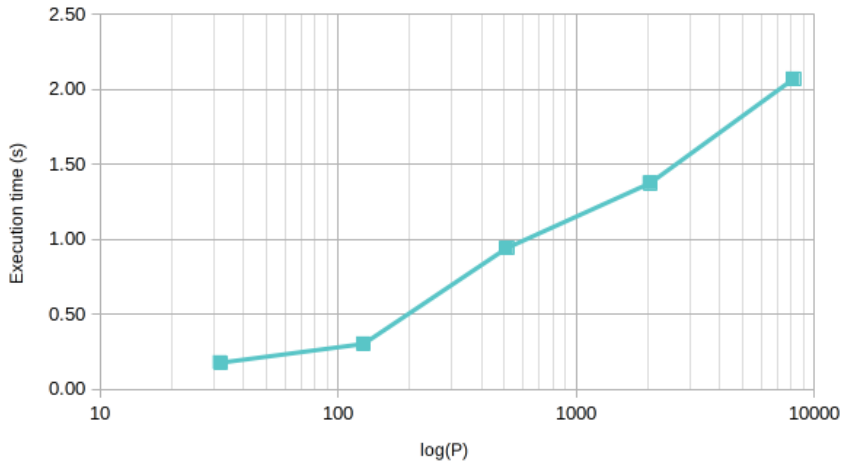


Figure 8.5: Execution time weak scaling on Cedar with $K = 128$ elements per processor (P).

Fig. 8.5 presents the weak scaling of the execution time. As one can see, the overhead of the load balancing algorithm increases with the number of processors. The execution time scales up well in Fig. 8.5: the execution time has an approximately linear relationship with computational resources. Additionally, the growth rate of the execution time is low: the slope of the curve in Fig. 8.5 is 2×10^{-4}, which is nearly negligible. The slow growth of the execution time is favourable for a large scale simulation.

8.4.2 Experiment Four

In Experiment Four, we apply the benchmark solution on a large scale mesh. We investigate the load balancing algorithm time consumption with fixed size problem. Since the weak scaling performance can be covered by the weak scaling test we have presented in Experiment Three, here we only focus on strong scaling.

Strong Scaling

The test case follows a similar approach to Experiment Three in 8.4.1. The wave propagates on the domain of size $[0, 128] \times [0, 128]$, with $K = 1.048576 \times 10^6$ elements and $p = 6$, initially. The maximum h-refinement level is set to be 3 and the polynomial order p ranges from 6 to 16. Refinement/load-balancing is applied every time step. We examine the load balancing routine for 10 time steps and evaluate its average performance. The refinement generates new $31,272$ elements. The measurement is done on Graham with OpenMPI 3.1.4 and TAU 2.29. The number of processors ranges from 256 to $16,384$.

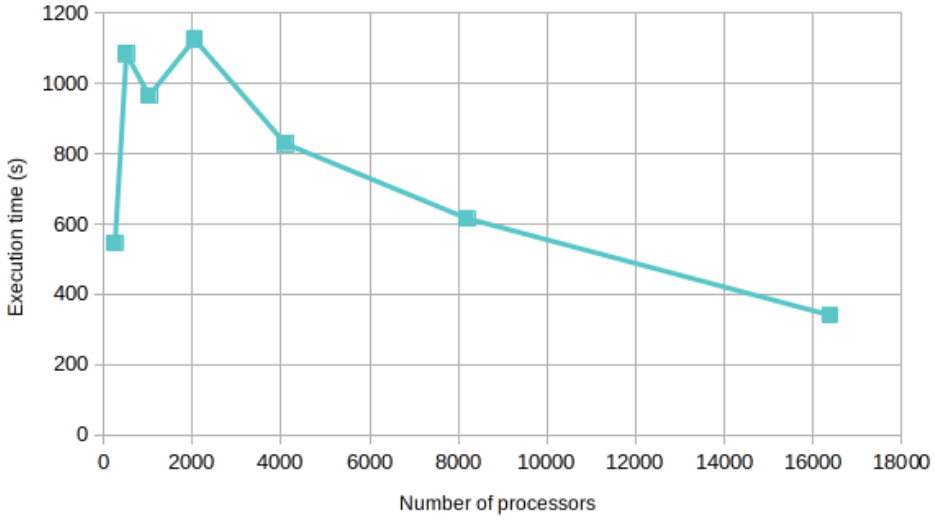


Figure 8.6: Execution time strong scaling on Graham with $K = 1.048576 \times 10^6$ elements, initially.

Fig. 8.6 shows the strong scaling results of execution time. As in Experiment Three, we observe an increment in execution time at the beginning of the curve again caused by the non-linear scaling MPI collective operation ($O(logP)$). However, the increment is temporary and a continuous scaling is observed for up to $16,384$ processors, indicating that the load balancing algorithm has potentially further scalability. Thus, we can draw the conclusion that the load balancing algorithm is suitable for large scale computations.

8.5 Memory Usage

The memory consumption of the repartitioning algorithm is a major concern since distributed systems have limited memory per core. Since the load balancing memory usage is independent of the solver's computation memory usage, the results of the experiments can be treated as references for general cases. The memory consumption of the repartitioning algorithm is offered

by the TAU profiler.

8.5.1 Experiment Three

The experiment is done in the same manner as in Section 8.4.1.

Strong Scaling

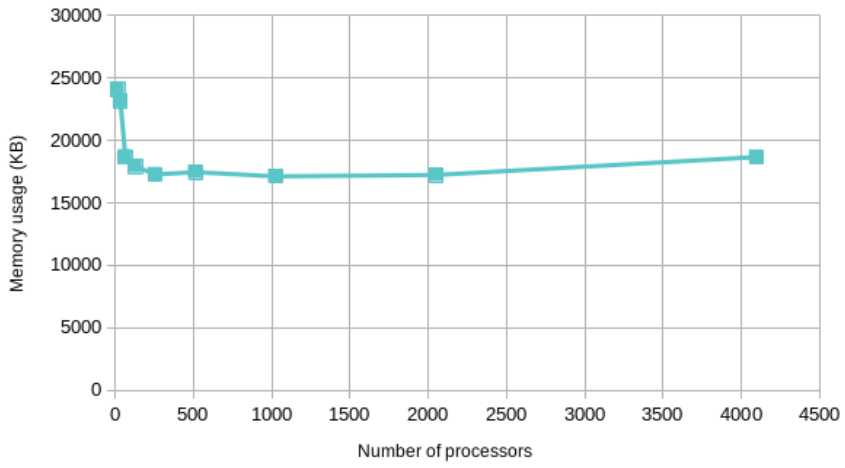


Figure 8.7: Memory consumption strong scaling on Graham with $K = 16,384$ elements, initially.

The strong scaling measurement is performed on Graham with up to $p = 4096$ processors. The result is presented in Fig. 8.7. As one can observe, the memory consumption scales up well with small numbers of processors. The strong scalability is limited by the collective communication operations. However, even though the algorithm stops scaling up after 512 processors, there is no considerable increment of memory usage after that.

Weak Scaling

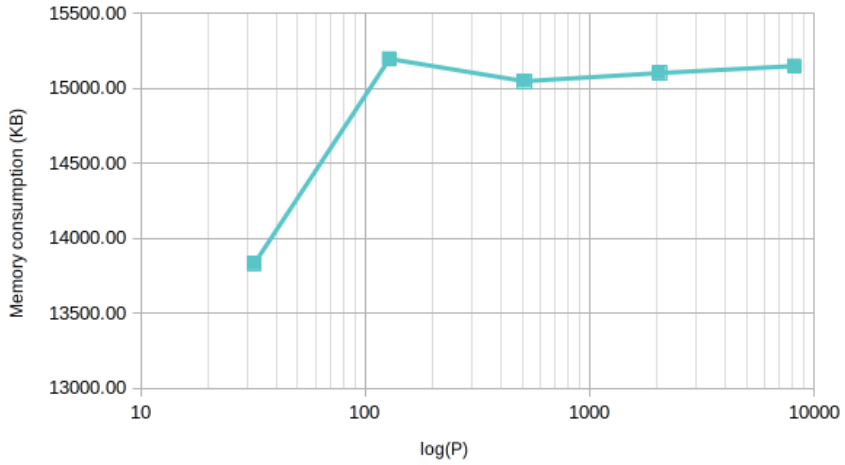


Figure 8.8: Memory usage weak scaling on Cedar with 128 elements per processor. (Note the small range of the memory consumption axis.)

Fig. 8.8 shows the weak scaling result of the memory usage. The weak scaling performs well: there is only a slight increment from 32 processors to 128 processors, then the curve reaches a plateau.

The memory usage strong scaling and weak scaling results both indicate that our load balancing strategy is memory economic. Additionally, for large numbers of processors, the memory consumption is almost independent of the problem size. This indicates that the repartitioning algorithm adds a constant memory overhead to the program. As memory consumption is a major bottleneck of some applications, our memory-friendly repartitioning algorithm would be very practical to implement for those scenarios.

8.5.2 Experiment Four

The experiment is done in the same manner as in the Section 8.4.2.

Strong Scaling

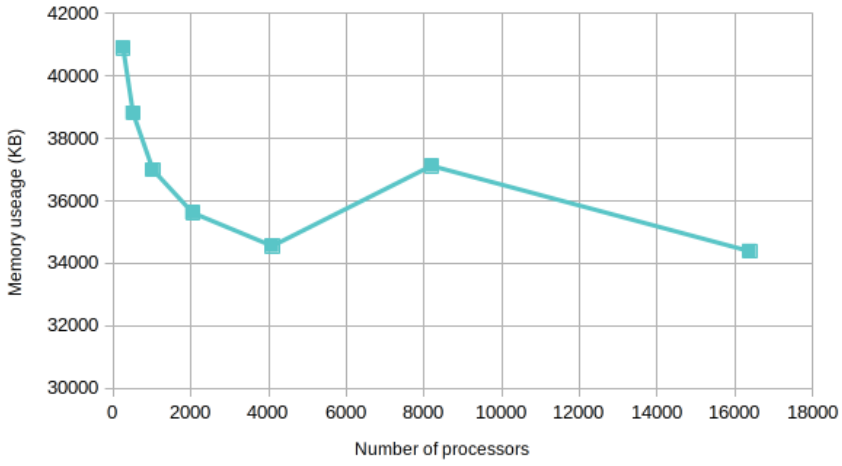


Figure 8.9: Memory consumption strong scaling on Graham with $K = 1.048576 \times 10^6$ elements, initially.

The memory usage with increasing processor number is shown in Fig. 8.9. The number of processors ranges form 256 to 16,384. A stable scaling is obtained up to 4096 processors. Then a small increment occurs: only 7% while doubling the number of processors. From 8,192 to 16,384 processors, no memory increment is observed. Thus, it is safe to say that memory scaling is obtained up to 4096 processors and the memory increment after the scaling limit is minor.

8.6 Load Balancing Quality

In this section we investigate the load balancing quality of our repartitioning strategy.

There are two criteria to evaluate the load balancing quality. The first one is the partition boundary sizes. A smaller interprocessor boundary means less communication. The second one is the workload balance rate, which is the balancing efficiency E defined in Section 7.4.1 Equation 7.5. The ideal situation is that the workloads are balanced evenly among the processors.

For the SFC-based repartitioning algorithm, the first criterion is determined by the natural good locality of the SFCs. Since studies have been done to prove that SFC-based partition provides reasonable communication costs, here we focus our investigation on the second criterion: the workload balance efficiency of the algorithm.

Although our algorithm is designed to balance the workload in one shot, for some highly unbalanced cases or if the repartition process is applied infrequently, the solver cannot obtain the optimal workload balance in one shot. This is due to the constraint that elements can only be transferred to at most two adjacent neighbour processors on the space filling curve.

Extreme cases would require one processor to send elements to more than two processors to achieve optimal balance. To tackle such cases, we evaluate the load efficiency after the repartition: if the efficiency does not reach the user-defined threshold, we apply the repartitioning algorithm recursively until the threshold is reached. This technique can guarantee we always obtain a well-balanced workload regardless of the problem circumstance and the repartitioning frequency.

Recall the balancing efficiency E

$$E = \frac{w_{ideal}}{L_{max}}, \tag{8.4}$$

where w_{ideal} is the ideal workload on a processor, the L_{max} is the maximum local workload among the processors. Thus, the efficiency is limited by the maximum local workload. The imbalance rate is defined by the most heavily loaded processor. Therefore, we use the load efficiency E to evaluate the quality of the repartitioning result.

8.6.1 Experiment Five

We first test the load balancing on a small scale problem. We use the same conditions of Experiment Three as described before. The mesh starts with $K = 16,384$ elements. To maximize the load imbalance, we refine the mesh every time step and record the load balancing efficiency before and after the repartitioning technique is applied. The result is shown in Fig. 8.10. The experiment is done on Graham with $P = 128$ processors.

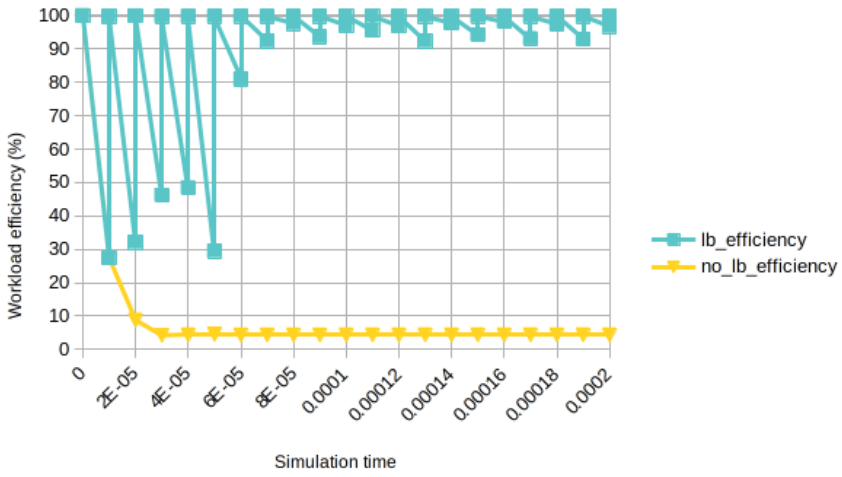


Figure 8.10: Load balancing efficiency on Graham with $K = 16,384$ elements, initially. Green curve represents the results with load balancing (LB) and the yellow curve stands for results without LB.

We compare the load balance imbalance with (green line in Fig. 8.10) and without (yellow line in Fig. 8.10) repartitioning. In the former case, the load balance rates are recorded before

repartitioning/after refinement and after redistribution of the workload. As one can observe from the green line in the figure, the vertical segments on the line with two vertices at the ends of the segments indicate the workload balance efficiencies at these two states. For example, with a fully balanced workload distribution at the beginning, after the first refinement the load efficiency drops to 27.3%, recovering to to 99.8% after the repartitioning process.

From Fig. 8.10, we can see that, at the beginning, the refinement is relatively more severe since the load efficiency drops dramatically. However, with our load balancing algorithm, we successfully balance the workload to the optimal situation, *i.e.*, the balancing efficiencies are close to 100%. Comparing with the workload distribution without dynamic load balancing, the balance rates remain low at 4% due to the highly dynamic property of the problem. Even though the dynamicity of the problem decreases after time 6.0×10^{-5}, the load balance circumstance is largely traumatized for the yellow line because of the high dynamicity at the beginning of the problem. This indicates that our repartitioning algorithm can be very important to the computation time of CFD applications since even a short period of high dynamic change during runtime potentially largely hurts the performance of the solver for a long time afterwards.

8.6.2 Experiment Six

Experiment Six tests the load balancing algorithm on a large scale problem. We use the benchmark solution with $K = 1.048576 \times 10^6$ elements, initially. We still apply refinement every time step to maximize the load imbalance. The investigation is done on Graham with $P = 4096$ processors.

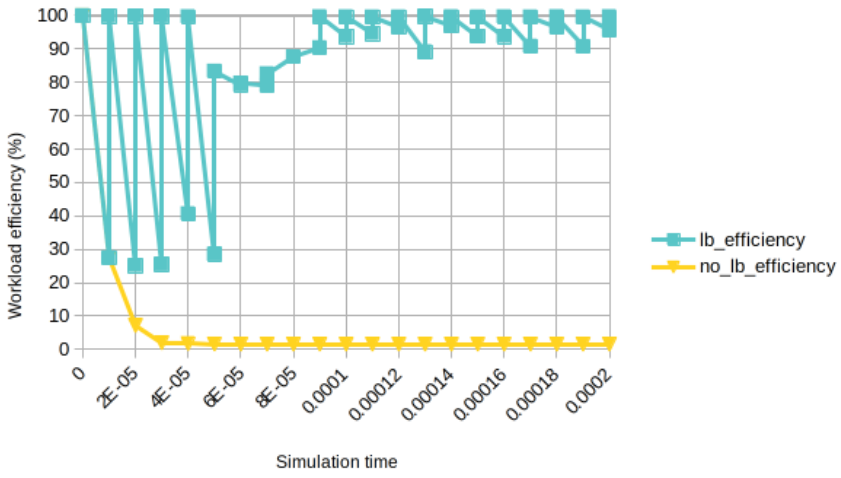


Figure 8.11: Load balancing efficiency on Graham with $K = 1.048576 \times 10^6$ elements, initially. Green curve represent the results with load balancing (LB) and the yellow curve stands for results without.

The experiment is done in the same manner as Experiment Five. We define the efficiency threshold as $80\%E_{opt}$. Thus, during the time period 5.0×10^{-5} to 9.0×10^{-5}, the workload efficiency is sub-optimal (but above 80%). The efficiency of the no load balancing case is again limited by the significant load dynamicity at the beginning: it stalls at 1.5%, which is extremely low.

From the two experiments above, we prove the validity of our repartitioning algorithm. The algorithm is able to tackle highly load imbalance cases, for instance, it can raise the efficiency from 28% back to approximately 100%. A fast executed and memory-friendly repartitioning technique like this is vital to the performance of the application since load imbalance causes fast processors to wait for the slow ones. We quantify the computational saving in time by using our load balancing algorithms in the next section.

8.7 Application Performance

Large scale industrial and scientific simulations often require weeks of computational execution time. With the help of our load balancing algorithms, we want to accelerate the execution and make full use of the valuable computational resources. In this section, we evaluate the overall performance of our application. To investigate the impact of our load balancing algorithm, we first execute the program under the same conditions with and without workload redistribution, then we study the strong scaling performance of the solver.

Firstly, two similar tests have been performed on two clusters: Cedar and Graham with different load imbalance magnitudes. We track the CPU time taken per simulation time step with and without load balancing. The speedup factors of our repartitioning algorithms are shown below. Secondly, the strong scalability of the wave solver is presented to indicate the parallel efficiency of our solver.

8.7.1 Experiment Seven: High Load Imbalance

We test the benchmark solution with $K = 16,384$ elements on Cedar. The number of processors ranges from 32 to $2,048$. The refinement/load-balancing algorithms are applied every 10 time steps for 100 time steps.

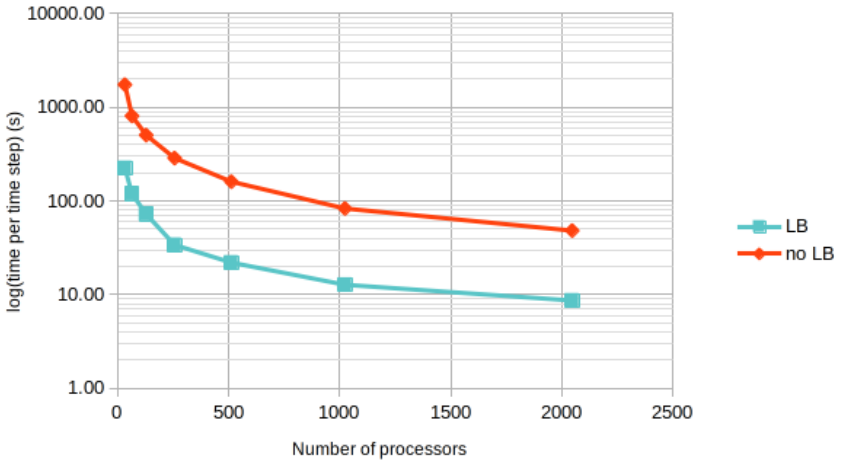


Figure 8.12: Performance comparison: time consumption of each simulation time step with load balancing (LB) (green line) and without LB (red line). A significant speedup can be observed for LB. (On Cedar)

Fig. 8.12 presents the time taken per simulation time step of the program under two circumstances: with load balancing (LB) and without LB. With a relatively high refinement frequency (comparing with the latter experiment in the following section), the load imbalance rate is high among the computational units. Thus, the idle time wasted for faster processors waiting for the slower ones strongly degrades the performance of the program. We can observe a significant speedup after workload repartition is performed.

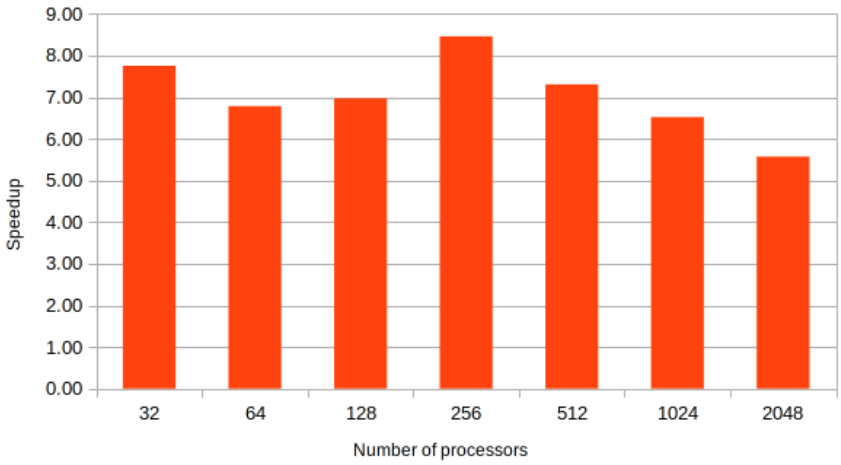


Figure 8.13: Performance comparison: speedup factors (Cedar)

Fig. 8.13 shows the speedup factors under increasing computational resources. We obtain the maximum speedup as 8.46 and the minimum as 5.57. Note that the increasing computational

resources alleviate the load imbalance among the processors: that is why the speedup drops after 256 processors. Therefore, with limited computational resources, our load balancing techniques can increase the computational efficiency. The average speedup is 7.05, which is considerable for a large scale simulation that, often takes weeks to execute, thereby dropping to matter of days.

8.7.2 Experiment Eight: Low Load Imbalance

Another test is achieved on Graham with the same problem but with a lower refinement/repartitioning frequency: every 20 time steps. 200 simulation time steps are performed. The simulations are done with processor number ranging from $P = 64$ to $2,048$.

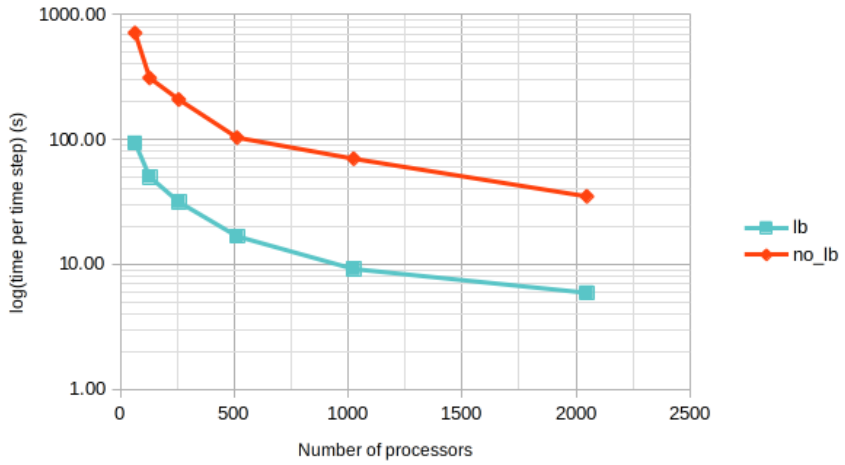


Figure 8.14: Performance comparison: time consumption of each simulation time step with load balancing (LB) (green line) and without LB (red line). After applying LB, a significant speedup can be observed. (On Graham)

Fig. 8.14 presents the time consumption of per simulation time step with (green line) and without load balancing (red line). Considerable speedup can be observed from the figure above.

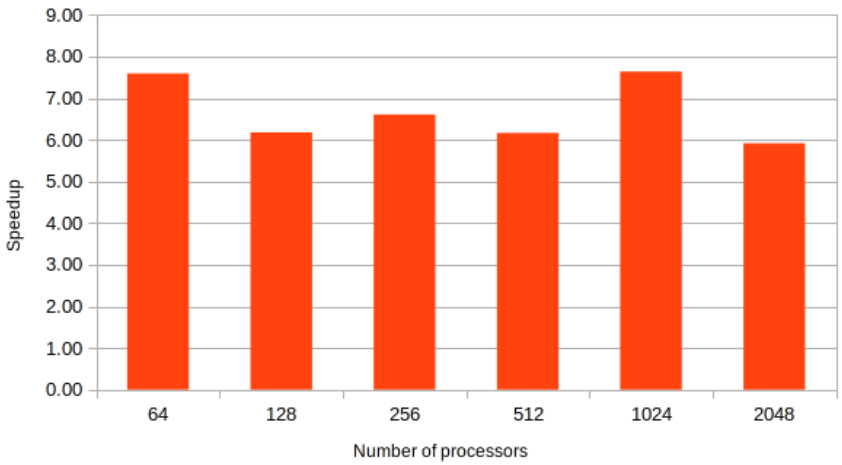
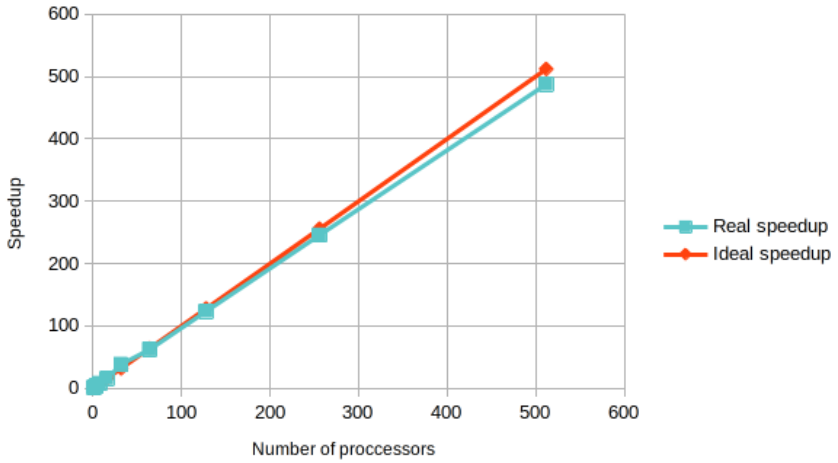


Figure 8.15: Performance comparison: speedup factors (Graham).

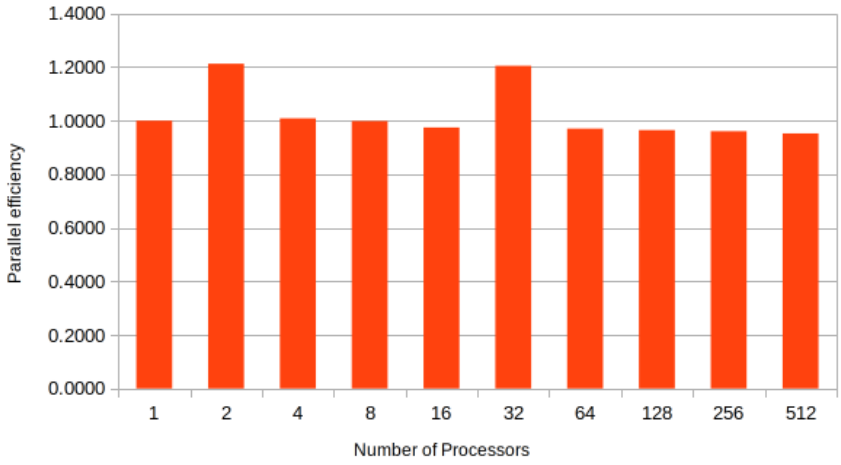
The speedup we obtain in Experiment Eight is shown in Fig. 8.15. Comparing with Experiment Seven, which has a higher refinement frequency, higher load imbalance is incurred in the former case. Therefore, slightly lower speedup factors are gained in this experiment: with a maximum of 7.64, a minimum of 5.91 and an average of 6.68.

8.7.3 Experiment Nine: Application Scalability with Low Refinement Level

In Experiments Seven and Eight, we demonstrate that load balancing can largely benefit a dynamic application. In this and the next sections, we showcase the scalability of our wave equation solver. A benchmark solution with initially 1024 elements is tested on Cedar. The number of processors ranges from 1 to 512. The h-refinement level ranges from 0 to 2 and the polynomial orders from 6 to 14. The refinement is guided by the error estimator. 3856 elements are formed during the simulation. With this set up, we can study the strong scaling of the AMR solver and the parallel efficiency of our load balancing algorithm.



(a) Speedup.



(b) Parallel efficiency.

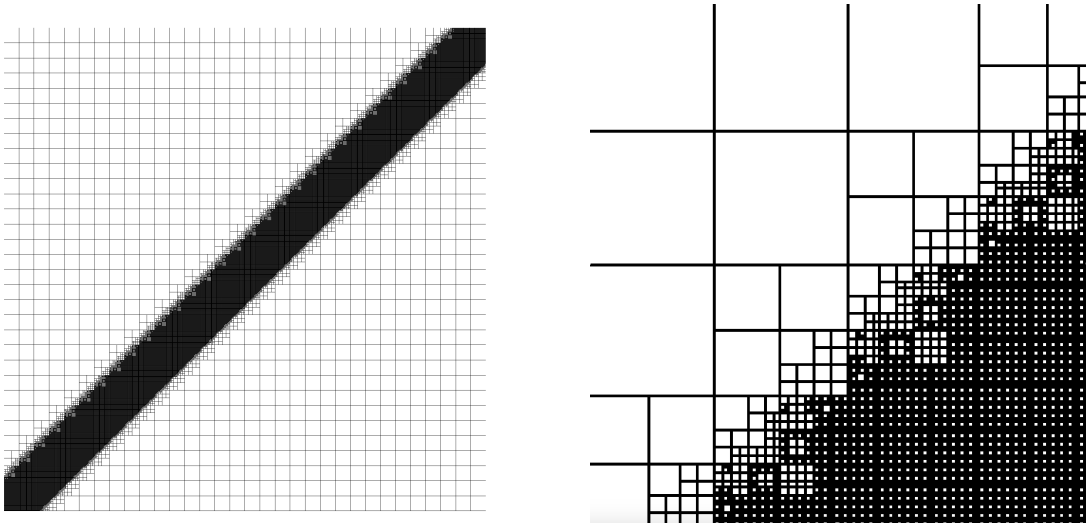
Figure 8.16: Strong scalability indication (maximum h-refinement level = 2).

As we can see in Fig. 8.16a, we obtain an approximately ideal speedup, which implies that the processors are contributing almost their full strength with the help of our load balancing algorithm. From 1 to 512 processors, the wave solver's parallel efficiency ranges from 95% to 120% (Fig. 8.16b), which is highly efficient.

8.7.4 Experiment Ten: Application Scalability with High Refinement Level

In this section, we increase the level of adaptivity to a maximum h-refinement level of 5. Other test conditions remain the same as in Experiment Nine. We investigate the strong scaling of

the solver with the number of processors ranging from 1 to 512. After 10 rounds of refinement, $154,486$ elements are formed. We use this test to examine the impact of the refinement level to the load balancing efficiency. The mesh is presented in Fig. 8.17.



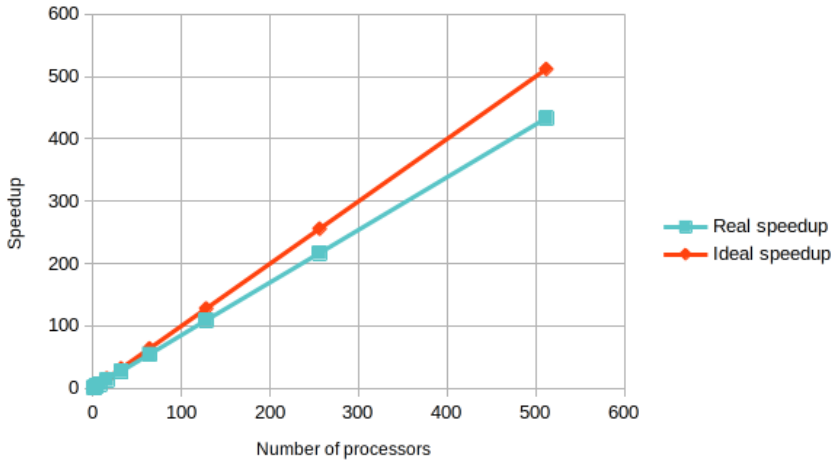
(a) Mesh at time $t = 0.001$.

(b) Zoomed in mesh.

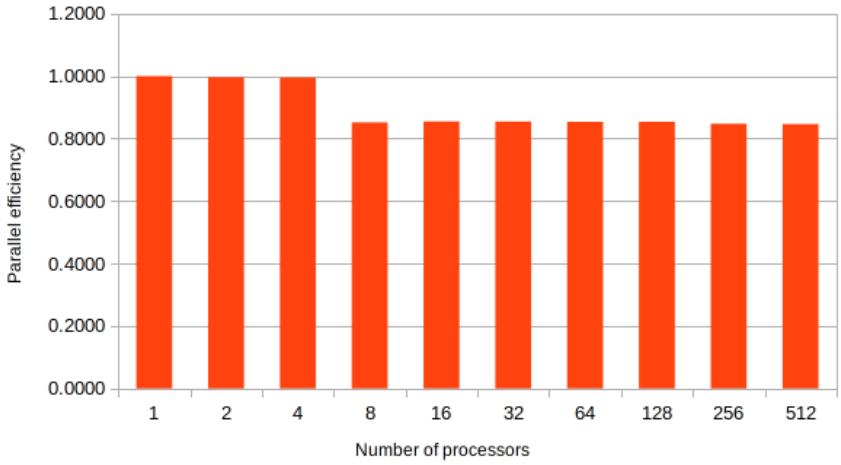
Figure 8.17: Strong scalability experiment: resolve a wave with h-refinement level of 5.

In Fig. 8.18a, we see a slightly less linear profile, but the speedup is still good: ranging from 84% to 99% (Fig. 8.18b). The performance loss is mostly due to load imbalance as we use high h-refinement level here. Meister *et al.* [54] obtained a parallel efficiency of 88% scaling from 16 to 512 cores in their AMR application with dynamic load balancing, which is similar to our result.

With the examples above, we demonstrate that our load balance algorithm is efficient and potent.



(a) Speedup.



(b) Parallel efficiency.

Figure 8.18: Strong scalability indication (maximum h-refinement level = 5).

From the experiments above, we can draw the conclusion that load imbalance is one of the main obstacles of high level parallelism, and the performance of applications that have high dynamicity can be largely diminished by it. Our dynamic load balancing technique is a potent method to alleviate or even eliminate the workload imbalance problem, providing significant savings of computational resources and user wait time.

Chapter 9

Conclusions

9.1 Conclusions

A cost-efficient wave equation solver with the ability to analyze complex fluid motion was constructed using the discontinuous Galerkin spectral element method and parallel adaptive mesh refinement. Two major achievements are presented in this work: (1) a new parallel adaptive mesh refinement and coarsening technique and (2) an efficient space-filling curve based dynamic load balancing algorithm.

High-order methods are crucial to support complex flow simulations. However, a large-scale, time-dependent simulation can be very computationally expensive. To obtain storage savings against a static fine mesh and still preserve the required accuracy, parallel adaptive mesh refinement strategies are implemented in this work, namely hp-refinement techniques. hp-adaptivity incurs discretization discontinuities between element interfaces, resulting in geometrically or/and functionally non-conforming interfaces. The mortar element method assures the exponential convergence of the program, as shown with our wave benchmark solution in Section 4.2.3.

With the help of the mortar element method, the geometric and functional restrictions on conforming interfaces are eliminated, which increases the flexibility of the AMR approaches. The AMR starts with a coarse mesh and high-resolution meshes are developed using an *a posteriori* error estimator to identify the computationally difficult regions. With this technique, fine meshes are used more strategically and cost-efficiently.

A desired AMR algorithm also requires efficient data operations. To achieve this goal, we resort to advanced data structures. Instead of adopting popular block-structured or tree-structured AMR, we proposed a hash table AMR technique. A hash table structure supports constant time complexity of the common data operations, such as insertion, search, deletion, which is highly beneficial to an application with high refinement frequency. The array-like data storage

approach of the hash table simplifies mesh hierarchies. All the data is maintained and managed in a linear structure manner. This feature contributes to an easy to comprehend and implement AMR method in comparison with tree-structured AMR, which utilizes a considerable number of pointers to build up the data hierarchies. Each element in the domain is granted a unique index which can be used to identify its refinement level and support the neighbour elements' queries. Many grid-based CFD applications require neighbour element access. With the knowledge of a neighbour element's unique index, the search operation in the hash table is a constant time operation ($O(1)$). This built-in character of the hash table is hard to find in other advanced data structures. In summary, the hash table AMR we offer is a suitable and preferable alternative to other AMR techniques. The implementation is presented in Section 5.2.

In Chapter 7, a dynamic load balancing algorithm was explained and presented. In our work, the space-filling curve – the Hilbert curve, which has the best locality, is chosen to reduce the multidimensional repartitioning problem into a one-dimensional one, while minimizing interprocessor boundaries. Thus, the theoretically NP-hard problem becomes a linear chains-on-chains partitioning problem. We alleviate the cost of the generation of the Hilbert curve by a table-driven algorithm (Section 7.3). The Hilbert curve can be placed inside the hash table with a linked-list. With these ingredients, a simple and efficient load balancing algorithm was presented in Section 7.7.

The repartitioning algorithm was implemented based on MPI operations, since they are portable across a wide range of computational architectures. The performance of our load balancing algorithm was investigated on two different platforms: Compute Canada's Graham and Cedar. The overhead and scalability of our load balancing algorithm was studied in terms of time consumption and memory usage. The wave equation solver was utilized as a benchmark solution. In the strong scaling execution time test, MPI collective operations were identified as the bottleneck of the repartitioning algorithm. However, the collective operations have the time complexity of $O(logP)$, which favours a large-scale computation. As for memory consumption of the algorithm, we obtain a good strong scaling for up to 4096 processors (Section 8.5.2). Moreover, the memory usage increment is not considerable after passing the optimal scaling point. Low-memory overhead was observed with the weak scaling test. This indicates that our load balancing algorithm is memory-friendly to the distributed memory systems which dominate most of the HPCs nowadays. Additionally, the quality of the repartitioning algorithm was demonstrated in Section 8.6. The good locality of the Hilbert curve ensures a reasonable communication cost. One of the major advantages of SFC-based repartitioning algorithms is that they provide highly-balanced repartitioning results. For highly dynamic applications, our fast-execution and high-quality load balancing algorithm will support frequent repartitioning, while still improving the performance of the application. Last but not least, we evaluated the solver's performance improvement with respect to the execution time using the load balancing technique. Considerable speedups were obtained with a broad range of computational cores. The maximum speedup factor we obtained was 8.46. Considering a large-scale engineering

or scientific simulation that executes for weeks, our load balancing algorithm can significantly accelerate the calculation and save valuable computational resources.

9.2 Future Work

We plan to expand the 2D DG-SEM acoustic wave equation solver into a 3D incompressible Navier-Stokes solver. While the discontinuous Galerkin spectral element method was initially developed for approximating hyperbolic systems of conservation laws, this method has already been extended to convection-diffusion equations [23] and elliptic problems [5]. Recently, the DG-SEM method has been applied to the incompressible Navier-Stokes equations [20]. In the incompressible Navier-Stokes equations, we have nonlinear convective terms and linear diffusive terms. Simple explicit Runge-Kutta methods impose severe time step restrictions on the diffusive term. An implicit time stepping scheme can alleviate the time step restriction but introduces additional computational costs to the nonlinear operator. A compromise is reached by choosing a high-order multi-stage implicit explicit (IMEX) Runge-Kutta time integration scheme [6] to avoid strict time step restrictions on the diffusion term while being computationally economic for the nonlinear convective term. Implicit schemes allow larger time steps in the solution smooth regions in the domain, *i.e.*, time adaptivity can be applied to reduce the computational cost. Additionally, the mortar element discretization has been extended into 3D [26]. The error estimator, as we explained in this work, is straightforward to formulate in 3D.

Since the hash table AMR and the load balancing algorithm are independent of the numerical computation, these two approaches stay intact when we change the differential equations. However, in 3D, four integers are needed to form the coordinate that represents an element, by adding an extra number to our pairing function. The value of the bijection grows large fast, even in 2D. We therefore need to prescribe a 64 bit integer to store the value of the bijection if we have millions of elements. One possible solution is to convert the bijection value to a string and use the string as the key of the hash table. As for establishing the Hilbert curve in 3D, although it was originally developed on a 2D square, now, it is applied in 3D [57, 68] and complex domain shapes [3]. The table-driven Hilbert curve generation algorithm also applies for 3D [17]. Thus, our load balancing algorithm can deal with complex geometries in 3D.

Once implemented for the Navier-Stokes equations, it will be pertinent to compare load balancing and parallel performance on the test problems of [20] to show improvement. The new algorithm will also enable a wider range of more complex problems to be tackled since the full potential of parallel platforms will be exploited.

Another possibility for future work is to integrate our hash table AMR and dynamic load balancing algorithm to construct a software library that could be used by existing solvers not only for fluid ones, but also for solid mechanics that have dynamic workload and using Cartesian grids. Moreover, we would like to compare our library's load balancing performance with existing

ones, such as *p4est*.

Bibliography

[1] D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.

[2] J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis. CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences. Technical Report NASA/CR-2014-218178, NASA, 2014.

[3] S. Aluru and F. E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings Fourth International Conference on High-Performance Computing*, pages 230–235. IEEE Computer Society, 1997. doi:10.1109/HIPC.1997.634498.

[4] G. Anagnostou, Y. Maday, C. Mavriplis, and A. T. Patera. On the mortar element method: generalization and implementation. In T. Chan, R. Glowinski, J. Périaux, and O. Widlund, editors, *Proceedings of the 3rd International Conference on Domain Decomposition Methods for Partial Differential Equations*, pages 157–173. SIAM, Philadelphia, 1989.

[5] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, 2002.

[6] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2):151 – 167, 1997.

[7] S. Bayyuk, K. Powell, and B. van Leer. A simulation technique for 2-D unsteady inviscid flows around arbitrarily moving and deforming bodies of arbitrary geometry. In *11th Computational Fluid Dynamics Conference, AIAA Paper, 1993-3391*, 1993. doi:10.2514/6.1993-3391.

[8] J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 15(1):127–138, 1994.

[9] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64 – 84, 1989.

[10] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[11] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In F. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures*, pages 188–199. Springer, Berlin, 1993.

[12] C. Bernardi, Y. Maday, and A. Patera. A new nonconforming approach to domain decomposition : The mortar element method. *Nonlinear Partial Differential Equations and Their Applications*, 384:13–51, 1994.

[13] C. E. Bichot and P. Siarry. *Graph Partitioning*. ISTE, London, 2011.

[14] T. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42:153–159, 1992.

[15] C. Burstedde. Parallel tree algorithms for AMR and non-standard data access. *Computing Research Repository (CoRR) on arXiv, Cornell University*, abs/1803.08432, 2018.

[16] C. Burstedde, L. Wilcox, and O. Ghattas. p4est : Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33:1103–1133, 2011.

[17] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[18] C. Canuto. *Spectral Methods Evolution to Complex Geometries and Applications to Fluid Dynamics*. Springer, Berlin, 2007.

[19] T. Cebeci, J. P. Shao, F. Kafyeke, and E. Laurendeau. *Computational Fluid Dynamics for Engineers*. Springer, Berlin, 2006.

[20] N. Chalmers, G. Agbaglah, M. Chrust, and C. Mavriplis. A parallel hp-adaptive high order discontinuous Galerkin method for the incompressible Navier-Stokes equations. *Journal of Computational Physics: X*, 2:100023, 2019.

[21] D. R. Chapman. Computational aerodynamics development and outlook. *AIAA Journal*, 17(12):1293–1313, 1979.

[22] E. Charlton and K. Powell. An octree solution to conservation laws over arbitrary regions (OSCAR). In *35th Aerospace Sciences Meeting and Exhibit, AIAA Paper 97-0198*, 1997.

[23] B. Cockburn and C.-W. Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[24] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. van Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. *Journal of Physics: Conference Series*, 78:012013, 2007.

[25] H. B. Enderton. *Elements of Set Theory*. Academic Press, San Diego, 1977.

[26] H. Feng, C. Mavriplis, R. Feng, and R. Biswas. Parallel 3D mortar element method for adaptive nonconforming meshes. *Journal on Scientific Computing*, 27(1–3):231–243, 2006.

[27] L. M. Goldschlager. Short algorithms for space-filling curves. *Software: Practice and Experience*, 11(1):99–99, 1981.

[28] J. Griffiths. Table-driven algorithms for generating space-filling curves. *Computer-Aided Design*, 17(1):37 – 41, 1985.

[29] V. Guinot. The Riemann problem. In *Wave Propagation in Fluids*, pages 161–191. John Wiley & Sons, Hoboken, 2013.

[30] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[31] D. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 1661–1669. IEEE, 05 2012. doi:10.1109/IPDPSW.2012.207.

[32] R. Henderson. Dynamic refinement algorithms for spectral element methods. *Computer Methods in Applied Mechanics and Engineering*, 175(3):395 – 411, 1999.

[33] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical Report SAND-93-0074 ON: DE93008362, Sandia National Labs., Albuquerque, NM (United States), 1993.

[34] D. Holzmüller. *Efficient Neighbor-Finding on Space-Filling Curves*. Bachelor's thesis, University of Stuttgart, 2017.

[35] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[36] H. V. Jagadish. Linear clustering of objects with multiple attributes. *AMC SIGMOD Record*, 19(2):332–342, 1990.

[37] H. Ji, F.-S. Lien, and E. Yee. A new adaptive mesh refinement data structure with an application to detonation. *Journal of Computational Physics*, 229:8981–8993, 2010.

[38] G. Karniadakis and S. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, Oxford, 2005.

[39] A. Khokhlov. Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations. *Journal of Computational Physics*, 143(2):519 – 543, 1998.

[40] D. Kopriva. *Implementing Spectral Methods for Partial Differential Equations*. Springer, Dordrecht, 2009.

[41] D. Kopriva, S. Woodruff, and M. Hussaini. Computation of electromagnetic scattering with a non-conforming discontinuous spectral element method. *International Journal for Numerical Methods in Engineering*, 53:105–122, 2002.

[42] D. A. Kopriva. A conservative staggered-grid Chebyshev multidomain method for compressible flows. II. A semi-structured method. *Journal of Computational Physics*, 128(2):475 – 488, 1996.

[43] H. Kreiss, J. Oliger, and Global Atmospheric Research Programme. Joint Organizing Committee and International Council of Scientific Unions and World Meteorological Organization. *Methods for the Approximate Solution of Time Dependent Problems*. International Council of Scientific Unions, World Meteorological Organization, 1973.

[44] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger. Optimization of all-to-all communication on the Blue Gene/L supercomputer. In *2008 37th International Conference on Parallel Processing*, pages 320–329. IEEE, 2008. doi:10.1109/ICPP.2008.83.

[45] D. Lasalle and G. Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236. IEEE, 2013. doi:10.1109/IPDPS.2013.50.

[46] M. Lieber and W. Nagel. Highly scalable SFC-based dynamic load balancing and its application to atmospheric modeling. *Future Generation Computer Systems*, 82:575–590, 2018.

[47] H. Liu, T. Cui, W. Leng, and L. Zhang. Encoding and decoding algorithms for arbitrary dimensional Hilbert order. *ArXiv*, abs/1601.01274, 2016.

[48] R. Löhner. Adaptive remeshing for transient problems. *Computer Methods in Applied Mechanics and Engineering*, 75(1):195 – 214, 1989.

[49] R. Löhner. Some useful renumbering strategies for unstructured grids. *International Journal for Numerical Methods in Engineering*, 36(19):3259–3270, 1993.

[50] R. Löhner and P. Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10):1135–1149, 1988.

[51] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330 – 354, 2000.

[52] C. Mavriplis. *Nonconforming Discretizations and A Posteriori Error Estimators for Adaptive Spectral Element Techniques*. PhD thesis, Massachusetts Institute of Technology, 1989.

[53] C. Mavriplis. Adaptive mesh strategies for the spectral element method. *Computer Methods in Applied Mechanics and Engineering*, 116(1):77 – 86, 1994.

[54] O. Meister, K. Rahnema, and M. Bader. Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells. *ACM Transactions on Mathematical Software*, 43(3), 2016.

[55] S. Miguet and J.-M. Pierson. Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing. In B. Hertzberger and P. Sloot, editors, *HPCN Europe*, volume 1225, pages 550–564. Springer, Berlin, 1997.

[56] B. Monien and S. Schamberger. Graph partitioning with the party library: helpful-sets in practice. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 198–205. IEEE, 2004. doi:10.1109/SBAC-PAD.2004.18.

[57] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[58] N. Offermans, A. Peplinski, O. Marin, and P. Schlatter. Adaptive mesh refinement for steady flows in Nek5000. *Computers & Fluids*, 197:104352, 2020.

[59] C.-W. Ou and S. Ranka. Parallel remapping of adaptive problems. *Journal of Parallel and Distributed Computing*, 42(2):109 – 121, 1997.

[60] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking*, pages 493–498. Springer, Berlin, 1996.

[61] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, 1996.

[62] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974 – 996, 2004.

[63] R. H. Pletcher, J. C. Tannehill, and D. Anderson. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, Washington, DC, 1997.

[64] R. Rabenseifner. Optimization of collective reduction operations. In M. Bubak, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9. Springer, Berlin, 2004.

[65] B. Radi and A. El Hami. Polynomial interpolation. In *Advanced Numerical Methods with Matlab® 1*, pages 35–65. John Wiley & Sons, Hoboken, 2018.

[66] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[67] L. Rosenberg and H. R. Strong. Addressing arrays by shells. *IBM Technical Disclosure Bulletin*, 14(10):3026–3028, 1972.

[68] H. Sagan. *Space-Filling Curves*. Springer, New York, 1994.

[69] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Boston, 1990.

[70] P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for MPI. In B. Mohr, J. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 49–57. Springer, Berlin, 2006.

[71] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14:219–240, 2002.

[72] H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

[73] M. Szudzik. The Rosenberg-strong pairing function. *arXiv.org*, 1706.04129, 2019.

[74] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics A Practical Introduction*. Springer, Berlin, 1999.

[75] L. N. Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*. unpublished text, 1996. available at `http://people.maths.ox.ac.uk/trefethen/pdetext.html`, acessed on Oct. 1st, 2020.

[76] T. Tu and D. R. O'Hallaron. Extracting hexahedral mesh structures from balanced linear octrees. In *Proceedings, 13th International Meshing Roundtable*, pages 191–200. Sandia National Laboratories, Williamsburg, 2004.

[77] T. Tu, D. R. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 4–4. IEEE, 2005. doi:`10.1109/SC.2005.61`.

[78] R. Vinuesa, S. Hosseini, A. Hanifi, D. Henningson, and P. Schlatter. Direct numerical simulation of the flow around a wing section using high-order parallel spectral methods. *9th International Symposium on Turbulence and Shear Flow Phenomena (TSFP-9)*, 2015.

[79] J. Williamson. Low-storage Runge-Kutta schemes. *Journal of Computational Physics*, 35:48–56, 1980.

[80] I. H. Witten and B. Wyvill. On the generation and use of space-filling curves. *Software: Practice and Experience*, 13(6):519–525, 1983.

[81] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, and J. E. Bussoletti. A locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics. *Journal of Computational Physics*, 92(1):1 – 66, 1991.

[82] K. Zhai, T. Banerjee, D. Zwick, J. Hackl, and S. Ranka. Dynamic load balancing for compressible multiphase turbulence. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 318–327. Association for Computing Machinery, New York, 2018.

[83] O. C. Zienkiewicz. *The Finite Element Method*. Elsevier/Butterworth-Heinemann, Oxford, 1977.