A Graphics Processing Unit Based Discontinuous Galerkin Wave Equation Solver with hp-Adaptivity and Load Balancing

Guillaume Tousignant

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

© Guillaume Tousignant, Ottawa, Canada, 2022

A Graphics Processing Unit Based Discontinuous Galerkin Wave Equation Solver with hp-Adaptivity and Load Balancing

by

Guillaume Tousignant

Abstract

In computational fluid dynamics, we often need to solve complex problems with high precision and efficiency. We propose a three-pronged approach to attain this goal. First, we use the *discontinuous Galerkin spectral element method (DG-SEM)* for its high accuracy. Second, we use *graphics processing units (GPUs)* to perform our computations to exploit available parallel computing power. Third, we implement a parallel *adaptive mesh refinement (AMR)* algorithm to efficiently use our computing power where it is most needed. We present a GPU DG-SEM solver with AMR and dynamic load balancing for the 2D wave equation.

The DG-SEM is a higher-order method that splits a domain into elements and represents the solution within these elements as a truncated series of orthogonal polynomials. This approach combines the geometric flexibility of finite-element methods with the exponential convergence of spectral methods.

GPUs provide a massively parallel architecture, achieving a higher throughput than traditional CPUs. They are relatively new as a platform in the scientific community, therefore most algorithms need to be adapted to that new architecture. We perform most of our computations in parallel on multiple GPUs.

AMR selectively refines elements in the domain where the error is estimated to be higher than a prescribed tolerance, via two mechanisms: *p-refinement* increases the polynomial order within elements, and *h-refinement* splits elements into several smaller ones. This provides a higher accuracy in important flow regions and increases capabilities of modeling complex flows, while saving computing power in other parts of the domain. We use the *mortar element method* to retain the exponential convergence of high-order methods at the non-conforming interfaces created by AMR.

We implement a parallel *dynamic load balancing* algorithm to even out the load imbalance caused by solving problems in parallel over multiple GPUs with AMR. We implement a *space-filling curve*-based repartitioning algorithm which ensures good locality and small interfaces.

While the intense calculations of the high order approach suit the GPU architecture, programming of the highly dynamic adaptive algorithm on GPUs is the most challenging aspect of this work. The resulting solver is tested on up to 64 GPUs on HPC platforms, where it shows good strong and weak scaling characteristics. Several example problems of increasing complexity are performed, showing a reduction in computation time of up to 3× on GPUs vs CPUs, depending on the loading of the GPUs and other user-defined choices of parameters. AMR is shown to improve computation times by an order of magnitude or more.

Keywords: Spectral element methods, discontinuous Galerkin, graphics processing units, adaptive mesh refinement, space-filling curves, Hilbert curve, dynamic load balancing, high performance computing.

Acknowledgements

I would like to thank my supervisor, Prof. Catherine Mavriplis. Her boundless support and patience helped me when I felt stuck, from tiny bugs to bigger setbacks. Her insight and knowledge helped me immensely approach new problems in ingenious ways. I am very thankful to have had the chance to have her as my thesis supervisor.

I would also like to thank three other members of our research team. Shiqi He, François Lepage and Mayank Vadsola overlapped with me for over a year, and acted a bit like my university of Ottawa big brothers and sisters. They gave me no end of advice and wisdom as people who went through the same process.

Then, I would like to thank my soon-to-be wife Ari, who always encouraged me no matter the rate at which I was progressing. She even quickly learned new skills to help me work on this, I am very lucky to have her by my side.

Also, I would like to thank my family. They have shown a lot of support for me, and a lot of interest for my work. This has helped encourage me to keep progressing.

I would like to thank every teacher I had over my whole academic journey. They helped me learn a tremendous amount, and cultivated my curiosity for the infinite number of things to learn next.

Finally, I would like to thank the different programs that supported this research. The support from the Natural Sciences and Engineering Research Council of Canada Discovery Grant (RGPIN-2017-05320) and a Grant from the Collaborative Research and Training Experience (CREATE- 481695-2016) program in Simulation-Based Engineering Science (SBES) are greatly appreciated. The Compute Canada organisation also helped me by providing access to HPC platforms I wouldn't have been able to use otherwise, and the necessary guidance and support to learn to use them.

Contents

Title Page i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures viii

List of Tables xiv

List of Algorithms xv

Lists of Symbols xvi

1 Introduction 1

2 Literature Review 9

2.1 Spectral Element Methods 9

2.2 Graphics Processing Units 10

2.3 Adaptive Mesh Refinement 11

2.4 Dynamic Load Balancing 13

3 Graphics Processing Units 15

3.1 Architecture 16

3.1.1 Hardware model 16

3.1.2 Programming model 17

3.2 Process Parallelism 20
3.3 Data Structure 21
3.3.1 Performance 21
3.3.2 Ease of use 22
3.3.3 Ease of programming 24
3.4 Implementation 25
3.4.1 Memory transfers 25
3.4.2 Kernels 27
3.4.3 Parallel reductions 28
3.4.4 MPI transfers 30

4 Discontinuous Galerkin Spectral Element Method **32**
4.1 Spectral Approximation 32
4.1.1 Basis functions 33
4.1.2 Polynomial interpolation 35
4.2 The Wave Equation Model 37
4.3 DG-SEM 38
4.3.1 Fluxes 44
4.3.2 Time integration 47
4.4 Implementation 49
4.4.1 Fluxes 49

5 Adaptive Mesh Refinement **51**
5.1 Strategies 53
5.1.1 p-refinement 53
5.1.2 h-refinement 53
5.1.3 hp-refinement 53
5.2 Error Estimation 55
5.3 Refinement Criteria 60
5.4 The Mortar Element Method 61
5.4.1 Projection from Element to Mortar 63

5.4.2 Projection from Mortar to Element . . . 65
5.5 Mesh Pre-condition . . . 68
5.5.1 Algorithm . . . 68
5.6 Implementation . . . 69
5.6.1 Counting refining elements . . . 70
5.6.2 Moving elements . . . 70
5.6.3 Moving faces . . . 71

6 Load Balancing **73**
6.1 Hilbert Curve . . . 74
6.1.1 Generation . . . 75
6.1.2 Refinement . . . 78
6.2 Workload Leveling . . . 85
6.3 Reconstruction . . . 88
6.3.1 Data transfer . . . 88
6.3.2 Connectivity . . . 89
6.4 Load Balancing Criteria . . . 91
6.5 Implementation . . . 91
6.5.1 Element exchange . . . 91
6.5.2 Mesh reconstruction . . . 93

7 Results **97**
7.1 Platforms . . . 97
7.1.1 Béluga . . . 97
7.1.2 Narval . . . 98
7.1.3 Consumer hardware . . . 98
7.2 Test Case . . . 98
7.3 Scaling Tests . . . 99
7.3.1 Strong scaling . . . 101
7.3.2 Weak scaling . . . 104
7.4 Adaptive Mesh Refinement Performance . . . 105

7.5 Dynamic Load Balancing Performance 110
7.5.1 Load balancing interval 113
7.5.2 Load balancing threshold 114
7.5.3 Overall load balancing performance 115
7.5.4 Polynomial order influence 116
7.6 Complex Case 117
7.7 Complex Meshes 120
7.7.1 Profiling 123

8 Conclusion **125**
8.1 Summary 125
8.2 Concluding Remarks 127
8.3 Future Work 128
8.3.1 GPU Computing 128
8.3.2 DG-SEM 129
8.3.3 AMR 130
8.3.4 Dynamic Load Balancing 131

Bibliography **132**

A Mesh Renumbering **139**

B Hybrid Solver **146**

List of Figures

1.1 Direct numerical simulation: Complex flow structures arising from turbulent flow over a multi-element airfoil simulated using spectral methods (reprinted from [72]). 2

1.2 GPU architecture [52]: GPUs dedicate much more die space to computations (in green). 3

1.3 Adaptive mesh refinement example for flow around a spacecraft: The mesh is more refined in certain flow regions to capture important features of the flow such as the bow shock ahead of the vehicle (reprinted from [67]). 4

1.4 Adaptive mesh refinement approaches: Two different approaches to refining the important areas of a mesh. (a) Finer meshes are patched onto the initial coarse mesh (reprinted from [5]) (b) Elements are refined individually (reprinted from [37]). 5

1.5 Two types of elemental interfaces: Interfaces can be either conforming or non-conforming. Elements are shown in the thicker black lines. (a) The collocation points in lilac line up along the interfaces. (b) The collocation points do not line up. 6

1.6 Load imbalance: The elements have split unequally in the two worker GPUs, purple and blue, one having a higher computational load. (a) Before refining (b) After refining 6

3.1 CPU and GPU architecture [52]: GPUs dedicate more space to data processing . 16

3.2 GPU programming model [52]: Execution is split into multiple parallelism levels 18

3.3 Data structure: A refined mesh split into two blocks (a) First block (b) Second block 21

3.4 Types of meshes: (a) Structured (reprinted from [63]) (b) Unstructured 23

3.5 Memory accesses: (a) Structured (b) Unstructured 23

3.6 Moving elements: The data stored in dynamic memory does not need to be copied 25

3.7 Reduction: A block of threads computes the minimum of data 30

4.1 Legendre polynomials: The first six Legendre polynomials. 34

4.2 Polynomial interpolation: The Lagrange interpolating polynomials of degree five. 36

4.3 2D simple domain: $[-1, 1] \times [-1, 1]$, with the four normals n_i, i = 0, 1, 2, 3. . . . 40

4.4 2D simple domain: Interior collocation points in lilac, boundary collocation points in blue. 42

4.5 States on both sides of an interface: The normal $\widehat{n}$ of the interface determines the left and right sides. $\mathbf{Q}^L$ and $\mathbf{Q}^R$ are the states on the left and right of the interface, respectively. w^+ and w^- indicate the left-to-right and right-to-left running waves, respectively. 46

5.1 Adaptive mesh refinement: The elements have split near the wave, where the solution is steeper. (a) Before refining (b) After refining 51

5.2 2D p-refinement: The element's polynomial order is increased from 4 to 6, with 5 to 7 collocation points respectively. 53

5.3 2D h-refinement: The element splits into 4 elements. 53

5.4 2D hp-refinement: Refinement of elements creates non-conforming interfaces. . . 54

5.5 Load imbalance: The elements have split unequally in the two worker GPUs, denoted in purple and blue, one having a higher computational load. (a) Before refining (b) After refining 55

5.6 Modes of the solution: We use the known modes and their decay rate to eventually extrapolate unknown modes. (a) Known modes (b) Modeled exponentially decaying modes 58

5.7 Modes decaying: Decay of spectral modes within an element and their best fit line. (a) $\sigma \leqslant 1$ (b) $\sigma > 1$ 60

5.8 Conforming elemental interface: The extrapolated solution lines up between elements. 61

5.9 Non-conforming elemental interfaces: The extrapolated solution does not line up between elements. 62

5.10 The mortar element method: Mortar is added between the elements, shown in blue. 62

5.11 Element to mortar projection: The elements' boundary values are projected to the face. 63

5.12 Mortar to Element projection: The faces' values are projected to the elements. . 66

5.13 Flowchart of the pre-condition algorithm: Multiple iterative passes of the algorithm can be performed to ensure the initial conditions are well resolved. 69

5.14 Moving elements from the old vector to the new: Two elements are h-refined (in red), the others are moved. 71

5.15 Moving faces from the old vector to the new: Two faces split (in red), the others are moved. 72

6.1 Load imbalance: The elements have split unequally in the two worker GPUs, purple and blue, one having a higher computational load. (a) Before refining (b) After refining 73

6.2 Hilbert curve: The first thee Hilbert curves. (a) 2×2 (b) 4×4 (c) 8×8 75

6.3 The four states splitting, with their children's state and ordering. 76

6.4 First levels of the Hilbert curve with statuses s ($s \in \{H, A, B, R\}$). 76

6.5 Child numbering: The $j = 0, 1, 2, 3$ children of an element, used to assign states and ordering. 77

6.6 Mesh refinement Hilbert curve: One element refining. (a) Before splitting (b) After splitting 78

6.7 The four missing combinations: Each corresponds to a single state. 81

6.8 Element rotation: The numbering of the nodes of an element can affect its refinement. (a) Expected orientation (b) Unexpected orientation 82

6.9 Refinement with rotated elements: The numbering is wrong when compared to the correct Hilbert curve from Figure 6.4c. (a) Initial mesh (b) Refined mesh . . 82

6.10 Different referentials: The different referentials will change the deduced state. (a) H (b) R 83

6.11 Refinement with rotated elements using local referential: This is not the next level curve, as seen in Figure 6.4c. (a) Initial mesh (b) Refined mesh 83

6.12 Refinement with rotation offset: The next level Hilbert curve is correct. (a) Initial mesh (b) Refined mesh 84

6.13 Mesh refining: The elements follow mixed levels of the Hilbert curve, without jumps or discontinuities. 85

6.14 Mesh before load balancing: The two blocks have different numbers of elements. 90

6.15 Mesh after load balancing: The two blocks have equal numbers of elements. . . . 90

7.1 Test case: A wave travels through a square domain at a 45° angle. 99

7.2 Strong scaling: The number of workers increases while the problem size stays fixed. $N = 4$, $K = 65536$, $B = 32$ 102

7.3 Strong scaling: The number of workers increases while the problem size stays fixed. $N = 6$, $K = 65536$, $B = 32$ . . . 103

7.4 Strong scaling: The number of workers increases while the problem size stays fixed. $N = 6$, $K = 65536$ (a) $B = 64$ (b) $B = 128$ . . . 103

7.5 Weak scaling: The problem size increases with the number of workers. $N = 4$, $K = 16384/node$ . . . 105

7.6 Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval = 5 . . . 106

7.7 Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval = 20 . . . 107

7.8 Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval = 100 . . . 107

7.9 Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval = 500 . . . 108

7.10 Adaptive mesh refinement performance: Simulation time and error for four pre-condition steps and increasing refinement interval. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, pre-condition steps = 4 . . . 109

7.11 Low load imbalance ($S = 3$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 17752$ (a) Complete domain (b) Area of interest . . . 110

7.12 Low load imbalance ($S = 3$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 17752$ (a) Complete domain (b) Area of interest . . . 111

7.13 Medium load imbalance ($S = 5$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 26734$ (a) Complete domain (b) Area of interest . . . 111

7.14 Medium load imbalance ($S = 5$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 26734$ (a) Complete domain (b) Area of interest . . . 112

7.15 High load imbalance ($S = 7$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 54837$ (a) Complete domain (b) Area of interest . . . 112

7.16 High load imbalance ($S = 7$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 54837$ (a) Complete domain (b) Area of interest 113

7.17 Load balancing performance interval test: Simulation time with refinement and load balancing while increasing load balancing interval. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval = 20, $P = 16$ (a) $S = 3$ (b) $S = 5$ (c) $S = 7$ 114

7.18 Load balancing performance threshold test: Simulation time with refinement and load balancing while increasing load balancing threshold. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval = 20, load balancing interval = 20, $P = 16$ (a) $S = 3$ (b) $S = 5$ (c) $S = 7$ 115

7.19 Load balancing performance: Simulation time with refinement and load balancing while increasing load imbalance. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval = 20, load balancing interval = 20, $P = 16$, load balancing threshold = 1.1 116

7.20 Polynomial order influence: Iteration time with increasing polynomial order N. (a) CPU and GPU. (b) GPU detail, emphasising the slope. 117

7.21 Complex case: A wave goes through a cloud of high pressure. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Initial conditions (b) After the wave and cloud collide 118

7.22 Complex case h-refinement: The elements split more where the cloud meets the wave. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide 118

7.23 Complex case p-refinement: The polynomial order increases more at the center of the cloud and where cloud meets the wave. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide 119

7.24 Complex case rank: As the mesh is refined load imbalance can occur, and elements are sent from one GPU to another when load balancing. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide 119

7.25 Complex mesh example: An airfoil in an unstructured circular domain with its initial mesh. $N_{initial} = 4$, $K_{initial} = 2128$, $S = 3$, $P = 4$ (a) Complete domain (b) Detail 121

7.26 Complex mesh after pre-condition: Elements have been added to better apply initial conditions. $N_{initial} = 4$, $K = 24072$, $S = 3$, $P = 4$ (a) Complete domain (b) Detail 121

7.27 Complex mesh: A wave impinging on an airfoil at $t = 1.5s$. $N_{initial} = 4$, $K = 55680$, $S = 3$, $P = 4$ (a) Pressure (b) Pressure σ, prescribing h-refinement or p-refinement (c) h-refinement, the split level denotes how many times elements have split (d) p-refinement, polynomial order N . . . 122

7.28 Element distribution: How elements are distributed between GPUs. $N_{initial} = 4$, $K = 55680$, $S = 3$, $P = 4$ (a) Element indices, showing how they are ordered along the pseudo-Hilbert curve (b) Rank, showing which GPU contains which elements . . . 123

A.1 Unsorted mesh: The mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned . . . 140

A.2 Circular sorted mesh: The elements are sorted according to their angle and the mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned . . . 140

A.3 Circular sorted mesh element ordering: The elements are very far from their neighbours. . . . 141

A.4 Hilbert renumbering: The element centres are rounded to an underlying Hilbert curve. . . . 142

A.5 Pseudo-Hilbert sorted mesh: The elements are sorted according to their index along a Hilbert curve, and the mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned . . . 143

A.6 Correct Hilbert curve: This is the real Hilbert curve, on a 128×128 domain where it is defined. . . . 143

A.7 Adaptive mesh refinement with an unstructured mesh: The mesh is refined, created elements follow the initial pseudo-Hilbert curve. (a) Initial mesh (b) Mesh after a single refinement step . . . 144

A.8 Adaptive mesh refinement with an unstructured mesh (detail): The mesh is refined, created elements follow the initial pseudo-Hilbert curve. There are some jumps along the curve around the airfoil. (a) Initial mesh (b) Mesh after a single refinement step . . . 144

B.1 Load balancing with heterogenous workers: The first worker is a GPU and has more elements in its mesh block. . . . 147

List of Tables

4.1 Coefficients of the third-order low storage Runge-Kutta method. 48

6.1 Children element state table: child state S, with s being the state of the parent element and j being the jth child. 77

6.2 Children element order table: child order P, with s being the state of the parent element and j being the jth child. 77

6.3 Element state deduction table: Element state as a function of curve entrance and exit side. 79

6.4 First element state deduction table: Element state as a function of curve exit side. 79

6.5 Last element state deduction table: Element state as a function of curve entrance side. 80

6.6 Element state deduction table: Element state as a function of curve entrance and exit sides. 81

6.7 Problem before repartition: The workers have uneven weight. 87

6.8 Problem after repartition: The workers have a better workload distribution. . . . 87

7.1 Load imbalance cases. 113

7.2 Program profiling: The relative GPU computation time spent in the three main modules, along with specific kernels and their relative GPU computation time. . 124

List of Algorithms

3.1 **cuda_memory:** A pressure solution array is allocated on the CPU, then transferred back and forth to the GPU. 26

3.2 **device_vector:** A pressure solution vector is allocated on the CPU, then transferred back and forth to the GPU. 27

3.3 **perform_gpu_computations:** Computations are performed in parallel on the GPU. 28

3.4 **reduce_wave_delta_t:** The minimum Δt of the mesh is reduced in parallel. . 29

3.5 **boundary_conditions:** The solution of elements on either side of the interface between GPUs is transferred. 31

4.1 **calculate_wave_fluxes:** Riemann solver for the wave equation fluxes. 50

6.1 **get_transfer_solution:** The solution data of elements is stored in parallel in an array. 92

6.2 **MPI_datatype:** A MPI datatype is created to send and receive arrays of elements. Only some data members are sent. 93

6.3 **find_faces_to_delete:** Faces with no remaining domain neighbour elements are marked for deletion. 94

6.4 **find_mpi_interface_elements_to_delete:** MPI interface elements should be deleted if the element on the other side of the interface is moved to the same process. 95

6.5 **find_mpi_interface_elements_to_keep:** MPI interface elements should be kept if a received element now connects to it. 95

Lists of Symbols

List of Symbols

a_m, b_m, g_m	3rd-order Runge-Kutta coefficients
a_n	function coefficient, Legendre approximation coefficient
$\widetilde{a}_n$	extrapolated mode
B	thread block size
c	speed of sound
C	a constant, total capacity
C_i	a constant
c_p	capacity of worker p
$\widehat{D}_{ik}$	weighted derivative matrix
D_{ij}	derivative matrix
E	parallel load efficiency
f	function
$\mathbf{f}$	x flux
F	Face projected solution
$\mathfrak{F}$	flux vector
$\mathbf{F}^*$	numerical fluxes
g	function
$\mathbf{g}$	y flux
I_p	number of tasks of worker p
J	face polynomial order
k	function coefficient, Legendre approximation coefficient
K	number of elements
l, l_i, l_j	Lagrange interpolating polynomial
L, L_k	Legendre polynomial, load imbalance
l_p	load imbalance of worker p
M	polynomial order in the y direction
n	n^{th} element

N	polynomial order
n_i	normal vector
p	pressure, worker index
P	projection matrix from faces to elements, number of workers
p_N	interpolated function
s	element side coordinate
S	surface, speedup, split level, eigenvector matrix
t	time
T	function coefficient, Legendre approximation coefficient
$u, u_n, \widehat{u}_n$	solution function, velocity component in x direction
u	x velocity
U	element boundary solution, solution approximation
$u_h(x)$	approximated solution
v	test function, y velocity
V	volume
w	polynomial weight
W	total workload
w_i	weight of task i
$w_{ideal,p}$	ideal weight of worker p
w^-	right-to-left wave
w_p	total weight of worker p
w^+	left-to-right wave
w^0	stationary wave
$\mathbf{X}$	vector field
x_k, y_l	collocation point
z	face side coordinate

List of Greek Symbols

Δl_k	minimum length of element k
Δt	time step size
$\Delta x_k, \Delta y_k$	size of element k in the x and y directions respectively
Λ	diagonal matrix of coefficient matrix $\mathbf{A}$
Ξ	face, mortar
Φ	computed flux on face
Ψ	element boundary solution projected on a face
Ω	element, domain
α	offset of linear least squares best fit
β	slope of linear least squares best fit, normal vector x component

δ_{ij}	Kronecker delta
σ	modes decay rate
τ	truncation error
ϕ_i	basis function
ψ	intermediate term in barycentric Lagrange interpolation

Other Symbols

$\bigtriangledown \cdot$	divergence operator
$\bigtriangledown$	gradient operator
$\int$	integral operator
∂	partial derivative
$\prod$	product operator
$\sum$	summation operator

Superscripts

$'$	function first derivative
-1	inverse
k	k$^{\text{th}}$ element
L	left side
n	n$^{\text{th}}$ element, solution at timestep n
R	right side
T	matrix transpose

Subscripts

0	initial state
i	value for task i, solution at point i
L	left side
max	maximum value
min	minimum value
n	value at time step n
p	value for worker p
R	right side
t	time derivative

x, y	spatial derivative
xx, yy	second spatial derivative

Other Notations

$P(s, j)$	ordering of jth child element of parent of state s
$S(s, j)$	statue of jth child element of parent of state s
$\cdot$	time derivative
∞	infinity
$\lVert\cdot\rVert$	$L2$ norm operator
$\lvert\cdot\rvert$	absolute value operator
$\overline{a}$	average value of a
$\widehat{}$	unit vector
bold	vector

List of Acronyms

AMR	adaptive mesh refinement
API	application programming interface
CCP	chains-on-chains partitioning
CFD	computational fluid dynamics
CFL	Courant Friedrichs Lewy
CGNS	CFD general notation system
CPU	central processing unit
CUDA	compute unified device architecture
DG-SEM	discontinuous Galerkin spectral element method
DNS	direct numerical simulation
DRAM	dynamic random-access memory
GPU	graphics processing unit
HPC	high performance computing
LES	large-eddy simulation
MPI	message passing interface
NACA	National Advisory Committee for Aeronautics
RAM	random-access memory
SBES	Simulation-Based Engineering Science
SEM	spectral element method
SFC	space-filling curve
SIMT	single-instruction, multiple-thread

SM streaming multiprocessor
VTK Visualization Toolkit

Chapter 1

Introduction

Fluid flows are an important part of everyday life. Fluid mechanics is a whole discipline dedicated to studying and describing the behaviour of liquids and gasses in motion. More and more industrial and scientific applications of this discipline surface everyday, from large-scales studies of climate encompassing the whole earth over millennia, to the study of microscopic cells flowing through tiny blood vessels. The aerospace industry is probably the most evident example of how prevalent aerodynamics are in today's world. Before the advent of computers, there were really only two approaches to study the dynamics of fluids: experimental and theoretical. Once the widespread use of computers became the norm, a third method appeared: *computational fluid dynamics (CFD)*. CFD is a numerical approach to solving the equations governing the motion of fluids.

Usage of CFD has been steadily rising in recent years [69]. It is easy to imagine why, when experimental results are so costly to obtain, and a theoretical approach is difficult to apply to complex cases. Nonetheless, CFD is not meant to replace those methods, but to be combined with them: reducing the need to perform experiments, while comparing results with those from actual experiments to verify and validate the numerical models [70].

Even with increasing computing power, some problems are difficult to solve with CFD. Problems of high-dimensionality and complex flows still cannot be solved economically. NASA's CFD Vision 2030 study [69] states that turbulent flow separation is one area that still cannot be predicted accurately. In order to simulate these problems, we will need increased processing power, more efficient use of that power, and higher resolution numerical methods.

Spectral methods are an answer to that need of more accurate numerical methods. Unlike traditional methods like finite differences methods, finite element methods and finite volume methods, spectral methods are high-order methods and converge exponentially fast. Such methods are used to solve complex flows, but are expensive and are only recently emerging from the research environment to tackle more and more complex problems from classical aerodynamics [72] to nuclear reactor flows [48], internal combustion [76], ocean simulation [49], atmosphere

simulation [26], and more. Figure 1.1 shows direct numerical simulation (DNS) of the transition to turbulence of a multi-element airfoil and the arising complex flow structures.

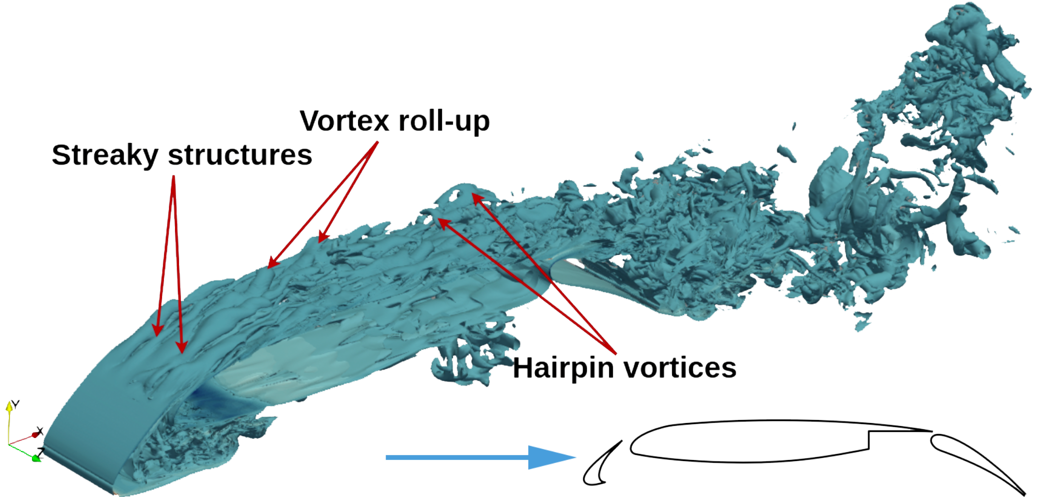


Figure 1.1: Direct numerical simulation: Complex flow structures arising from turbulent flow over a multi-element airfoil simulated using spectral methods (reprinted from [72]).

Instead of modelling the solution as individual values at points, or a linear or quadratic function within a space subdivision, spectral methods represent a function as a truncated series:

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

where $\phi_n(x)$ are high order basis functions. We use this representation in the governing equations to compute the unknowns $\widehat{u}_n$. Spectral methods are usually divided into two categories [33]: collocation methods and Galerkin methods. Galerkin methods solve governing equations in integral form over the domain of interest [64], which is broken up into elements. This method, in combination with the spectral approximation, will be used in this work. It is called the *Galerkin spectral element method*. This method can be formulated both in *continuous* and *discontinuous* versions. Continuous methods decompose the domain into elements, within which the solution is represented as functions. These functions join at the element boundaries, making the method continuous. Discontinuous methods allow the solution to be discontinuous at element boundaries, using fluxes between elements to stabilise the scheme. We refer to these as the *discontinuous Galerkin spectral element method (DG-SEM)*. We will use this second method, as it is easier to parallelise. Since elements are independent and only connect through fluxes the algorithm should map better to massively parallel architectures, where the solution of all elements will be computed concurrently. Non-conforming interfaces are more straightforward to implement with the flux formulation, which should simplify the mesh refinement algorithm. Both SEM methods, continuous or discontinuous, combine the accuracy

and exponential convergence of spectral methods with the geometric flexibility of finite element methods.

With the stagnation of traditional computer *central processing unit (CPU)* operating frequencies over the last few years [57], computer systems have evolved to use more parallel architectures. CPUs are now composed of several computing cores [51] executing tasks together or in parallel, and contemporary *high performance computing (HPC)* platforms consist of several whole computers networked together executing tasks in parallel. Amidst those changes, an inherently parallel architecture has emerged in scientific computing. *Graphics processing units (GPUs)* are computer chips that were initially intended only for computer graphics, for massively parallel workloads. GPUs can now be programmed similarly to traditional CPUs [56], offering their thousands of simpler cores to many kinds of computation. This architecture is optimised for maximal bandwidth, to process great amounts of data as fast as possible. Figure 1.2 reflects this, with the GPU having much more die space dedicated to computations in green. Recently, these processors have been incorporated into HPC platforms [15], enabling massively parallel workloads with theoretical processing power much greater than that using traditional CPUs.

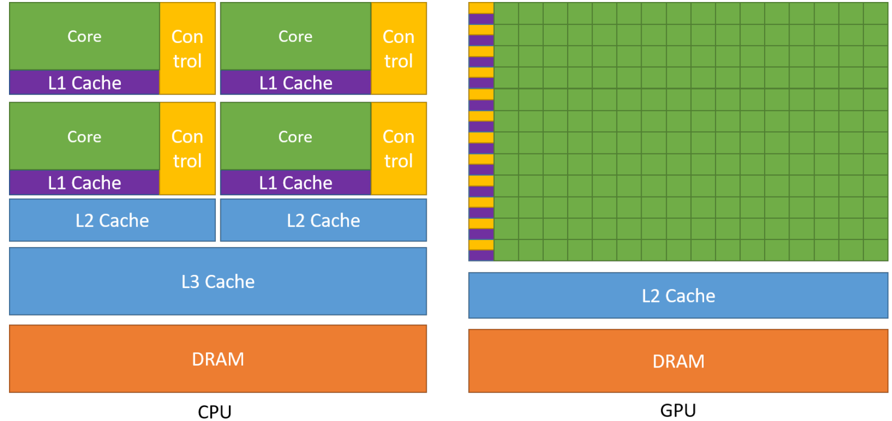


Figure 1.2: GPU architecture [52]: GPUs dedicate much more die space to computations (in green).

We will use GPUs for our computations with the aim of increasing the available processing power to solve complex problems. This will not come easily, as GPU architectures are optimised for static workloads executing the exact same code on all GPU cores. We use the CUDA parallel computing platform [18], which enables programming GPUs using the C++ language with some extensions, much like CPU programming.

Even with highly accurate methods and the processing power of GPU-enabled HPC platforms, the available resources must be spent judiciously. To get the accurate results we want, meshes need to be extremely fine. A mesh this fine throughout the whole domain can become impractical to compute even on the largest HPC systems. The limited resources must be spent

where they will count most. Creating grids that are more refined in areas of interest is possible if those areas are known beforehand. This is a time-consuming process that must be started over for every new problem. It is sometimes not possible to predict where areas of interest will be, for example when modeling chaotic turbulent flows. It is also possible that the mathematically important areas are not easy to identify.

Adaptive mesh refinement (AMR) methods have been developed in order to solve this problem. These methods aim to identify the areas of the mesh that need refinement, and refine those areas to obtain more accurate results while not wasting resources on areas of less interest. Figure 1.3 shows an example of a problem that has been refined in order to better capture the bow shock around a spacecraft. In 1984, Berger and Oliger [5] described a recursive method where refined grids can contain further refined grids, used for the finite difference method to resolve hyperbolic equations. Another AMR method by Khokhlov [37] has reported a factor of 10 savings of both memory and computing time for a 2D solver for the Euler equations. We will use adaptive mesh refinement to make more efficient use of the resources we have and to ease working with the more limited amount of memory of GPUs compared to CPUs.

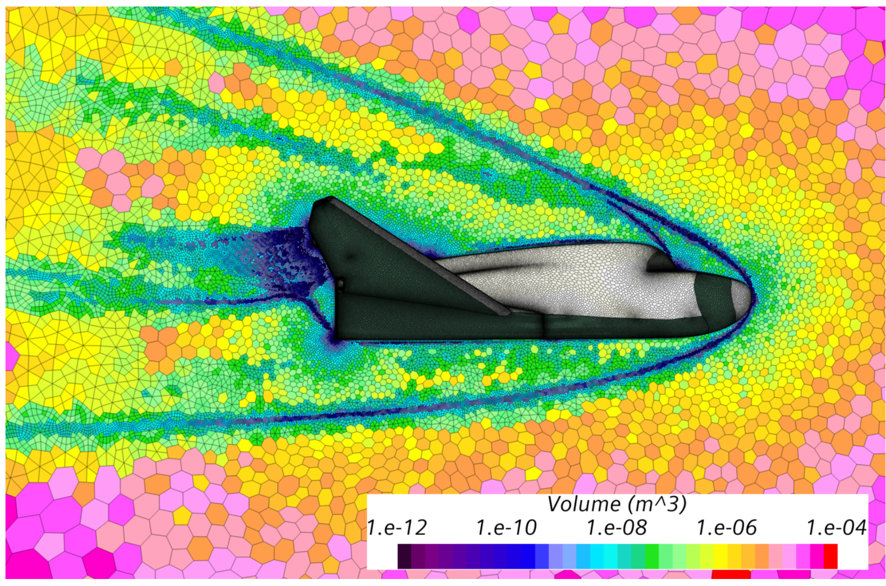
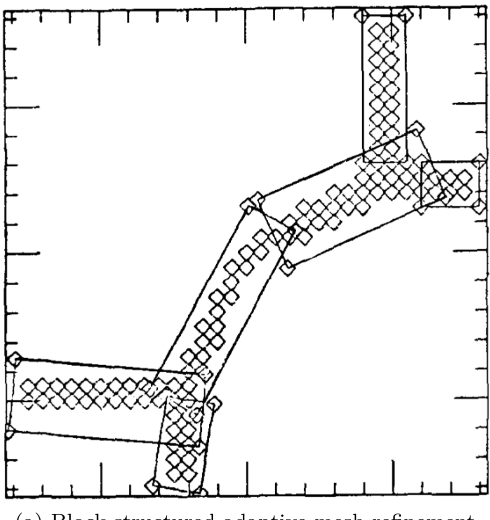
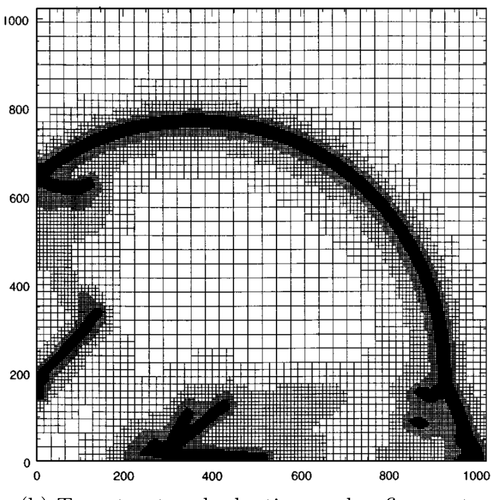


Figure 1.3: Adaptive mesh refinement example for flow around a spacecraft: The mesh is more refined in certain flow regions to capture important features of the flow such as the bow shock ahead of the vehicle (reprinted from [67]).

Two general approaches to AMR exist. Berger and Oliger [5] use *block-structured* AMR, where uniformly refined meshes are overlaid on coarse meshes, as shown in Figure 1.4a. Another approach is *tree-structured* AMR, where elements themselves are refined recursively as in Figure 1.4b. Khokhlov [37] describes such an approach that works fully in parallel. We will use a tree-structured approach, since it can easily work in parallel by having elements refine independently.



(a) Block-structured adaptive mesh refinement



(b) Tree-structured adaptive mesh refinement

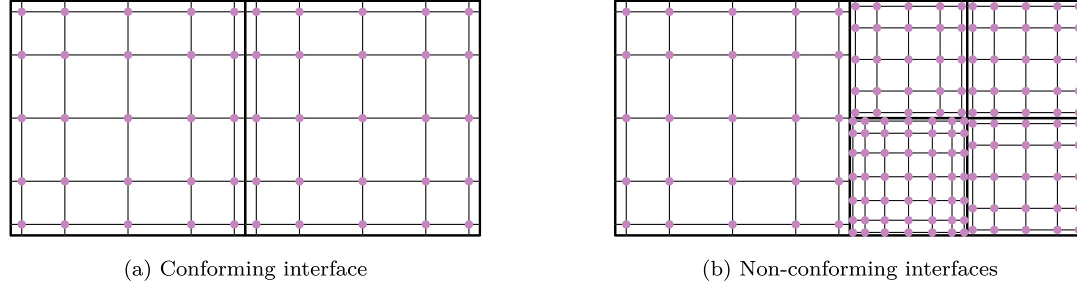
Figure 1.4: Adaptive mesh refinement approaches: Two different approaches to refining the important areas of a mesh. (a) Finer meshes are patched onto the initial coarse mesh (reprinted from [5]) (b) Elements are refined individually (reprinted from [37]).

Two types of refinement will be used in this work. *p-refinement* increases the polynomial order of the spectral approximation inside an element, increasing the number of collocation points inside the element. *h-refinement* splits elements into smaller elements, increasing the number of elements in the mesh.

The choice of which refinement type to use, as well as the choice of which area of the mesh to refine is guided by an error estimator. Mavriplis [44] describes an *a posteriori error estimator* that uses the spectral approximation to estimate the error and the smoothness of the solution. The smoothness of the solution is used to determine which refinement type is expected to increase accuracy most effectively. Elements with a high error get refined, using p-refinement if their solution is smooth or h-refinement if it is not.

The benefits of AMR do not come without added complications. Two main complications are: non-conforming interfaces and load imbalance when performing computations in parallel.

As the mesh refines, it becomes non-uniform. Elements can have neighbours with a different polynomial order, different sizes, or both. The interfaces between such elements are called *non-conforming interfaces*. Such interfaces can be *functionally non-conforming*, *geometrically non-conforming*, or both. Figure 1.5 shows these different types of interfaces. Several methods have been developed to deal with these interfaces. We use the *mortar element method* [43], as it retains the exponential convergence of spectral methods.

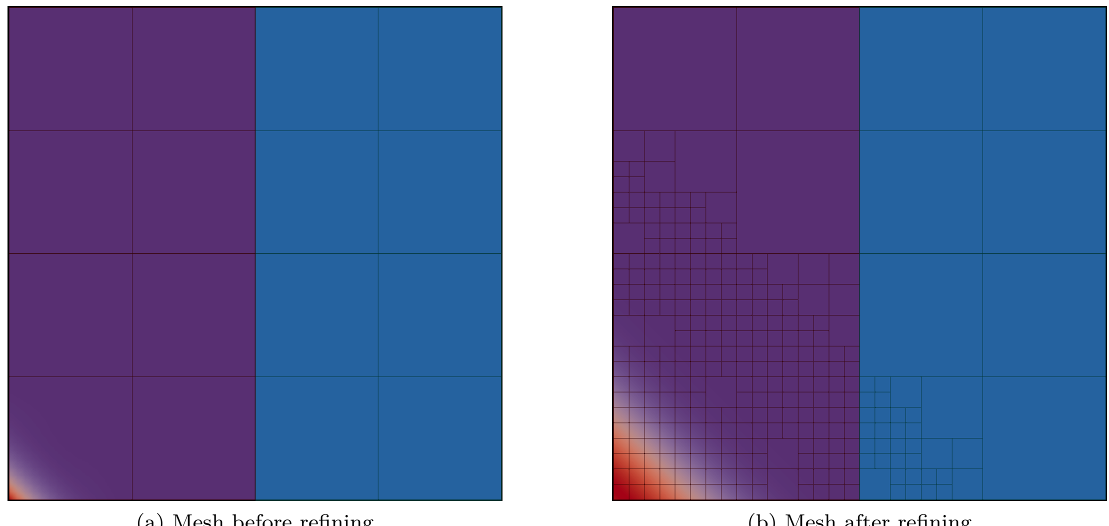


(a) Conforming interface

(b) Non-conforming interfaces

Figure 1.5: Two types of elemental interfaces: Interfaces can be either conforming or non-conforming. Elements are shown in the thicker black lines. (a) The collocation points in lilac line up along the interfaces. (b) The collocation points do not line up.

The second complication of AMR is load imbalance. Solving the problem on multiple GPUs splits the mesh into blocks, one per GPU. As the mesh is refined, it is unlikely that all blocks are refined equally. Every GPU must synchronise at every time step, therefore GPUs with less work to execute will be waiting while the more heavily loaded GPUs finish their computations. The simulation time will be driven by the most heavily loaded GPU, which will degrade performance on parallel systems. Figure 1.6 shows an imbalanced system.



(a) Mesh before refining

(b) Mesh after refining

Figure 1.6: Load imbalance: The elements have split unequally in the two worker GPUs, purple and blue, one having a higher computational load. (a) Before refining (b) After refining

In order to avoid this problem, the mesh needs to be repartitioned as it is refined. This is the process of *dynamic load balancing*. When the mesh is repartitioned, elements are sent between processors to even out their computational load. The repartitioning scheme dictates which elements are sent from one processor to another. This scheme should satisfy several conditions: it must even the load between the workers, it must be fast to execute, and it must

minimise the boundaries between the mesh blocks it creates.

Many approaches to this problem have been studied. *Graph-based* repartitioning schemes represent a mesh as nodes connected by edges, which can then be partitioned by splitting it into parts of equal size while severing as few edges as possible. The well-known *Metis* [34] library uses this technique. Graph-based techniques produce very well partitioned meshes, but can be slower to execute than other techniques and need to know the topology of the entire mesh to partition it, which can be a problem on parallel systems where the problem should be split between the different workers.

A different approach is space-filling curve (SFC) based repartitioning schemes. These schemes reduce multidimensional problems to one dimension by mapping elements to a single curve going through the entire domain [59]. The elements can then be indexed in 1D, and repartitioned by splitting that line into segments of equal size. SFCs are very fast to generate, which is especially relevant to this work as the repartitioning will be performed often while the computation is underway. Some curves, like the *Hilbert curve* [32], retain a very good locality, which will help reduce the size of inter-block boundaries. This can improve the performance of the program by reducing the amount of communication between GPUs. In this work, we will implement a SFC based repartitioning scheme using the Hilbert curve in order to perform dynamic load balancing.

In summary, we want to solve complex problems with high accuracy. We will need a high resolution numerical method, more processing power than traditional CPU programming methods possess, and a way to use that increased processing power as efficiently as possible.

We propose using the discontinuous Galerkin spectral element method as a very accurate numerical method. We perform the computations on graphics processing units to access their great amount of parallel processing power. We use this increased amount of computing power more efficiently by performing adaptive mesh refinement to use more resources where needed, and less elsewhere. We finally add dynamic load balancing to keep the parallel efficiency of the program as high as possible while the topology changes. This approach tackles the main challenges necessary to keep advancing CFD.

In order to showcase these techniques, we solve a sample problem using the 2D wave equation:

$$\frac{\partial^2 p}{\partial t^2} - c^2(p_{xx} + p_{yy}) = 0 \tag{1.2}$$

$$u_t = -p_x \tag{1.3}$$

$$v_t = -p_y \tag{1.4}$$

where p is pressure, c is the speed of sound, and u and v are the x and y direction velocities respectively.

The structure of this thesis follows the list of concepts elaborated above. Chapter 2 discusses the current state of the art of the domain. Chapter 3 presents the GPU architecture and how we tackle the implementation of our algorithms on that special architecture. In Chapter 4, we present the DG-SEM and the derivation used for the sample problem. Then, Chapter 5 explains our proposed algorithm for parallel AMR on GPUs. We then present a methodology to perform dynamic load balancing and the challenges of a dynamic mesh structure in Chapter 6. The performance of the program as well as several comparisons of the impact of the different features using HPC systems are presented in Chapter 7. We conclude with Chapter 8, where we also present ideas for future work.

Chapter 2

Literature Review

The issues that this work aims to overcome are not new, and as such a lot of work has been performed to address them. This section highlights the current state of the art in this domain, and where this work fits within it.

2.1 Spectral Element Methods

Spectral methods were developed by Gottlieb and Orszag [23], starting as early as 1969. Spectral methods attain high order accuracy by modeling solutions as sums of weighted orthogonal functions. In 1984, Patera [58] proposed to combine spectral methods with the finite-element method, creating the spectral element method (SEM). This method has the accuracy of spectral methods and the geometric flexibility of finite-element methods.

Continuous spectral element methods force the solution to be continuous at the boundary between elements by incorporating collocation points at the boundary of elements. *Discontinuous* spectral element methods let the solution be discontinuous at element boundaries, and incorporate fluxes to counterbalance these discontinuities. The discontinuous Galerkin formulation, first proposed by Reed and Hill in 1973 [64], is combined with spectral element methods to create the discontinuous Galerkin spectral element method (DG-SEM) as discussed by Warburton and Hesthaven [30] in 2002. In 2009, Kopriva [41] described a suite of algorithms that can be used to assemble a full solver using this method. We will use the DG-SEM for its geometric flexibility and the parallel simplicity of the flux formulation. This method is used in a wide array of domains, including aerodynamics [4], turbomachinery [17] and ocean and atmospheric modeling [19].

Large scale spectral element solvers have existed for quite a while now. One such solver is Nek5000, an open-source Navier Stokes spectral element solver that runs on CPUs and uses structured grids. It has been proven to scale well to above a million processors [54]. This solver

has been used to perform simulations of micro-scale blood flow [53], direct numerical simulation (DNS) of flows in internal combustion engines [3], and nearly exascale simulations of flows in nuclear reactors [47]. Other solvers exist, such as the Nektar++ [8] library. It is a spectral finite element library used to create purpose-built solvers for different kinds of problems, and works on unstructured grids. Example applications include aerodynamic large-eddy simulation (LES) of sports cars [46], Formula One cars [8], and DNS of flows over various airfoils.

These are the kind of solid foundations we wish to build upon in this work. These solvers traditionally use CPUs to compute their solutions. The next logical step is to perform these computations using GPUs in order to increase their performance. These methods will need to be ported to this new architecture. Additionally, none of these solvers use adaptive mesh refinement (AMR). This will have to be implemented in order to increase the efficiency of computations, using the limited processing resources at our disposal where it counts most. This creates the double challenge of using GPUs for high-order methods and adaptivity, which is especially challenging on GPUs as they are more tailored for static workloads. GPUs excel at solving very parallel problems where every core of the GPU executes the exact same computation on neatly organised data. AMR is not a naturally parallel process, and the resulting meshes incur divergence in the execution on different cores. Nonetheless, if AMR is implemented correctly it should decrease the execution time of the program while reducing resource usage.

2.2 Graphics Processing Units

In recent years, the scientific computing world has started using GPUs to perform computations. Some traditionally CPU-based commercial solvers were upgraded to run entirely on GPUs, or have some parts of the code GPU-accelerated. Krawezik and Poole present a new GPU-accelerated direct sparse solver for Ansys Fluent, boasting a 2.9× speedup compared to the CPU-only solver. Open-source solutions like OpenFOAM [1] have also been extended to work on hybrid CPU and GPU architectures. Another example is RapidCFD [68], a collection of OpenFOAM solvers ported to Nvidia CUDA. These solvers see an improvement of up to 2× and a parallel speedup of 3× when using eight GPUs instead of a single one.

Some brand new solvers have been created for GPUs, such as the AmgX solver, a distributed algebraic multigrid solver. This solver has been shown to have a 2× to 5× speedup compared to a CPU implementation [50].

In the realm of spectral methods, there have been a few advances using GPUs. In 2009, Klöckner et al. demonstrated a DG-SEM solver running on a consumer level GPU that demonstrated a 50× speedup compared to the machine's CPU [39]. The next year, Gödel et al. presented a GPU-accelerated discontinuous Galerkin solver for electric fields, capable of running on multiple GPUs in parallel [21]. The Nek5000 solver gained several GPU compatible branches for Nekbone [22][11], a simplified version of the program that implements its core

solver.

Purpose-built GPU solvers also exist. For example, NekRS is a Navier Stokes solver designed to run on diverse hardware, including CPUs and GPUs [16]. It is a rewrite of the Nek5000 solver, specifically for parallel architectures. It uses the OCCA [45] library, a parallel API that provides code that can be compiled to different architectures, notably OpenMP CPU code, CUDA GPU code and OpenCL code. NekRS shows a parallel efficiency of 80%, and 80% to 90% of the realizable peak performance of HPC systems for large problems [16].

These works have shown that SEM solvers can have very good performance on the GPU architecture. This gives a promising foundation for this work, as the core solver needs to be fast. These solvers use static meshes however, meaning that the mesh does not change during the simulation. This can be problematic for complex problems, when uniformly increasing the mesh resolution to capture fine detail in the solution makes the problem prohibitively expensive to compute. The mesh can be refined beforehand in areas of the flow expected to need refinement, however these areas cannot always be predicted in advance, they may change position in time, or may not be easy to predict visually. The challenge of this work is to implement adaptive mesh refinement in the context of parallel GPU computing.

2.3 Adaptive Mesh Refinement

AMR has been shown to significantly speed up the computation of fluid flow problems. Chalmers et al. report a 3× speedup as well as an improved error when using AMR [10]. The processing time and memory savings will be especially welcome when considering that GPUs have less memory than CPUs.

AMR implementations on CPUs date from quite some time. Berger and Oliger [5] presented the method using overset grids in 1984, and Khokhlov [37] described a tree-structured method that can be accessed and modified in parallel in 1998.

Some software solutions exist to enhance existing code by implementing adaptive mesh refinement. The PARAMESH toolkit [42] for example has demonstrated speedups of over 10× for high refinement levels, when compared to using a uniformly refined mesh. It also enables users to work with the meshes it creates in parallel. The p4est library [7] provides parallel AMR to programs by implementing forest of octrees algorithms. It has been proven to scale well up to 220,000 CPU cores.

Traditional CPU AMR algorithms are well understood, and full books have been written on the subject, such as the work by Plewa et al. [62]. Support for AMR has even started to be added to certain well-established solvers like Nek5000 [55], where the parallel scaling of the method has been studied up to 32,768 CPU cores [60]. This particular implementation uses h-refinement only however.

These are good results, indicating that if the challenges in implementing AMR on GPUs are overcome, significant gains can be expected. These challenges are not small, as the non-conforming meshes produced by AMR increase divergence between GPU threads, and can create load imbalance when the problem is solved in parallel and the mesh is not refined equally in each block. The refinement process itself is also not naturally parallel, therefore care should be taken in the implementation to have as many parts of the process executed in parallel on GPUs as possible. Finally, adaptive meshes complicate the memory layout on GPUs, which are optimised to perform computations on static packed data.

Adaptive mesh refinement is not unique to fluid dynamics. The field of astrophysics has numerous AMR solvers. First, Enzo [6] is an open-source solver dedicated to solving astrophysical fluid flows, notably dark matter dynamics and heating and cooling of flows. This solver is GPU-accelerated, resulting in a 5× speedup over its CPU-only implementation. Then there are the GAMER [65] and GAMER-2 [66] solvers, both open-source GPU-accelerated AMR solvers tailored to astrophysical simulations, including hydrodynamics, magnetohydrodynamics, self-gravity and particle flows. The latter has been shown to scale up to 4096 GPUs and 65,536 CPU cores. In the field of data visualisation, a lot of work has to be done in order to directly represent the data generated by AMR programs. Wang et al. [73] propose an algorithm to accurately render tree-based AMR data in difficult regions such as across discontinuities in the refinement level. Their method uses ray tracing and is computed on CPUs, and can render data without having to interpolate it to a single refinement level.

In the high-order fluid dynamics field, Giuliani and Krivodonova propose an algorithm for GPU-based adaptive mesh refinement for use with high-order methods to calculate gas dynamics [20]. They use triangular meshes, and implement an edge-colouring algorithm in order to avoid race conditions when computing fluxes. They show that AMR can be used to solve problems that are not possible to solve in a reasonable amount of time without refining the mesh locally, and that with a high order solver the time spent on AMR can be relatively low, on the order of 4%. This solver works on a single GPU, and as such does not include dynamic load balancing to address the pitfalls of AMR on multiple GPUs.

We aim to implement AMR in our DG-SEM solver on GPUs. The AMR should be fast so that it does not dominate the execution time, therefore it should be parallelised, run as much as possible on the GPU, and have a structure and resulting meshes compatible with the GPU architecture. We target HPC platforms, therefore the code should have an additional level of parallelism. The program should be able to execute on multiple GPUs and on multiple computers. This means that both the solver and the AMR modules should be able to execute in parallel on multiple mesh blocks. This adds an additional challenge, as the mesh is unlikely to be refined equally in all GPUs. This causes a load imbalance, and reduces the performance of the program as the less heavily loaded GPUs will have to wait for the most heavily loaded GPU at each iteration. We implement dynamic load balancing in order to rearrange the workload between the GPUs to even out the computational load.

2.4 Dynamic Load Balancing

Load balancing is not uniquely a CFD issue. Many load balancing algorithms have been developed in a variety of disciplines to address the issue. For example, Willebeek-LeMair and Reeves [74] discuss different strategies to minimise the execution time of applications running in parallel on different computers when the computational load cannot be predicted. Cardellini et al. [9] describe load balancing algorithms for web servers that avoid overwhelming servers and aim to fully utilise a cluster. In 1988, Kobayashi et al. [40] described an algorithm to dynamically load balance a parallel ray-tracing system by subdividing the space into cubes, and dispatching them to a hierarchy of multiprocessors. This approach has been shown to produce better results than static load balancing by 30% to 50%, and scale up to 1728 processors.

Implementing dynamic load balancing starts by choosing a repartitioning scheme. Graph-based algorithms model a mesh as a group of nodes connected by edges, and try to partition the nodes while breaking as few edges as possible. These solutions can be slow to execute, but result in well-partitioned meshes. The METIS package [35] implements such an algorithm. It executes serially, partitioning meshes using multiple levels of graphs. The ParMETIS library [36] implements a similar algorithm, and can work in parallel. A repartitioning algorithm working in parallel is highly desirable for this work, as each GPU is paired to a single CPU core. As much of the work as possible will have to be executed in parallel on the GPU. These solvers can partition meshes to multiple blocks and repartition AMR meshes [36]. These libraries are widely used, notably by the Nek5000 AMR prototype [60].

Another repartitioning scheme hinges on splitting a one-dimensional list of elements into smaller segments, also called chains-on-chains partitioning (CCP). These algorithms are very fast, but the quality of their partitions depend on the characteristics of the mapping from multidimensional space to the one-dimensional list. One such mapping uses space-filling curves (SFCs). SFCs traverse a multidimensional domain in its entirety, assigning a one-dimensional index to elements. The Peano [59] and Hilbert [32] curves are examples of such curves. SFCs are also widely used as repartitioning schemes, for example by the PARAMESH library [42] and the GAMER family of solvers [66].

In the domain of high-order methods, in 2021 He [29] created a CPU-based dynamic load balancing algorithm for a DG-SEM solver with AMR [29], using the Hilbert curve backed by a hash map data structure. This implementation showed good scaling to 16,384 CPU cores, and a speedup between 5× and 8× compared to a non load-balanced case.

Dynamically load balancing tasks executing on GPUs is more complicated than on traditional CPUs. Due to the parallel execution of tasks on GPUs, memory access patterns needed for optimal performance, and the higher cost of transferring data between GPUs, the performance may not scale as expected when redistributing a workload. We want to leverage the increased processing power of GPUs, therefore the load balancing algorithm should not only redistribute a GPU workload, but also execute on the GPUs in parallel as much as possible.

This very dynamic and serial process will have to be adapted to the GPU architecture.

Dynamically load balancing tasks executing on GPUs is an active area of research. Chen et al. [13] discuss a task-based algorithm for balancing molecular dynamics workloads on a single or multiple GPUs. Their algorithm offers an up to 2.5× speedup, and scales linearly on up to four GPUs. The load balancing itself executes serially on the CPU. Kijsipongse and U-ruekolan [38] propose an algorithm to load-balance K-Means clustering problems running on GPUs that outperforms a parallel CPU implementation by 6.5×.

Building on He's work [29], we develop in this work a dynamic load balancing algorithm based on CCP using the Hilbert curve, exploiting its speed of execution and good data locality. The data locality quality of the curve is well suited for use on GPUs, as memory transfers resulting from large interfaces between GPUs are even more expensive than on CPUs. The algorithm will execute in parallel for multiple GPUs, and aim to execute as much of the code as possible on the GPUs themselves to accelerate the process.

Chapter 3

Graphics Processing Units

In this work, we aim to accelerate the execution of our solver using a relatively new architecture, GPUs. GPUs are massively parallel computer chips that have recently started to be used in the scientific computing domain. GPUs were initially designed to perform repetitive graphical computations, such as the output to a computer screen, video processing, and synthetic image generation. These computations are *embarrassingly parallel*, meaning that they are comprised of many identical computations independent of each other. GPUs are therefore optimised for parallel efficiency. Since these problems deal with huge amounts of data, GPUs are also optimised for maximum bandwidth. They are significantly different from *central processing units (CPUs)*, the traditional architecture on which code is run. CPUs are optimised to solve serial problems as fast as possible, meaning they prioritise serial speed over parallel efficiency, and latency over bandwidth.

Solving fluid flows on a mesh is a parallel problem, as at most steps it is possible to compute the solution on the different elements making up the mesh in parallel. On the other hand, elements must exchange information at their interfaces at each time step, meaning careful synchronisation will be needed. Finally, the adaptive mesh refinement process detailed in Chapter 5 and the dynamic load balancing process from Chapter 6 are not inherently parallel processes. A significant programming effort then becomes necessary to break up these algorithms into parallel tasks to be run on GPUs. Overall, if it is possible to efficiently parallelise these sequential parts of the problem, the gains from running the main solving loop on the highly parallel GPUs should incur a significant speedup to the program.

3.1 Architecture

3.1.1 Hardware model

Graphics processing units and central processing units are fundamentally made up of the same core components. First, data processing hardware, shown in green in Figure 3.1, consists of the parts of the chips that do calculations on data. Control flow hardware, shown in yellow, schedules instructions to be executed on processing hardware. In orange is the main memory of either architecture. Main memory or *dynamic random-access memory (DRAM)*, shown in orange, stores data to be used for computations. The main memory is very big, and takes a relatively long time to access. Finally, in blue and purple are different levels of cache memory. Cache is much smaller and much faster than main memory, and as such stores parts of the memory that the processing hardware is actively using in order to speed up access.

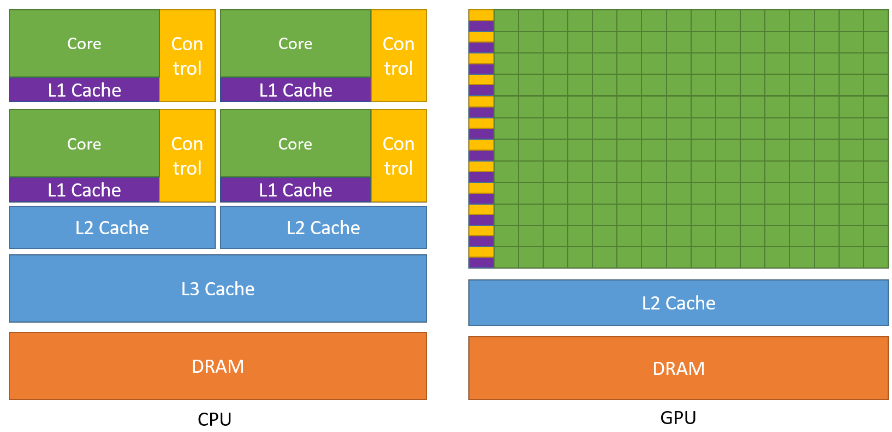


Figure 3.1: CPU and GPU architecture [52]: GPUs dedicate more space to data processing

The two architectures differ in the balance between the size and capacities of these components. Chip area is limited by the processes used to manufacture them, so the different components must share this limited space, and sacrifices must be made. GPUs dedicate much more space to data processing, increasing throughput. Their compute units are also smaller, to work on more data in parallel. CPUs, on the other hand, have fewer bigger compute units with a higher clock speed in order to reduce latency. Their compute units are called cores. The GPU compute units are called CUDA cores, and are arranged in rows as *streaming multiprocessors (SMs)*.

In order to reduce latency, CPUs have a lot of die space dedicated to cache. More cache helps keep data close in access time. More data is loaded from main memory at a time, assuming that data that is near needed data will also be needed later. The deeper hierarchy of cache of CPUs helps keep the cores working on the current problem as fast as possible, never starving for memory. GPUs have less total cache and fewer levels of cache to make room for more CUDA

cores. As a throughput oriented device, it is less important how fast individual tasks complete, as long as more tasks overall are completed. The reduced cache is acceptable because if a CUDA core is waiting on data from the main memory that was not found in cache, it will simply pause execution of that thread and execute another thread. On GPUs, the smallest and fastest cache, L1 cache, is shared within a SM. Accessing shared data is therefore fast as long as conflicts are avoided.

Finally, CPUs also reduce latency by using much more potent control flow units. Firstly, they have one control flow unit per core, where GPUs have one control flow unit per row of cores. This limits how many instructions can be dispatched to the cores making up a SM. The cores will therefore execute the same instruction at the same time in groups of 32, called warps. This makes branching much more expensive if it happens within a warp. If a part of a program has two branches, A and B, the cores executing branch A will go first, followed by the cores executing branch B. These have to be executed sequentially because the control flow unit can only dispatch one instruction per warp. This multiplies the computation time by the number of taken branches. CPU cores only execute a single branch, or can even execute both branches at once while waiting for the result of the conditional, keeping only the correct result once the conditional has been evaluated. The more powerful control flow units of CPUs may also perform branch prediction, consisting of the CPU guessing the most likely branch based on the result of previous computations and starting to execute this branch while waiting for the conditional. All this is performed in an effort to reduce latency as much as possible. Programs running on GPUs must compensate by avoiding divergence as much as possible within warps.

3.1.2 Programming model

This work uses the CUDA programming model. It is a programming language, framework and runtime application programming interface (API) developed to enable programming on GPUs with an extension of the C++ language. Programs using the language execute on the CPU like normal programs, but can schedule parts of the program, called kernels, to be executed on the GPU. Kernels are run in parallel on the GPU, asynchronously from the CPU. The CPU is therefore free to do computations of its own while the GPU is executing, including adding more kernels or transfers to the GPU execution queue. Multiple such queues, called streams, can exist. The purpose of using multiple streams is to execute independent computations on the GPU concurrently, and maximise GPU usage in case a stream is waiting. GPUs can concurrently execute kernels, transfer data from the CPU to the GPU, and transfer data from the GPU to the CPU. These transfers are necessary because CPU and GPU main random-access memory are separate. Data needed for computations on the GPU that cannot be generated on it must be explicitly transferred from the CPU, and results living on the GPU must be transferred back to the CPU in order to be displayed or written to disk. As GPUs are also generally running the displays of non-server computers, results that are only meant to be viewed can be directly displayed and such transfers can be skipped. This is not the case for this work.

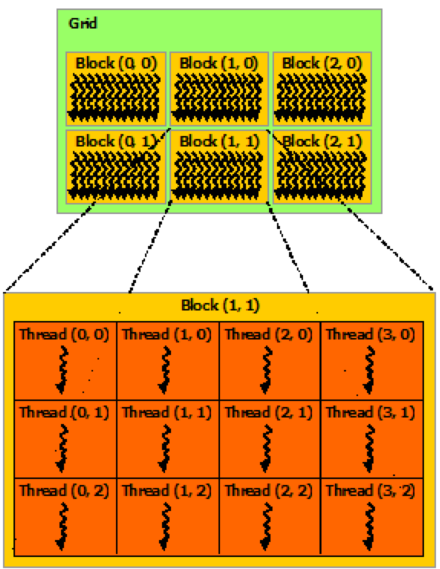


Figure 3.2: GPU programming model [52]: Execution is split into multiple parallelism levels

The programing model ties in with the hardware model from Subsection 3.1.1 by decomposing the execution of kernels to multiple parallel levels. A kernel will be executed in parallel, with each core executing the same function. Figure 3.2 shows how the programming model decomposes a problem. The whole problem to be solved is represented as a multidimensional grid of tasks, with two dimensions in this example. The kernel will execute until the whole grid has been computed. The grid is broken up into blocks of threads. The different blocks will be dispatched to the streaming multiprocessors of the GPU, and each thread of that block will be executed on a CUDA core of the SM. A SM can execute multiple blocks at the same time if there are more cores within the SM than threads within a block. Blocks need to be completely independent of each other, as they cannot be synchronised and may execute in any order. They also cannot share local memory, as each SM has its own L1 cache. Threads within the same block can depend on one another, as they are executed concurrently on the same SM. Threads from a block execute the same instruction at the same time in groups of 32, or warps. This is the *single-instruction, multiple-thread (SIMT)* architecture. If there are more threads within a block than cores within a SM, warps will be run sequentially until all threads are done with the instruction. Threads within a block can share local memory, and synchronise with each other. This is useful for algorithms with steps like reductions, where threads will wait until all threads have computed one level of the reduction, to use the result of that level for the next one.

Typical execution of a program that adds an array A to an array B and stores the result in an array C will proceed as follows. The CPU side of the program will create two arrays of data in its memory, and initialise them as needed. It will then create two arrays of the same size on the GPU using the CUDA runtime API, and transfer the data from the CPU arrays A and B to the GPU arrays. It will then create a result array C on the GPU, and one on the CPU.

It will then launch a kernel on the GPU, with pointers to the GPU A and B arrays, and the GPU C array. The number of elements in the vectors will dictate the size of the execution grid. The number of elements will be split in blocks of threads of fixed size, for example blocks of 128 threads. The number and size of blocks are given to the GPU when calling the kernel. The CPU then must wait until the computation is done using a synchronisation with the CUDA runtime, transfer the data from the GPU C array to the CPU C array, and display the results.

This model fits well with many CFD methods, including the DG-SEM used in this work. We work on a mesh made up of elements and faces connecting them. Elements are independent during most of the computation and their computations can be executed in parallel. The only step from the main solver loop needing extra care is the computation of the fluxes between the elements. The flux computation needs the data from the elements on either side of a face and stores the resulting flux back in both elements so they can compute their derivative. Since there can be multiple faces on each side of an element as detailed in Section 5.4, and each element has multiple sides, multiple threads will attempt to read and write to the same locations at the same time. This is a race condition, where the result of the computation will depend on the order in which the threads try to access the memory location. The results may also be wrong. If two threads read a value, add their contribution, and store the value, only whichever thread wrote its result last will have its contribution accounted for in the final result. Different approaches are possible to solve this issue. For example, Giuliani and Krivodonova [20] use an edge colouring method where the same kernel is run multiple times on different edges in order to avoid writing to the elements at the same time. Our program parallelises this part of the computation on the faces connecting the elements instead, as described in Section 4.4. As long as there are enough elements in the mesh to saturate the cores of the GPU with work, this part of the program should be efficient to run on the GPU.

The adaptive mesh refinement and dynamic load balancing algorithms shown in Chapters 5 and 6 are not so easily parallelised. These algorithms are sequential in nature, with many operations needing the result of previous ones to complete their calculations. In order to parallelise them, a few support structures have to be added for the different processing threads to be able to find the data they have been assigned, and the different memory locations they are allowed to access. In addition, a kernel operating on certain objects, elements for example, should not modify other objects, such as faces. This is done to avoid race conditions when multiple threads try to access the same memory location, faces in this example. The practical consequence of this is that objects moving to a new index in their storage arrays cannot update their neighbour faces to point to their new index. Their neighbours must therefore have the required information to compute where the element will move when the face moves itself, and update the index itself. By designing the algorithms this way, a kernel running on an object type will have each thread writing to only a single object corresponding to its index, avoiding race conditions. More information on how the algorithms are designed is available in Sections 4.4, 5.6 and 6.5.

The meshes generated by adaptive mesh refinement and dynamic load balancing are also

not naturally suited for GPUs. These processors are more suited for static repetitive tasks. Element splitting, changing their polynomial order and moving elements around will reduce the efficiency of the computations if care is not taken to tune the algorithms to this platform. Elements with different polynomial orders in a thread warp will introduce branching, and the warp will take as much time to execute as the highest polynomial order elements. Moving and splitting elements will put more pressure on the limited cache of GPUs, as threads will loop over different elements after refinement and load balancing. The interval between each refinement and load balancing will have to be tuned to trade off between having an optimal mesh for computations and having a static mesh for more iterations.

3.2 Process Parallelism

When trying to solve very large problems, a single node with a single GPU will have insufficient processing power and memory to solve the problem in a reasonable amount of time, or at all. To work with these larger problems, the work needs to be split at another level than GPU parallelism.

The program can use multi-block meshes, with each block being worked on by one process and one GPU. If there are fewer GPUs available on a system than there are processes, some processes will share a single GPU by using asynchronous execution streams. The different processes communicate together using *message passing interface (MPI)* at their boundaries. Since the solution data resides on the GPU, it must first be copied to the main CPU memory before it is sent through the MPI runtime. It then needs to be copied to the receiving process' GPU. This exchange necessitates multiple transfers on different levels and should be optimised as much as possible. An approach to minimise the number of interfaces between the different mesh blocks is explained in Section 6.1.

It is possible to split the mesh into more blocks than there are GPUs in a system. In this case, there will be one process per block, and some processes will share GPUs. The GPUs will be split into virtual partitions, using concurrent execution streams. These streams execute concurrently on a GPU, leveraging the fact that some operations can happen at the same time. A GPU can execute up to one running kernel, one transfer from the CPU to the GPU, and one transfer from the GPU to the CPU. Additionally, the GPU can execute one kernel while another kernel is waiting for memory. These partitions do not increase performance, but are useful to test running more blocks than there are available GPUs, or running the program with a mesh that is split into more blocks than available GPUs without re-splitting it.

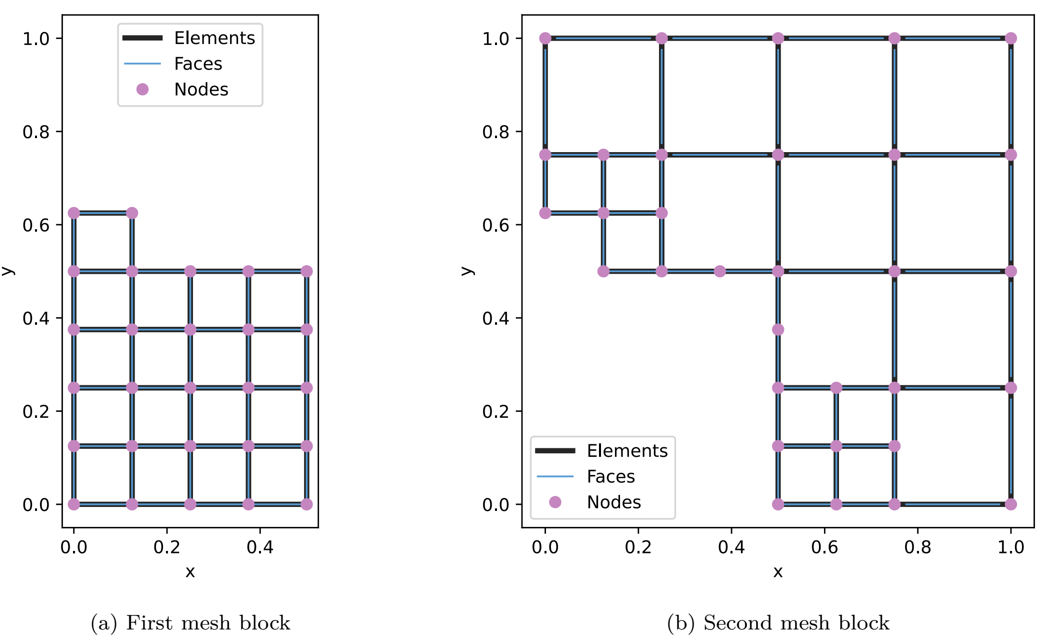
A simple mesh generator program and a mesh partitioner are available as part of this work. The mesh generator can create square meshes of arbitrary resolution with different boundary conditions. The mesh partitioner can split arbitrary single block meshes to multi-block meshes.

These tasks, each consisting of a mesh block, do not necessarily need to run on GPUs.

A CPU version of the program is provided to compare the efficiency of the program and to aid debugging. That version uses CPU workers doing the same computations. The program could even be modified to dispatch work to CPU and GPU workers to use the full computing power of a system. Some work would need to be done to ascertain the relative capacity of the different workers, as a GPU worker running on a complete GPU is much faster than a CPU worker running on a single CPU core. CPU workers could also use multiple threads, with an architecture similar to the GPU one.

3.3 Data Structure

The data structure used in this work has been designed to provide good performance on the GPU architecture, while keeping the ease of use and programming of CPU code. The mesh is unstructured, made up of elements, faces and nodes. Figure 3.3 shows the different components of a two block mesh.



(a) First mesh block

(b) Second mesh block

Figure 3.3: Data structure: A refined mesh split into two blocks (a) First block (b) Second block

3.3.1 Performance

In order to get good performance and fit naturally with the GPU architecture, the different objects are stored as flat one-dimensional arrays on the GPU. The CUDA memory allocation functions are run on the CPU and return a continuous block of memory on the GPU, and

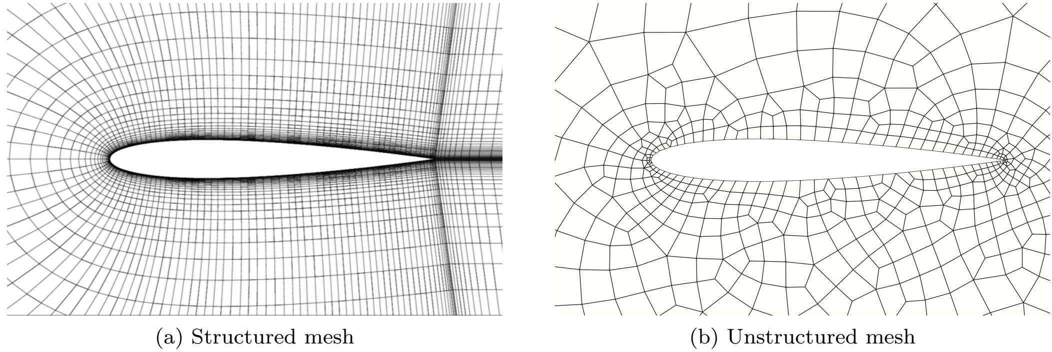
the memory transfers between the CPU and GPU copy continuous blocks of memory. This matches well with having flat arrays of objects. Other data structures, like trees, would need multiple memory allocations from the CPU and a kernel to assemble them in some kind of structure. This would be manageable for a non-adaptive solver as it would be performed once, but with mesh refinement and load balancing the mesh changes constantly. With arrays, an array of the new size is allocated and the objects are moved to it. Thanks to the structure from Subsection 3.3.3, the elements themselves have a known constant size regardless of polynomial order, and have the bulk of their data outside the element object, making them fast to move and making the element array smaller. The flat arrays also reduce indirection and structure the memory accesses. Memory indirection occurs when data is not accessed directly, but multiple data structures must be accessed sequentially to find the address of where the data is stored. Threads from a kernel executing on elements directly access the element at their thread index in the element array.

This greatly simplifies transfers between the CPU and GPU, and mesh reallocation when refining or load balancing. On the other hand, it carries less context on the relation between mesh blocks, complicating the element exchanges from the load balancing process of Chapter 6. It also means that elements and faces move around in the arrays as more are added, and their indices must be updated in other objects linking to them.

Some data needs to be transferred to the CPU during the computation, such as fluxes between mesh blocks in different processes, and data to be written to the output files. These transfers use arrays allocated on the CPU and GPU, and a kernel storing the data into the GPU array. The array can then be transferred to the CPU, and either written to disk or sent to another CPU and then its GPU.

3.3.2 Ease of use

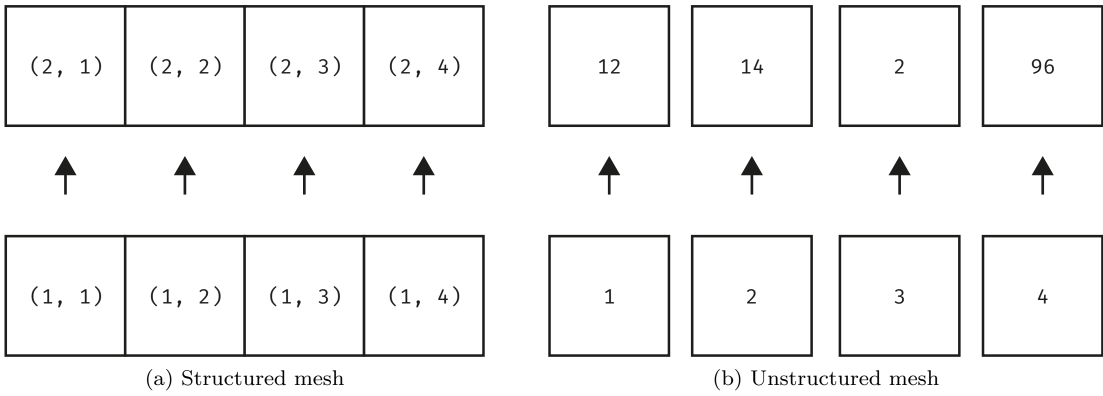
In order to increase ease of use, the program is made to work with unstructured meshes. Unstructured meshes allow for increased flexibility when meshing, and an unstructured mesh solver can easily be made to also read structured meshes. In unstructured meshes, elements are not necessarily numbered in any order. Elements hold an explicit list of the indices of their neighbours. In structured meshes, elements are placed in a regular grid, and have an index describing their location. In 2D, this is the row and column of the element. The neighbours of an element are found by simply incrementing or decreasing by one the index. Figure 3.4 shows a comparison of the two types of meshes used to represent the same geometry.



(a) Structured mesh (b) Unstructured mesh

Figure 3.4: Types of meshes: (a) Structured (reprinted from [63]) (b) Unstructured

Here a concession has been made for flexibility, as structured meshes would be faster than unstructured meshes on a GPU. Threads on a GPU benefit from accessing memory in an ordered fashion. With a structured mesh, thread warps need only one memory access to obtain all their neighbours. As shown in Figure 3.5a, the neighbours are packed in memory, and their index can be computed easily by incrementing the element indices. For unstructured meshes, neighbour indices have to be fetched from an area in memory, and the neighbours are not guaranteed to be contiguous in memory. This adds an indirection, and may require multiple memory accesses. Figure 3.5b shows the relation between elements from an unstructured mesh and their neighbours. Since unstructured meshes enable more complex meshes to be represented than strictly regular quadrilaterals, this tradeoff is accepted, and acts as a worst case scenario for the performance of GPUs.



(a) Structured mesh (b) Unstructured mesh

Figure 3.5: Memory accesses: (a) Structured (b) Unstructured

The program uses the *CFD general notation system (CGNS)* mesh format. Both the provided mesh generator and mesh partitioner output the format, and the mesh partitioner and solver can read the format. The format is widely used, which should enable this program to read the many meshes already available using the format.

The program outputs data using the *Visualization Toolkit (VTK)* format. Once again, this is a widely used format, notably used by the *ParaView* visualisation application. The results of the program should then be easy to manipulate and display.

3.3.3 Ease of programming

The program is written partly using the object-oriented paradigm, especially for the data structure part. Elements, faces and nodes are objects, each with multiple data members and member functions. This groups many parameters together that would be separate arrays otherwise, making it easier to program and keep track of the different members of objects. This has the disadvantage of putting more pressure on the limited cache of GPUs and of potentially increasing the number of memory accesses needed. This is the structure of arrays vs array of structures debate. It increases cache pressure when a loop only accesses a subset of the data members of an object. For example, the solution update function only uses the solution and derivative of an element. When threads loop over those elements, they have to load entire elements into cache. If the solution and derivative were stored into separate arrays, many more elements could fit in cache as only their solution and derivative would be fetched. This would also improve memory accesses, as warps of threads would need fewer memory accesses since the data is continuous.

On the other hand, using objects simplifies many parts of the code. Elements can be sent and received from process to process as a whole: adding a data member does not necessarily mean modifying every part of the code that copies or creates elements. This is especially useful as we can use constructors and destructors for objects. On the other hand, using objects simplifies many parts of the code. Elements can be sent and received from process to process as a whole: adding a data member does not necessarily mean modifying every part of the code that copies or creates elements. This is especially useful as we can use constructors and destructors for objects.

Constructors and destructors are used here because the objects use dynamic memory for their solution data. As elements and faces can have a different polynomial order, they need to store more or less data depending on the polynomial order. This makes it impossible to create an array of objects, unless all objects use the maximum size they can have, negating the memory advantage of lower-order elements. Instead, in this work objects use dynamic memory to hold their variable size arrays. They hold pointers to that memory, making the objects themselves fixed in size. This has the added benefit of making the objects smaller and easier to move around, as only the pointers and the geometry data have to be moved while the bigger solution data stays in dynamic memory. Figure 3.6 shows how an element is moved from the sixth position of an array to the fifth of another. Only the data contained directly in the element object is copied, while the bulk of its data, the yellow pressure solution array in the case of the wave equation solver, stays in place in dynamic memory. The green arrow represents the

element being moved, and the red arrows represent pointers.

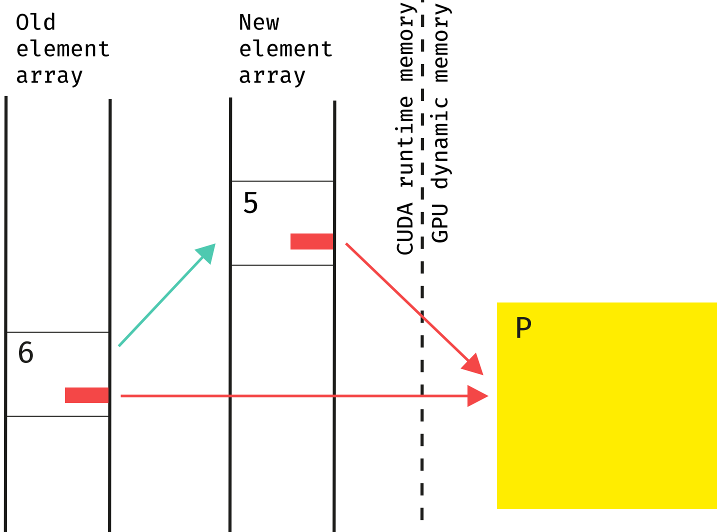


Figure 3.6: Moving elements: The data stored in dynamic memory does not need to be copied

Unlike the main arrays which are allocated from the CPU on the GPU using the CUDA runtime, GPU dynamic memory is allocated on the GPU itself. When the elements are created in a kernel, each thread allocates memory in parallel in the object constructor. That memory is deallocated when the element is deleted. Coupled with move semantics of modern C++, this allows an automatic management of that memory on the GPU, similar to how we would use smart pointers or arrays in usual C++. Dynamic memory allocation on the GPU is a relatively new feature, which enables programming patterns closer to those used for CPU programming.

3.4 Implementation

3.4.1 Memory transfers

GPU memory is separate from CPU memory. In this work, we allocate memory on the GPUs in two ways: memory is allocated using the CUDA runtime from the CPU, and memory is allocated in the GPU dynamic heap directly from the GPU.

Memory allocated by the CUDA runtime can be transferred directly to and from the GPU. We use it to manage arrays of high-level objects, such as elements, faces, and nodes. These allocations and transfers can be performed asynchronously using execution *streams*. When using streams, operations are queued by the CPU for asynchronous execution on the GPU. The CPU is then free to continue executing the program. The CPU will only wait for the GPU to finish execution if a non-asynchronous function is added to the stream, or we issue an

explicit synchronisation instruction such as *cudaStreamSynchronize*. The CPU then waits until all operations queued on the stream are completed on the GPU. The results can then be used on the CPU.

Algorithm 3.1 shows how we would use that kind of memory. After creating our execution stream, we allocate and fill a pressure solution array on the CPU as usual. We then allocate an array on the GPU, and copy the data from the CPU array to it. We can then perform computations on the GPU using that array with the *perform_gpu_computations* kernel. Kernels are explained in Subsection 3.4.2. We can then copy back the data from the GPU array to the CPU array using *cudaMemcpyAsync*. We synchronise with the execution stream to ensure the data has finished copying, at which point we can use the CPU array. To finish up, we deallocate the CPU array, the GPU array, and the execution stream.

Algorithm 3.1 cuda_memory: A pressure solution array is allocated on the CPU, then transferred back and forth to the GPU.

```
auto main(int argc, char* argv[]) -> int {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    const size_t length = 1000;
    double* pressure_cpu = new double[length];
    for (size_t i = 0; i < length; ++i){
        pressure_cpu[i] = 1.0;
    }
    double* pressure_gpu = nullptr;

    cudaMallocAsync(&pressure_gpu, length * sizeof(double), stream);
    cudaMemcpyAsync(pressure_gpu, pressure_cpu, length * sizeof(double),
        cudaMemcpyHostToDevice, stream);

    const numBlocks = 10;
    const blockSize = 100;
    perform_gpu_computations<<<numBlocks, blockSize, 0, stream>>>(
    length, pressure_gpu);

    cudaMemcpyAsync(pressure_cpu, pressure_gpu, length * sizeof(double),
        cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);

    display_cpu_results(length, pressure_cpu);

    delete[] pressure_cpu;
    cudaFreeAsync(pressure_gpu, stream);
    cudaStreamDestroy(stream);
    return 0;
}
```

Since dealing with pointers directly can become error-prone in larger programs, we encapsulate that behaviour into a vector type. Algorithm 3.2 shows a simplified version of Algorithm 3.1 using this new type.

Algorithm 3.2 device_vector: A pressure solution vector is allocated on the CPU, then transferred back and forth to the GPU.

```cpp
using SEM::Device::Entities::device_vector;

auto main(int argc, char* argv[]) -> int {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    std::vector<double> pressure_cpu(1000, 1.0);
    device_vector<double> pressure_gpu(pressure_cpu, stream);

    const numBlocks = 10;
    const blockSize = 100;
    perform_gpu_computations<<<numBlocks, blockSize, 0, stream>>>(
        pressure_gpu.size(), pressure_gpu.data());

    pressure_gpu.copy_to(pressure_cpu, stream);
    cudaStreamSynchronize(stream);

    display_cpu_results(pressure_cpu);

    cudaStreamDestroy(stream);
    return 0;
}
```

Memory allocated in the GPU dynamic heap from the GPU is allocated by the *new* or *malloc* functions similarly to how we would normally do this on the CPU, but from the GPU using GPU functions like kernels. We encapsulate these operations into *cuda_vector*, another vector type similar to that of Algorithm 3.2.

Memory from GPU heap cannot be transferred directly to the CPU using CUDA functions like *cudaMemcpyAsync*, as heap memory is not mapped to the CUDA runtime. To transfer data from that kind of memory, we must copy it to a CUDA runtime allocated array on the GPU using a kernel.

We use heap memory for dynamic data structures living on the GPU that the CPU does not need to use, like the variably-sized solution arrays of individual elements and faces, and the variably-sized lists of faces connected to an element's sides.

3.4.2 Kernels

Functions launched from regular CPU code and executing on the GPU are called *kernels*. Once a kernel is launched it is queued to be asynchronously executed on the GPU. The CPU is then free to continue executing instructions. These kernels, like memory transfers, are queued using execution streams.

Algorithms 3.1 and 3.2 show a kernel being launched, with an array of pressure and its size as arguments. Two other interesting elements are the *numBlocks* and *blockSize* parameters. These two parameters are used between angled brackets <<< >>> to configure the kernel launch. They represent the number of blocks of threads and the number of threads in a block, respectively. As described in Subsection 3.1.2, this dictates how the kernel executes in parallel.

Algorithm 3.3 shows the anatomy of a kernel, as well as a device function. Kernels are annotated with __ **__global__** __, and device functions with __ **__device__** __. Device functions are regular functions compiled to be called from the GPU. The topology of the parallel work grid is stored in the *gridDim* and *blockDim* variables, and the various indices of the threads are stored in the *blockIdx* and *threadIdx* variables.

Algorithm 3.3 perform_gpu_computations: Computations are performed in parallel on the GPU.

```
__device__
auto get_pressure(double p, size_t i) -> double {
    return std::pow(p, i);
}

__global__
auto perform_gpu_computations(size_t n, double* pressure) -> void {
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t i = index; i < n; i += stride) {
        pressure[i] = get_pressure(pressure[i], i);
    }
}
```

3.4.3 Parallel reductions

Parallel reductions are operations that are difficult to make efficient on GPUs and saturate them with work. Harris et al. [27] show a few different ways to optimise these operations. In this work, we need to compute the minimum time step over the whole mesh at every time step. This operation uses geometric and solution data from the elements to find each element's minimum time step size. This data is only available on the GPU.

An implementation must saturate the GPU with work as much as possible: blindly looping one thread over the whole mesh will only use one of the approximately five thousand cores of the GPU, for one thousandth of the speed. Algorithm 3.4 shows how the Δt is computed in parallel over all the elements of a mesh. It implements several of the optimisations from [27].

Algorithm 3.4 reduce_wave_delta_t: The minimum Δt of the mesh is reduced in parallel.

```
template <unsigned int blockSize>
__device__
auto warp_reduce_delta_t_2D(volatile deviceFloat *sdata, unsigned int tid) -> void {
    if (blockSize >= 32) sdata[tid] = std::min(sdata[tid], sdata[tid + 16]);
    if (blockSize >= 16) sdata[tid] = std::min(sdata[tid], sdata[tid + 8]);
    if (blockSize >= 8) sdata[tid] = std::min(sdata[tid], sdata[tid + 4]);
    if (blockSize >= 4) sdata[tid] = std::min(sdata[tid], sdata[tid + 2]);
    if (blockSize >= 2) sdata[tid] = std::min(sdata[tid], sdata[tid + 1]);
}

template <unsigned int blockSize>
__global__
auto reduce_wave_delta_t(
        deviceFloat CFL,
        size_t N_elements,
        const Element2D_t* elements,
        deviceFloat *g_odata) -> void {

    __shared__ deviceFloat sdata[(blockSize >= 64) ? blockSize: blockSize+blockSize/2];
    unsigned int tid = threadIdx.x;
    size_t i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = std::numeric_limits<deviceFloat>::infinity();

    while (i < N_elements) {
        deviceFloat delta_t_wave =
        CFL * elements[i].delta_xy_min_
            /(elements[i].N_ * elements[i].N_ * Constants::c);

        if (i+blockSize < N_elements) {
            delta_t_wave = std::min(
                delta_t_wave,
                CFL * elements[i+blockSize].delta_xy_min_
                    /(elements[i+blockSize].N_ * elements[i+blockSize].N_ * Constants::c));
        }

        sdata[tid] = std::min(sdata[tid], delta_t_wave);
        i += gridSize;
    }
    __syncthreads();

    if (blockSize >= 1024) {
        if (tid < 512) { sdata[tid] = std::min(sdata[tid], sdata[tid + 512]); }
        __syncthreads();
    }
    if (blockSize >= 512) {
        if (tid < 256) { sdata[tid] = std::min(sdata[tid], sdata[tid + 256]); }
        __syncthreads();
    }
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] = std::min(sdata[tid], sdata[tid + 128]); }
        __syncthreads();
    }
    if (blockSize >= 128) {
        if (tid < 64) { sdata[tid] = std::min(sdata[tid], sdata[tid + 64]); }
        __syncthreads();
    }

    if (tid < 32) warp_reduce_delta_t_2D<blockSize>(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Algorithm 3.4 implements several advanced CUDA features not discussed previously. The gist of it is that the threads from a block allocate a working array *sdata* in shared memory. That memory is local to a block of threads, and cannot be accessed within other blocks. The threads then perform several reduction steps, where threads take the minimum of two values with fewer and fewer threads active. Figure 3.7 shows how these steps are performed, with initially four

threads active. Once there are 32 threads left, a warp reduction function is called. This is necessary to remove branching within a warp. All threads will execute these instructions, with threads that are not needed operating on useless data. Once this is done, the first thread copies the result into the output array *g_odata*. There will be one result per block of threads, the final step of the reduction will be performed on the CPU. The function is templated on the size of the thread blocks. All the apparent branching based on *blockSize* will disappear when compiled, as the code is generated for a specific *blockSize*.

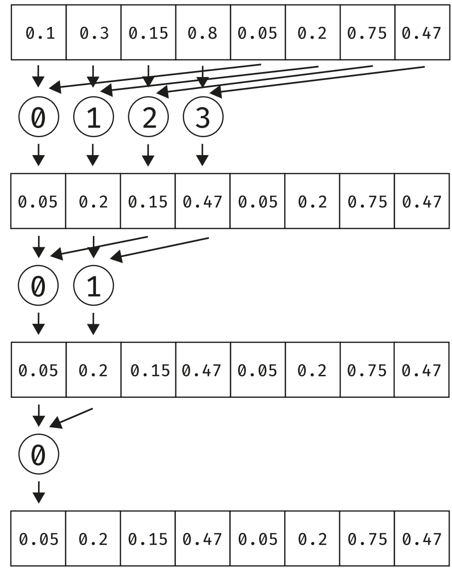


Figure 3.7: Reduction: A block of threads computes the minimum of data

3.4.4 MPI transfers

The fact that the solution data resides on the GPU complicates transfers between the different parallel workers. This is because only CPUs can communicate with each other over MPI. Solution data therefore needs to be fetched from the GPU to the CPU, transferred between CPUs, then transferred from the receiving CPU to its GPU. This is compounded by the fact that elements can have a different polynomial order N which changes the size of their solution arrays, and that these solution arrays are stored in GPU heap memory.

Algorithm 3.5 displays a simplified version of the MPI transfers necessary at each time step to update the solution at boundaries between GPUs. This increased cost compared to traditional CPU versions of these algorithms highlights the need to minimise the surface area between GPUs, as described in Chapter 6. Note that variables ending with an underscore are data members of the mesh object.

Algorithm 3.5 boundary_conditions: The solution of elements on either side of the interface between GPUs is transferred.

```
auto boundary_conditions() -> void {
    int global_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &global_rank);
    int global_size;
    MPI_Comm_size(MPI_COMM_WORLD, &global_size);

    get_MPI_interfaces<<<mpi_outgoing_numBlocks_, mpi_blockSize_, 0, stream_>>>(
        mpi_origin_.size(), elements_.data(),
        mpi_origin_.data(), mpi_origin_side_.data(), max_N_,
        device_p_.data(), device_u_.data(), device_v_.data());

    device_p_.copy_to(host_p_, stream_);
    device_u_.copy_to(host_u_, stream_);
    device_v_.copy_to(host_v_, stream_);
    cudaStreamSynchronize(stream_);

    for (size_t i = 0; i < mpi_process_.size(); ++i) {
        MPI_Isend(
            host_p_.data() + mpi_outgoing_offset_[i] * (max_N_ + 1),
            mpi_outgoing_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);
        MPI_Irecv(
            host_receiving_p_.data() + mpi_incoming_offset_[i] * (max_N_ + 1),
            mpi_incoming_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);

        MPI_Isend(
            host_u_.data() + mpi_outgoing_offset_[i] * (max_N_ + 1),
            mpi_outgoing_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);
        MPI_Irecv(
            host_receiving_u_.data() + mpi_incoming_offset_[i] * (max_N_ + 1),
            mpi_incoming_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);

        MPI_Isend(
            host_v_.data() + mpi_outgoing_offset_[i] * (max_N_ + 1),
            mpi_outgoing_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);
        MPI_Irecv(
            host_receiving_v_.data() + mpi_incoming_offset_[i] * (max_N_ + 1),
            mpi_incoming_size_[i] * (max_N_ + 1),
            MPI_DOUBLE, mpi_process_[i], /*[...]*/);
    }

    MPI_Waitall(6 * mpi_process_.size(), requests_.data(), statuses_.data());

    device_receiving_p_.copy_from(host_receiving_p_, stream_);
    device_receiving_u_.copy_from(host_receiving_u_, stream_);
    device_receiving_v_.copy_from(host_receiving_v_, stream_);

    put_MPI_interfaces<<<mpi_incoming_numBlocks_, mpi_blockSize_, 0, stream_>>>(
        mpi_destination_.size(), elements_.data(),
        mpi_destination_.data(), max_N_,
        device_receiving_p_.data(), device_receiving_u_.data(), device_receiving_v_.data());
}
```

Chapter 4

Discontinuous Galerkin Spectral Element Method

In this chapter we present the methodology of the *discontinuous Galerkin spectral element method (DG-SEM)* and detail its application to solving the 2D wave equation. This method is part of the broader class of spectral methods, applied within local elements as in finite-element methods. This method exhibits the exponential convergence of spectral methods, while adding the ability to model difficult geometries of finite-element methods. The method is described in [41], along with efficient computer programming implementation strategies. The main model problem showcased throughout this work depicts a diagonal wave moving through a square domain.

4.1 Spectral Approximation

Spectral methods, as described by Gottlieb and Orszag [23], model functions as sums of weighted orthogonal functions, often polynomials. Let us denote $\phi_n(x)$ as the polynomial of degree n, and a_n as the weight of said polynomial. The weights make up the spectrum of the solution.

$$f(x) = \sum_{n=0}^{\infty} a_n \phi_n(x) \tag{4.1}$$

Since it is impractical to sum an infinite number of polynomials on a computer, the sum is truncated to a polynomial order N. This becomes the polynomial order of the approximation. The truncated part becomes τ, the truncation error.

$$
\begin{aligned}
f(x) &= \sum_{n=0}^{\infty} a_n \phi_n(x) \\
&= \sum_{n=0}^{N} a_n \phi_n(x) + \sum_{n=N+1}^{\infty} a_n \phi_n(x) \\
&= \sum_{n=0}^{N} a_n \phi_n(x) + \tau \\
&\approx \sum_{n=0}^{N} a_n \phi_n(x)
\end{aligned}
\tag{4.2}
$$

Since we discard part of the spectrum, it is important that the retained polynomial terms capture the essential parts of the solution. We therefore want the weights to decay as fast as possible. This will be a characteristic of the polynomials we choose. For example, Fourier series, Legendre polynomials and Chebyshev polynomials have been proven [41] to have exponentially decaying coefficients for smooth functions $f(x)$, giving them spectral accuracy.

4.1.1 Basis functions

Spectral methods exploit the orthogonality property of sets of functions. For orthogonal functions in the domain $[a, b]$, we have:

$$
\int_a^b \phi_i(x)\phi_j(x)dx = C_i\delta_{ij} \tag{4.3}
$$

with C_i being a constant, and δ_{ij} being the Kronecker delta, defined as such:

$$
\delta_{ij} = \begin{cases} 1, & \text{if} \quad i = j, \\ 0, & \text{if} \quad i \neq j. \end{cases} \tag{4.4}
$$

The inner product of two functions $f(x)$ and $g(x)$ continuous in $[a, b]$, is:

$$
(f, g) = \int_a^b f(x)g(x)dx. \tag{4.5}
$$

We can therefore rewrite Equation 4.3 in inner product form:

$$
(\phi_i, \phi_j) = \int_a^b \phi_i(x)\phi_j(x)dx = C_i\delta_{ij}. \tag{4.6}
$$

Some sets of functions are only orthogonal in the context of integration with a weighting function w:

$$(\phi_i, \phi_j) = \int_a^b \phi_i(x)\phi_j(x)w(x)dx = B_i\delta_{ij}, \tag{4.7}$$

where B_i is a constant.

Fourier series can be used for periodic problems. As for non-periodic problems, Legendre polynomials and Chebyshev polynomials are more practical. Legendre polynomials have a weighting function of $w(x) = 1$, which simplifies computations. We will therefore use the Legendre polynomials.

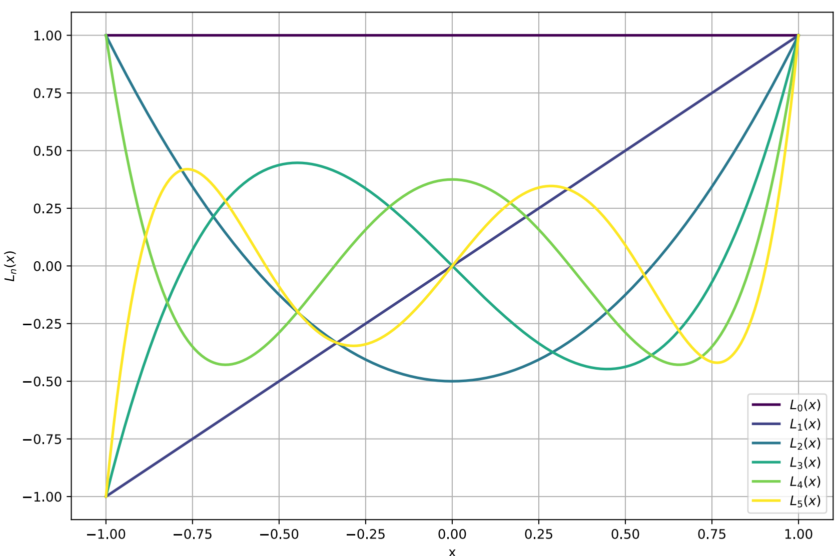


Figure 4.1: Legendre polynomials: The first six Legendre polynomials.

The Legendre polynomials, illustrated in Figure 4.1, were discovered by Adrien-Marie Legendre in 1782. Using their associated weights $w(x) = 1$ and the interval $[-1, 1]$, the polynomials are orthogonal. The Legendre polynomials $L_k(x)$ of degree k therefore have the property:

$$\int_{-1}^{1} L_{k_1}(x)L_{k_2}(x)dx = 0, \quad if \;\; k_1 \neq k_2. \tag{4.8}$$

The Legendre polynomials can be generated using a three-term recursive formula. Knowing that the first two polynomials are $L_0(x) = 1$ and $L_1(x) = x$, all subsequent polynomials can be generated from the following formula.

$$L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x) \tag{4.9}$$

These polynomials will help us compute integrals using Gauss quadrature. For an interval $[-1,1]$, the Gauss quadrature formula is:

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=0}^{N} w_i f(x_i). \tag{4.10}$$

The points x_i used for the quadrature are the Gauss quadrature points, or collocation points. Using the roots of the (N + 1)$^{\text{th}}$ Legendre polynomial, the quadrature is exact for polynomials of degree $\leq 2N+1$. This is the *Gauss-Legendre quadrature rule.* We use the roots of the (N + 1)$^{\text{th}}$ Legendre polynomial to get the points x_i and weights w_i.

$$x_i = roots\ \ of\ \ L_{N+1}(x), \quad i = 0, \dots, N \tag{4.11}$$

$$w_i = \frac{2}{(1-x_i^2)\left[L'_{N+1}(x_i)\right]^2} \tag{4.12}$$

4.1.2 Polynomial interpolation

Polynomial interpolation fits data to a polynomial passing through known points. The polynomial has an order equal to the number of points minus one. In this work we use Lagrange integrating polynomials l_j of degree N.

$$l_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x - x_i}{x_j - x_i} \tag{4.13}$$

At each Gauss-Legendre collocation point x_i only one of the integrating polynomials has the value 1, the others having a value of 0. This mimics the Kronecker delta δ_{ij}.

$$l_j(x_i) = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{4.14}$$

Figure 4.2 displays the interpolating polynomials of degree five:

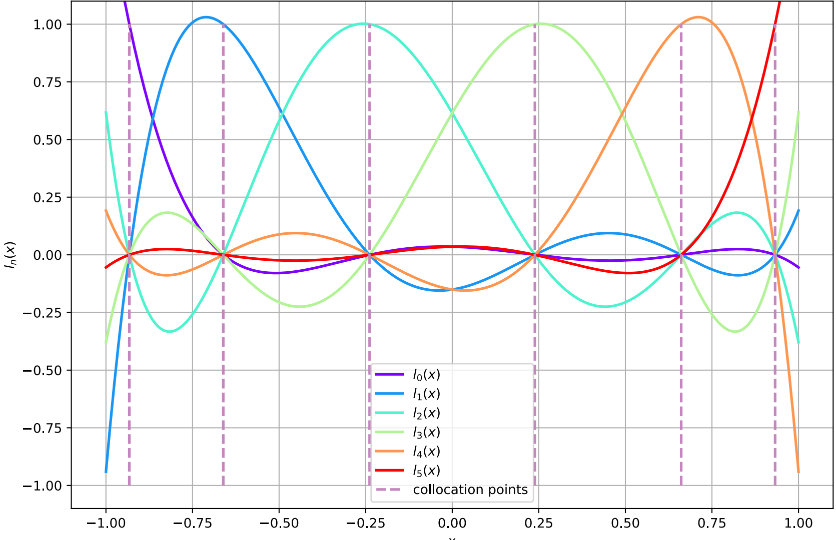


Figure 4.2: Polynomial interpolation: The Lagrange interpolating polynomials of degree five.

With this, we can interpolate a function $p_N(x)$ of degree N:

$$p_N(x) = \sum_{j=0}^{N} p(x_j) l_j(x). \tag{4.15}$$

We can also modify the Lagrange interpolation to the barycentric form.

$$p_N(x) = \psi(x) \sum_{j=0}^{N} p(x_j) \frac{w_j}{x - x_j} \tag{4.16}$$

$$\psi(x) = \prod_{i=0}^{N} (x - x_i) \tag{4.17}$$

$$w_j = \frac{1}{\prod_{\substack{i=0 \\ i \neq j}}^{N} (x_j - x_i)} \tag{4.18}$$

We retain the same property as for the initial form:

$$\psi(x) \sum_{j=0}^{N} \frac{w_j}{x - x_j} = 1. \tag{4.19}$$

We can then rewrite the complete Lagrange interpolation in barycentric form. By pre-computing the barycentric weights w_j, we can compute the interpolated value at any point

x using only the known values at the collocation points, without computing the interpolating polynomials.

$$p_N(x) = \frac{\sum_{j=0}^{N} p(x_j) \frac{w_j}{x - x_j}}{\sum_{j=0}^{N} \frac{w_j}{x - x_j}} \tag{4.20}$$

4.2 The Wave Equation Model

We want to solve the 2D wave equation:

$$\frac{\partial^2 p}{\partial t^2} - c^2 (p_{xx} + p_{yy}) = 0 \tag{4.21}$$

$$u_t = -p_x \tag{4.22}$$

$$v_t = -p_y \tag{4.23}$$

with p being the pressure, u and v being the two components of the velocity, and c being the sound speed. The three equations can be combined into one.

$$\frac{\partial^2 p}{\partial t^2} + c^2 \left((u_x)_t + (v_y)_t \right) = 0 \tag{4.24}$$

We then integrate once with respect to time and apply the proper boundary conditions to get:

$$p_t + c^2 (u_x + v_y) = 0. \tag{4.25}$$

We can also write Equation 4.25 in matrix form,

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_x + \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_y = 0, \tag{4.26}$$

or in vector form:

$$\mathbf{q_t} + B\mathbf{q_x} + C\mathbf{q_y} = 0, \tag{4.27}$$

where

$$\mathbf{q} = \begin{bmatrix} p \\ u \\ v \end{bmatrix}. \tag{4.28}$$

B and C are constant matrices and as such can be combined with the derivatives.

$$\mathbf{f} = B\mathbf{q} \tag{4.29}$$

$$\mathbf{g} = C\mathbf{q} \tag{4.30}$$

$$\mathbf{q_t} + \mathbf{f_x} + \mathbf{g_y} = 0 \tag{4.31}$$

By combining $\mathbf{f}$ and $\mathbf{g}$ into a flux vector $\mathfrak{F}$,

$$\mathfrak{F} = \mathbf{f}\widehat{x} + \mathbf{g}\widehat{y}, \tag{4.32}$$

we get the conservative form of the wave equation:

$$\mathbf{q_t} + \bigtriangledown \cdot \mathfrak{F} = 0, \tag{4.33}$$

where $\widehat{x}$ and $\widehat{y}$ are the unit vectors in the x and y directions, respectively.

We apply the divergence theorem to Equation 4.33 to obtain two integrals, one on the control volume V and one on the surface S of said volume.

$$\frac{d}{dt}\int_V \mathbf{q}dV = -\int_S \mathfrak{F} \cdot \widehat{n}dS \tag{4.34}$$

where $\widehat{n}$ is the surface normal vector, pointing outwards.

4.3 DG-SEM

Section 4.1 shows how to approximate functions with polynomials and how to compute integrals on fixed intervals, for example $[-1, 1]$. We must then apply those tools to arbitrary domains. Pure spectral approximations apply the polynomials to the whole domain through a simple mapping. This can be problematic when discontinuities or very steep solutions are present, as the finite order polynomials cannot match the solution, and solutions exhibit oscillations. Also, these methods only work for simple domain shapes, such as quadrilaterals, triangles and circles in 2D. To accommodate more complex domains, such as blood vessels and airfoils,

the method must be modified.

Patera proposed using spectral element methods [58] to combine the accuracy of spectral methods and the generality of finite element methods. Finite element methods can model complex geometries by splitting the domain into multiple elements, with the equations being solved on the elements. Spectral element methods are used to solve a wide range of complex problems [14], including the full Navier-Stokes equations. Continuous spectral element methods use the Gauss-Lobatto quadrature points to make sure the solution is continuous at element boundaries. These quadrature points include points at -1 and 1.

In this work, we use the *discontinuous Galerkin spectral element method (DG-SEM)*. The DG-SEM uses the Gauss quadrature points described in Subsection 4.1.1. The discontinuous Galerkin method, first proposed by Reed and Hill in 1973 [64], permits the absence of quadrature points at the edges of the domain, letting the different elements be discontinuous at their boundaries. These discontinuities are then counterbalanced by fluxes between elements. Hesthaven and Warburton [31] give a comprehensive reference on the discontinuous Galerkin method. The present work follows the derivations from Kopriva [41] for the DG-SEM. This method is especially well suited for the intended work on GPUs. It is highly accurate and geometrically flexible. It is also easy to parallelise, as the discontinuities between elements make them independent. Each element can be computed independently in parallel, then the fluxes are also computed in parallel. The flux formulation also simplifies working with non-conforming interfaces, such as those resulting of adaptive mesh refinement, as described in Section 5.4.

We start by deriving the discontinuous Galerkin approximation for a simple 2D domain spanning $[-1, 1] \times [-1, 1]$ shown in Figure 4.3.

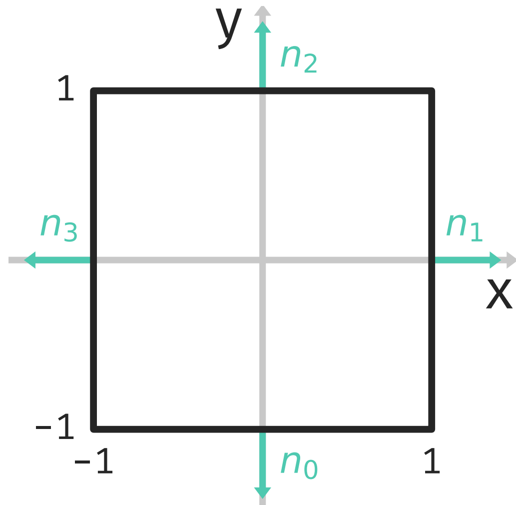


Figure 4.3: 2D simple domain: $[-1,1]\times[-1,1]$, with the four normals n_i, i = 0, 1, 2, 3.

We approximate our solutions by polynomials of degree N in both directions, in Lagrange form.

$$\mathbf{q} \approx \mathbf{Q} = \sum_{i=0}^{N}\sum_{j=0}^{N} \mathbf{Q}_{ij} l_i(x) l_j(y) \tag{4.35}$$

$$\mathfrak{F} \approx \mathbf{F} = \sum_{i=0}^{N}\sum_{j=0}^{N} \left(\mathbf{F}_{ij}\widehat{x} + \mathbf{G}_{ij}\widehat{y}\right) l_i(x) l_j(y) \tag{4.36}$$

where $\mathbf{F}_{ij}\widehat{x} + \mathbf{G}_{ij}\widehat{y} = B\mathbf{Q}_{ij}\widehat{x} + C\mathbf{Q}_{ij}\widehat{y}$.

We rewrite Equation 4.33 in weak form, using the test functions v.

$$(\mathbf{q_t}, v) + (\bigtriangledown \cdot \mathfrak{F}, v) = 0 \tag{4.37}$$

The test functions themselves can also be expressed in Lagrange form:

$$v = \sum_{i=0}^{N}\sum_{j=0}^{N} \widetilde{v}_{ij} l_i(x) l_j(y). \tag{4.38}$$

Substituting Equation 4.38 in 4.37, we obtain

$$\sum_{i=0}^{N}\sum_{j=0}^{N}\left[\int_{\Omega}\mathbf{q_t}l_i(x)l_j(y)dxdy+\int_{\Omega}(\bigtriangledown\cdot\mathfrak{F})\,l_i(x)l_j(y)dxdy\right]\widetilde{v}_{ij}=0, \tag{4.39}$$

where Ω is the whole domain. Since $\widetilde{v}_{ij}$ are arbitrary, the relation must be true for all i, j independently. Therefore:

$$\int_{\Omega}\mathbf{q_t}l_i(x)l_j(y)dxdy+\int_{\Omega}(\bigtriangledown\cdot\mathfrak{F})\,l_i(x)l_j(y)dxdy=0,\quad i,j=0,\ldots,N \tag{4.40}$$

and

$$\int_{\Omega}\mathbf{Q}_t\phi_{ij}dxdy+\int_{\Omega}\bigtriangledown\cdot\mathbf{F}\phi_{ij}dxdy=0,\quad i,j=0,\ldots,N \tag{4.41}$$

with $\phi_{ij}=l_i(x)l_j(y)$. We can then rewrite Equation 4.41 in inner product form:

$$(\mathbf{Q}_t,\phi_{ij})+(\bigtriangledown\cdot\mathbf{F},\phi_{ij})=0. \tag{4.42}$$

We then use Green's first identity to extract the boundary contribution from Equation 4.42's second term. Green's first identity is stated as follows.

$$\int_{\Omega}\bigtriangledown\cdot(\phi\mathbf{X})\,d\Omega=\int_{\Omega}\bigtriangledown\phi\cdot\mathbf{X}+\phi\bigtriangledown\cdot\mathbf{X}d\Omega=\oint_{S}\phi\mathbf{X}\cdot\widehat{n}dS \tag{4.43}$$

where Ω is the domain, and S is that domain's boundary. ϕ is a scalar function, and $\mathbf{X}$ is a vector field. $\widehat{n}$ is the boundary's normal vector, pointing outwards. Applied to Equation 4.42's second term, we obtain:

$$(\bigtriangledown\cdot\mathbf{F},\phi_{ij})=\int_{-1}^{1}\int_{-1}^{1}\phi_{ij}\bigtriangledown\cdot\mathbf{F}dxdy=\oint_{S}\phi_{ij}\mathbf{F}\cdot\widehat{n}dS-\int_{-1}^{1}\int_{-1}^{1}\mathbf{F}\cdot\bigtriangledown\phi_{ij}dxdy. \tag{4.44}$$

The complete equation becomes:

$$(\mathbf{Q}_t,\phi_{ij})+\oint_{S}\phi_{ij}\mathbf{F}^*\cdot\widehat{n}dS-\int_{-1}^{1}\int_{-1}^{1}\mathbf{F}\cdot\bigtriangledown\phi_{ij}dxdy=0. \tag{4.45}$$

The boundary conditions are weakly enforced using numerically computed boundary fluxes $\mathbf{F}^*$, as explained in Subsection 4.3.1.

We then need to evaluate the integrals. A distinction is made between the interior terms that are evaluated using the collocation points within an element shown in lilac in Figure 4.4, and the boundary term calculated at collocation points on the element boundaries, shown in

blue.

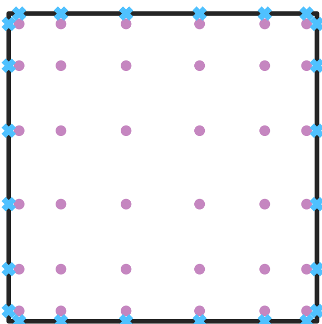


Figure 4.4: 2D simple domain: Interior collocation points in lilac, boundary collocation points in blue.

We start by applying Gauss quadrature to the first term of Equation 4.45.

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t\phi_{ij}dxdy &= \int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t l_i(x)l_j(y)dxdy \\ &= \sum_{k=0}^{N}\sum_{l=0}^{N}\frac{d\mathbf{Q}\left(x_k,y_l\right)}{dt}l_i(x_k)l_j(y_l)w_k^{(x)}w_l^{(y)}\end{aligned} \tag{4.46}$$

Since the integrand is a polynomial of degree $2N$, and the Gauss quadrature is exact for polynomials up to $2N+1$, the integral calculation is exact. We use the fact that only one integrating polynomial l_i is equal to 1 and all others are 0 at each collocation point x_k to get:

$$\int_{-1}^{1}\int_{-1}^{1}\mathbf{Q}_t\phi_{ij}dxdy = \frac{d\mathbf{Q}\left(x_i,y_j\right)}{dt}w_i^{(x)}w_j^{(y)}. \tag{4.47}$$

We do the same for the third term of Equation 4.45.

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1}\mathbf{F}\cdot\triangledown\phi_{ij}dxdy &= \int_{-1}^{1}\int_{-1}^{1}\mathbf{F}(x,y)l_i'(x)l_j(y)+\mathbf{G}(x,y)l_i(x)l_j'(y)\;\;dxdy \\ &= \sum_{k=0}^{N}\sum_{l=0}^{N}\left[\mathbf{F}(x_k,y_l)l_i'(x_k)l_j(y_l)+\mathbf{G}(x_k,y_l)l_i(x_k)l_j'(y_l)\right]w_k^{(x)}w_l^{(y)} \\ &= \sum_{k=0}^{N}\mathbf{F}(x_k,y_j)l_i'(x_k)w_k^{(x)}w_j^{(y)}+\sum_{l=0}^{N}\mathbf{G}(x_i,y_l)l_j'(y_l)w_i^{(x)}w_l^{(y)}\end{aligned} \tag{4.48}$$

Now only the second term is left, the integral over the boundary. We start by applying Gauss quadrature to the developed integral.

$$
\begin{aligned}
\oint_S \phi_{ij}\mathbf{F}^* \cdot \widehat{n} dS =& \int_{-1}^{1} l_i(x) l_j(-1)\mathbf{F}^*(x,-1)\cdot(-\widehat{y})\,dx \\
&+ \int_{-1}^{1} l_i(x) l_j(1)\mathbf{F}^*(x,1)\cdot(\widehat{y})\,dx \\
&+ \int_{-1}^{1} l_i(-1) l_j(y)\mathbf{F}^*(-1,y)\cdot(-\widehat{x})\,dy \\
&+ \int_{-1}^{1} l_i(1) l_j(y)\mathbf{F}^*(1,y)\cdot(\widehat{x})\,dy \\
=& \sum_{k=0}^{N} l_i(x_k) l_j(-1)\mathbf{F}^*(x_k,-1)\cdot(-\widehat{y})\,w_k^{(x)} \\
&+ \sum_{k=0}^{N} l_i(x_k) l_j(1)\mathbf{F}^*(x_k,1)\cdot(\widehat{y})\,w_k^{(x)} \\
&+ \sum_{l=0}^{N} l_i(-1) l_j(y_l)\mathbf{F}^*(-1,y_l)\cdot(-\widehat{x})\,w_l^{(y)} \\
&+ \sum_{l=0}^{N} l_i(1) l_j(y_l)\mathbf{F}^*(1,y_l)\cdot(\widehat{x})\,w_l^{(y)} \\
=& l_j(-1)\mathbf{F}^*(x_i,-1)\cdot(-\widehat{y})\,w_i^{(x)} \\
&+ l_j(1)\mathbf{F}^*(x_i,1)\cdot(\widehat{y})\,w_i^{(x)} \\
&+ l_i(-1)\mathbf{F}^*(-1,y_j)\cdot(-\widehat{x})\,w_j^{(y)} \\
&+ l_i(1)\mathbf{F}^*(1,y_j)\cdot(\widehat{x})\,w_j^{(y)}
\end{aligned} \tag{4.49}
$$

Note that the integrands are also of order $2N$, and are computed exactly. We put Equations 4.47, 4.49 and 4.48 in Equation 4.45 and divide by the weights $w_i^{(x)} w_j^{(y)}$.

$$
\begin{aligned}
&\frac{d\mathbf{Q}(x_i,y_j)}{dt} \\
&+ \left\{ \left[l_i(-1)\,\mathbf{F}^*(-1,y_j)\cdot(-\widehat{x})\,\frac{1}{w_i^{(x)}} + l_i(1)\,\mathbf{F}^*(1,y_j)\cdot(\widehat{x})\,\frac{1}{w_i^{(x)}} \right] - \sum_{k=0}^{N} \mathbf{F}(x_k,y_j)\,\frac{l_i'(x_k)\,w_k^{(x)}}{w_i^{(x)}} \right\} \\
&+ \left\{ \left[l_j(-1)\,\mathbf{F}^*(x_i,-1)\cdot(-\widehat{y})\,\frac{1}{w_j^{(y)}} + l_j(1)\,\mathbf{F}^*(x_i,1)\cdot(\widehat{y})\,\frac{1}{w_j^{(y)}} \right] - \sum_{l=0}^{N} \mathbf{G}(x_i,y_l)\,\frac{l_j'(y_l)\,w_l^{(y)}}{w_j^{(y)}} \right\} \\
&= 0, \quad i,j = 0,\ldots,N
\end{aligned} \tag{4.50}
$$

We define $l_i'(x_k)$ as the derivative matrix D_{ki}. Since it is constant, we also incorporate the

weights w_i and w_k.

$$
\begin{aligned}
\widehat{D}_{ik} &= -\frac{D_{ki} w_k}{w_i} \\
&= -\frac{l_i'(x_k) w_k}{w_i}
\end{aligned} \tag{4.51}
$$

This simplifies Equation 4.50 to:

$$
\begin{aligned}
&\frac{d\mathbf{Q}(x_i, y_j)}{dt} \\
&+ \left\{ \left[l_i\left(-1\right) \mathbf{F}^*\left(-1, y_j\right) \cdot \left(-\widehat{x}\right) \frac{1}{w_i^{(x)}} + l_i\left(1\right) \mathbf{F}^*\left(1, y_j\right) \cdot \left(\widehat{x}\right) \frac{1}{w_i^{(x)}} + \sum_{k=0}^{N} \mathbf{F}\left(x_k, y_j\right) \widehat{D}_{ik}^{(x)} \right] \right\} \\
&+ \left\{ \left[l_j\left(-1\right) \mathbf{F}^*\left(x_i, -1\right) \cdot \left(-\widehat{y}\right) \frac{1}{w_j^{(y)}} + l_j\left(1\right) \mathbf{F}^*\left(x_i, 1\right) \cdot \left(\widehat{y}\right) \frac{1}{w_j^{(y)}} + \sum_{l=0}^{N} \mathbf{G}\left(x_i, y_l\right) \widehat{D}_{jl}^{(y)} \right] \right\} \\
&= 0, \quad i, j = 0, \ldots, N
\end{aligned} \tag{4.52}
$$

The collocation points, weights and derivative matrices from Equation 4.52 can be computed directly and stored for the whole computation. Algorithms for those are discussed in Section 4.4. The next section discusses the computation of the numerical boundary fluxes $\mathbf{F}^*$.

4.3.1 Fluxes

We now need to compute the numerical fluxes $\mathbf{F}^*$ to approximate the physical fluxes $\mathfrak{F}$. We recall the formulation for the fluxes from Equation 4.34. A more detailed derivation for the flux is found in [41].

$$
\mathfrak{F} \cdot \widehat{n} = \mathbf{f} n_x + \mathbf{g} n_y = (B n_x + C n_y)\mathbf{q} = A\mathbf{q} \tag{4.53}
$$

with matrices B and C taken from Equation 4.26.

$$
B = \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} C = \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{4.54}
$$

We also define the normal vector $\widehat{n} = \alpha\widehat{x} + \beta\widehat{y} = n_x\widehat{x} + n_y\widehat{y}$. We can then write the A matrix:

$$A = \begin{bmatrix} 0 & \alpha c^2 & \beta c^2 \\ \alpha & 0 & 0 \\ \beta & 0 & 0 \end{bmatrix}. \tag{4.55}$$

We will use an upwind scheme [71] to calculate the flux at element interfaces. On an interface between two elements, the left and right elements in that coordinate system, the state at the interface is the left state if the wave goes from left to right, or the right state if the wave goes from right to left. This is the upwind state of the elemental boundary.

Since it is not evident what the upwind state of the system is in Equation 4.53, we will decouple the wave components. We decompose A into a diagonal matrix Λ, its right eigenvectors S, and their inverse S^{-1}.

$$\Lambda = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} S = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{\alpha}{2c} & -\frac{\alpha}{2c} & \beta \\ \frac{\beta}{2c} & -\frac{\beta}{2c} & -\alpha \end{bmatrix} S^{-1} = \begin{bmatrix} 1 & \alpha c & \beta c \\ 1 & -\alpha c & -\beta c \\ 0 & \beta & -\alpha \end{bmatrix} \tag{4.56}$$

The diagonal matrix can be further decomposed into three wave directions: left-to-right, right-to-left, and stationary.

$$\begin{aligned} \Lambda = \begin{bmatrix} c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} &= \begin{bmatrix} c & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \Lambda^+ + \Lambda^- + \Lambda^0 \end{aligned} \tag{4.57}$$

We can then split the wave into three components, each containing only one direction.

$$A = S\Lambda^+ S^{-1} + S\Lambda^- S^{-1} + S\Lambda^0 S^{-1} = A^+ + A^- + A^0 \tag{4.58}$$

We can evaluate the flux with:

$$\mathbf{F} \cdot \widehat{n} = A^+\mathbf{q} + A^-\mathbf{q} + A^0\mathbf{q}. \tag{4.59}$$

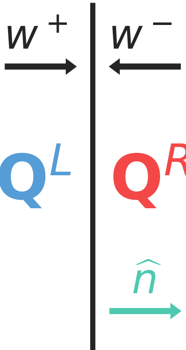


Figure 4.5: States on both sides of an interface: The normal $\widehat{n}$ of the interface determines the left and right sides. $\mathbf{Q}^L$ and $\mathbf{Q}^R$ are the states on the left and right of the interface, respectively. w^+ and w^- indicate the left-to-right and right-to-left running waves, respectively.

Figure 4.5 shows a boundary, with the associated states and waves. This guides us in choosing the states $\mathbf{q}$ in Equation 4.59. The upwind state for the left-to-right running wave is $\mathbf{Q}^L$, and the upwind state for the right-to-left running wave is $\mathbf{Q}^R$. The stationary wave only has zeros in its matrix, and can be removed. This gives us the formulation for the numerical fluxes. This is what is called the Riemann problem.

$$\mathbf{F}^* \cdot \widehat{n} = A^+ \mathbf{Q}^L + A^- \mathbf{Q}^R \tag{4.60}$$

We can now compute the numerical fluxes. We define the characteristic variable $\mathbf{W}$.

$$\mathbf{W} = S^{-1}\mathbf{q} = \begin{bmatrix} 1 & \alpha c & \beta c \\ 1 & -\alpha c & -\beta c \\ 0 & \beta & -\alpha \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} p + c\,(\alpha u + \beta v) \\ p - c\,(\alpha u + \beta v) \\ \beta u - \alpha v \end{bmatrix} \equiv \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} \tag{4.61}$$

We define w^+, w^- and w^0 as the left-to-right, right-to-left and stationary waves, respectively:

$$\begin{aligned} w^+ &\equiv p + c\,(\alpha u + \beta v) \\ w^- &\equiv p - c\,(\alpha u + \beta v) \\ w^0 &\equiv \beta u - \alpha v. \end{aligned} \tag{4.62}$$

Next, we multiply with the diagonal matrices Λ, which contain the direction of the waves:

$$\Lambda^+ \mathbf{W} = \begin{bmatrix} c & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} = c \begin{bmatrix} w^+ \\ 0 \\ 0 \end{bmatrix},$$
$$\Lambda^- \mathbf{W} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} = -c \begin{bmatrix} 0 \\ w^- \\ 0 \end{bmatrix}, \tag{4.63}$$
$$\Lambda^0 \mathbf{W} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Finally, multiplying by S, we get the full numerical flux:

$$\mathbf{F}^* \cdot \widehat{n} = S\Lambda^+ S^{-1} \mathbf{Q}^{\mathbf{L}} + S\Lambda^- S^{-1} \mathbf{Q}^{\mathbf{R}} = \begin{bmatrix} \frac{c}{2}\left(w^{+,L} - w^{-,R}\right) \\ \frac{n_x}{2}\left(w^{+,L} + w^{-,R}\right) \\ \frac{n_y}{2}\left(w^{+,L} + w^{-,R}\right) \end{bmatrix}. \tag{4.64}$$

The superscripts L and R indicate which state is used, according to Equation 4.60.

With the relation 4.64, we can compute the numerical fluxes. The algorithm used to compute these fluxes is presented in Subsection 4.4.1. It is commonly known as the Riemann solver.

4.3.2 Time integration

Now that we know how to compute the interior terms and the numerical fluxes of Equation 4.52, we can compute the time derivative $\mathbf{Q}_t$. We need to integrate the equation in time. We will use a third-order low-storage Runge-Kutta method, as described by Williamson [75]. As the name suggests, this method uses low memory to attain high temporal accuracy. This is especially useful in the context of GPU computation, as GPUs tend to have less available memory than usual HPC CPU platforms. This method requires $2N$ storage, where N is the number of variables. Concretely, we will need to store the computed time derivatives in temporary arrays, which will then be added to the real time derivative arrays.

The method works as follows. For a differential equation

$$u_t = H(u, t), \tag{4.65}$$

we will use Δt to represent the time step, n to represent the number of time steps thus far, and $t_n = n\Delta t$ as the current time. u is the solution vector with N variables, and U^n is the

approximation of the solution at t_n. The method gives the approximation of the solution at the next timestep as:

$$
\begin{aligned}
&U \leftarrow U^n \\
&T \leftarrow H\left(U, t_n\right) \\
&U \leftarrow U + \frac{1}{3}\Delta t T \\
&T = -\frac{5}{9}T + H\left(U, t_n + \frac{1}{3}\Delta t\right) \\
&U \leftarrow U + \frac{15}{16}\Delta t T \\
&T \leftarrow -\frac{153}{128}T + H\left(U, t_n + \frac{3}{4}\Delta t\right) \\
&U^{n+1} \leftarrow U + \frac{8}{15}\Delta t T
\end{aligned} \tag{4.66}
$$

T is introduced as a temporary variable to store the time derivative after partial time steps. As U and T are overwritten at each of the three stages, the storage required is only $2N$, where N is the number of variables making up U. The algorithm can be summarised as the following stage run three times, where m is the stage number.

$$
\begin{aligned}
&T \leftarrow a_m T + H\left(U, t_n + b_m \Delta t\right) \\
&U \leftarrow U + g_m \Delta t T
\end{aligned} \tag{4.67}
$$

The following table lists the coefficients a_m, b_m and g_m.

m	a_m	b_m	g_m
0	0	0	$\frac{1}{3}$
1	$-\frac{5}{9}$	$\frac{1}{3}$	$\frac{15}{16}$
2	$-\frac{153}{128}$	$\frac{3}{4}$	$\frac{8}{15}$

Table 4.1: Coefficients of the third-order low storage Runge-Kutta method.

This method is an explicit time stepping method, therefore we have to fulfill the *Courant Friedrichs Lewy (CFL)* condition on the time step size to ensure stability. The CFL condition is stated as:

$$
\left|\frac{c\Delta t}{\Delta x}\right| \leq \text{CFL} \tag{4.68}
$$

where the CFL number is usually 1. As we have a higher-order time integration scheme, the CFL number can be increased above this number [24].

To account for the uneven spacing between the collocation points inside an element, we will estimate the smallest spacing in an element as:

$$\Delta x = \frac{\Delta l_k}{N^2} \tag{4.69}$$

where Δl_k is the minimum length of element k, and N is the polynomial order of that element. The global timestep will be chosen as the minimum timestep of all elemental timesteps.

$$\Delta t = \min_{k \in 0:K} (\Delta t_k) \tag{4.70}$$

$$\Delta t_k \leq \text{CFL} \frac{\Delta l_k}{c N^2} \tag{4.71}$$

4.4 Implementation

4.4.1 Fluxes

Algorithm 4.1 shows how fluxes are computed on the GPU. This function is a good example of measures that have to be taken in order to avoid race conditions when performing computations in parallel on GPUs. In most of the program, we parallelise over elements, with each thread performing computations on elements. When computing fluxes, both elements on either side of a face add their contribution to the face. In addition, an element can have multiple faces on each side, a side effect of the non-conforming interfaces from Section 5.4. If we parallelise over elements, race conditions will occur when multiple elements try to add their contribution to the same face at the same time. The possibility then arises that the contribution of some elements would be dropped if it is overwritten by another element. This is not a problem for serial or multi-process single threaded CPU programs.

To remedy this, we parallelise over faces for the flux computation. Earlier computations, also parallelised over faces, project the boundary solution of both elements to two solution arrays, left and right, in the face objects. The flux computation can then compute the fluxes and store them in a single flux solution array in the face objects. Afterwards, we will be able to once again parallelise over elements to project these fluxes back to the elements, and continue the normal execution of the program.

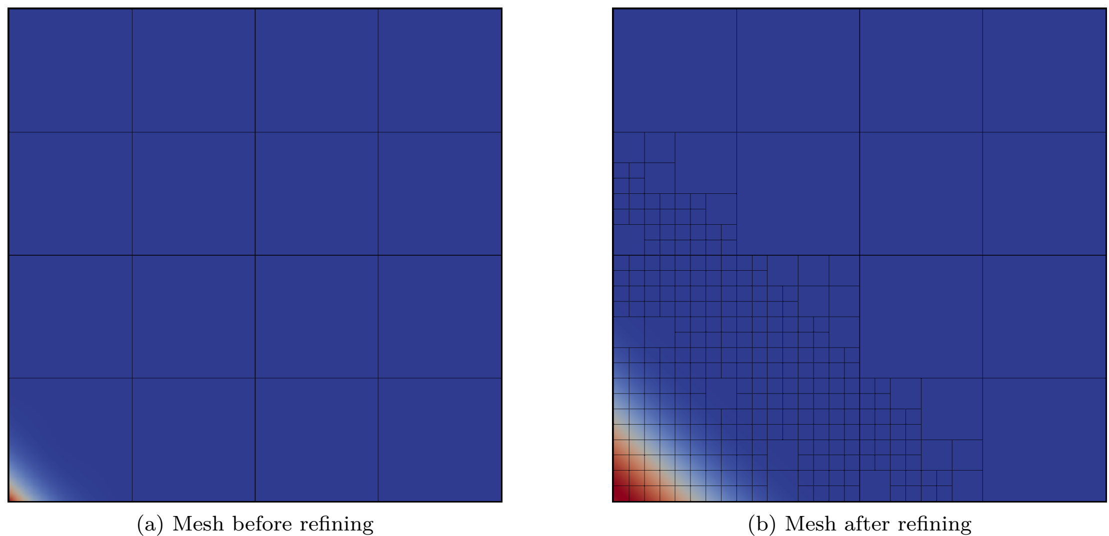
Algorithm 4.1 calculate_wave_fluxes: Riemann solver for the wave equation fluxes.

```
__global__
auto calculate_wave_fluxes(size_t N_faces, Face2D_t* faces) -> void {
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t face_index = index; face_index < N_faces; face_index += stride) {
        Face2D_t& face = faces[face_index];

        // Computing fluxes
        for (int i = 0; i <= face.N_; ++i) {
            const Vec2<deviceFloat> u_L {face.u_[0][i], face.v_[0][i]};
            const Vec2<deviceFloat> u_R {face.u_[1][i], face.v_[1][i]};

            const deviceFloat w_L = face.p_[0][i] + Constants::c * u_L.dot(face.normal_);
            const deviceFloat w_R = face.p_[1][i] - Constants::c * u_R.dot(face.normal_);

            face.p_flux_[i] = Constants::c * (w_L - w_R) / 2;
            face.u_flux_[i] = face.normal_.x() * (w_L + w_R) / 2;
            face.v_flux_[i] = face.normal_.y() * (w_L + w_R) / 2;
        }
    }
}
```

Chapter 5

Adaptive Mesh Refinement

Computers constantly increase in power, thanks to incremental progress made on known processes and new architectures such as those described in Chapter 3. However, processing power and memory are still limited. While the size of problems studied has increased in step with the available resources, it is still necessary to carefully manage those limited resources in order to maximise the efficiency of simulations. Some flow regions may be more interesting or harder to compute, benefiting from an increase in resolution. On the other hand, some flow regions may have less happening in them or be easier to compute, and a decrease in resolution may be acceptable. An example is given in Figure 5.1, where the areas near the wave in red have been refined.



(a) Mesh before refining

(b) Mesh after refining

Figure 5.1: Adaptive mesh refinement: The elements have split near the wave, where the solution is steeper. (a) Before refining (b) After refining

It is possible to increase the number of elements and/or the polynomial order of an entire mesh before solving the problem. This increases resolution in important areas of the flow, but also increases resolution everywhere else in the domain, where the increased computation cost provides no benefit. It is sometimes possible to predict where to refine before solving the problem, such as around static shock waves in predictable locations. In these cases, the mesh can be refined in those areas before the computation starts. However, it is not always possible to know these areas beforehand, or these areas may move as time advances if the problem is transient. The error may also be higher in unforeseen areas.

Adaptive mesh refinement is the process of automatically altering the mesh as the computation goes, in areas where the effect of that increase in resolution is most needed. To find out where to refine, the solution error must be estimated, and a refinement method must be chosen. The error can be estimated by extrapolating the value of the next mode in the spectral approximation, and the slope of the last few modes can guide the choice of the refinement method. The two refinement methods studied here are h-refinement and p-refinement, increasing the number of elements and the polynomial order, respectively.

Once the mesh has been refined, the element boundaries can become geometrically or functionally non-conforming. In order to accurately compute fluxes between the elements, we use the mortar element method [43]. Mortar, in the form of faces, is added between the elements, and the boundary solution is projected from the elements to the faces' own collocation points. After the fluxes are computed on the faces, they are projected back from the faces to the elements.

This whole process must be executed in parallel and with as much of the work as possible executed on the GPU. GPUs are more efficient at working with fixed workloads where all threads of a block execute the same instruction. Refining the mesh moves memory around and can introduce additional branching if the elements have different polynomial orders or non-conforming interfaces. The refinement process itself is also hard to implement on GPUs, as elements must either move, change their polynomial order or split into multiple elements in parallel. If the different threads are not perfectly coordinated, we risk operating on the same destination memory location, creating race conditions.

When solving problems in parallel, the mesh will be split into blocks, each assigned to a process and its GPU. As the mesh is refined, it is possible that the blocks are refined unequally. This can lead to a load imbalance between the different GPUs. As the different GPUs need to synchronise at each timestep, lightly-loaded GPUs will always be waiting for heavily-loaded GPUs. The computation time will therefore be driven by the most heavily-loaded GPU, and the speedup incurred by parallelizing the program will be reduced. To improve this side effect of mesh refinement, a dynamic load balancing algorithm is implemented in Chapter 6.

5.1 Strategies

5.1.1 p-refinement

The first refinement strategy is p-refinement. When refining this way, elements increase their polynomial order to increase their resolution. The lower-order solution is projected to the higher-order collocation points. Figure 5.2 shows the collocation points before and after an element is refined from a N of four to six. In this work, the polynomial order is increased in steps of 2, and is required to be identical in the x and y directions. These restrictions are not necessary but are implemented to simplify programming and increase GPU efficiency.

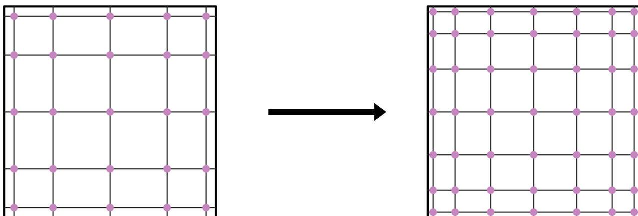


Figure 5.2: 2D p-refinement: The element's polynomial order is increased from 4 to 6, with 5 to 7 collocation points respectively.

5.1.2 h-refinement

The second refinement strategy is h-refinement. h-refinement splits single elements into several elements occupying the same space. In this work we exclusively deal with quadrilaterals, and these elements split into four smaller elements when h-refining as shown in Figure 5.3 for the same reasons as stated above. The solution is projected from the initial big element to each of the four offspring elements.

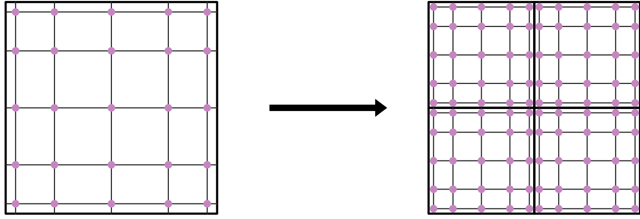


Figure 5.3: 2D h-refinement: The element splits into 4 elements.

5.1.3 hp-refinement

In order to maximise computation efficiency, both p-refinement and h-refinement are used together. An hp-adaptive program uses either p-refinement or h-refinement on an element-by-

element basis, depending on which is expected to reduce error the most. The choice between the two methods is detailed in Section 5.3.

Once the mesh has been refined, it is possible that the interfaces between elements become non-conforming. Each interface is made up of two element edges on either side of the interface, shown in bold black on Figure 5.4. Recall from Chapter 4 that fluxes are computed between the elements from the extrapolated values of the solution on collocation points corresponding to the polynomial order of the elements. These sit on the bold black lines in the figure. This works directly if two neighbouring elements have the same polynomial order and they completely share an edge. However, after p-refinement, interfaces can become functionally non-conforming, as seen with the horizontal interface in the figure. The nodes do not match anymore since one element has more points than the other. After h-refining, interfaces can become geometrically non-conforming, as seen with the topmost interface in the figure. The nodes do not match because one element edge is shorter than the other. Finally, interfaces can be both geometrically and functionally non-conforming, as seen with the bottom interface in the figure. We will have to find a way to connect those elements in a way that retains the spectral convergence of the method. Such a method is described in Section 5.4.

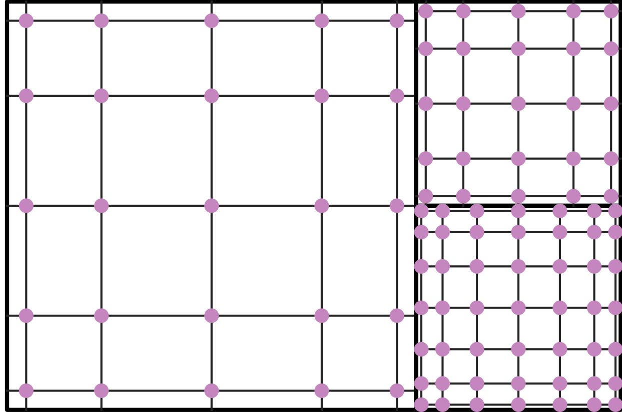
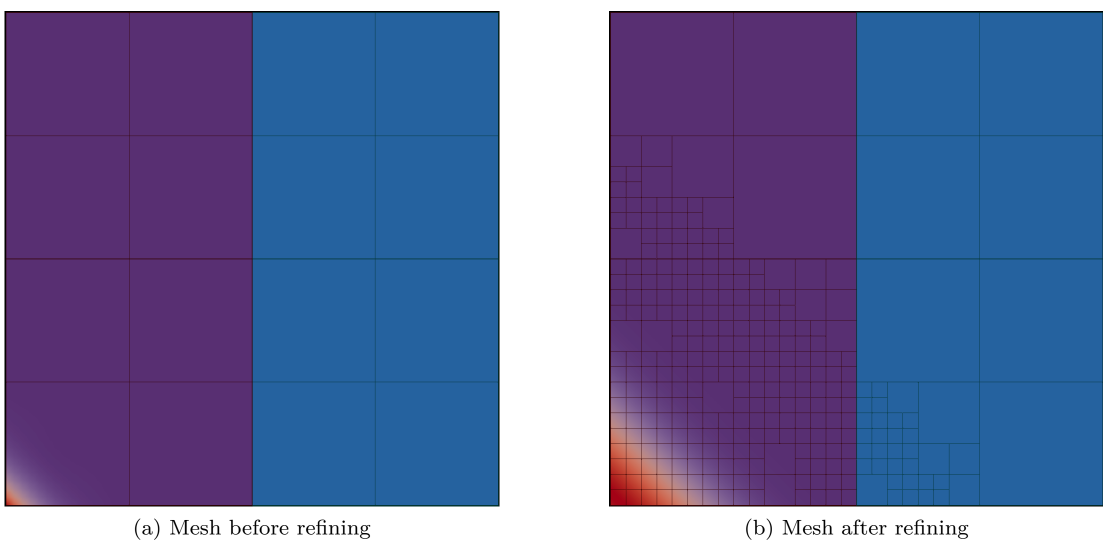


Figure 5.4: 2D hp-refinement: Refinement of elements creates non-conforming interfaces.

The implication of AMR in the context of high performance computing, the central challenge of this thesis, is the appearance of load imbalance and loss of computing efficiency on and between processors. For example, the mesh from Figure 5.1 is split into two blocks for two GPUs to work in parallel, as shown in Figure 5.5. Initially, both the left (purple) and right (blue) GPUs have the same number of elements in their domain. After refining, the left GPU has more elements as the wave is in its domain and more elements have split there. Elements will have to be sent from the left GPU to the right GPU in order to even out the computational load, while ensuring that there are as few interfaces as possible between the GPUs. Interfaces need to be minimised because the GPUs need to communicate at every timestep when computing fluxes on these interfaces. This involves transfers from the GPUs to their CPU, between

CPUs, and finally back from the CPUs to their GPU. This happens twice, once for each side of the interface. This is an expensive process, and improving locality of the elements will help reduce the runtime of that part of the computation. This is known as dynamic load balancing, and Chapter 6 is dedicated to solving this problem.



(a) Mesh before refining

(b) Mesh after refining

Figure 5.5: Load imbalance: The elements have split unequally in the two worker GPUs, denoted in purple and blue, one having a higher computational load. (a) Before refining (b) After refining

5.2 Error Estimation

We cannot simply refine everywhere in the domain at every occasion, since this is no better than having a finer mesh to start with. The mesh must only be refined where the expected error is higher. We therefore estimate the error of each element of the mesh, then refine only those elements which require better resolution.

As noted in Chapter 4, we represent the solution as an infinite series of polynomials. In this case we use the Legendre polynomials $L_n(x)$ of degree n, with the coefficients a_n being the modes of the solution.

$$u(x) = \sum_{n=0}^{\infty} a_n L_n(x) \tag{5.1}$$

It is of course impossible to use an infinite sum on computers, therefore we truncate this sum to N, the polynomial order of the solution. The remaining terms, denoted by τ, are the truncation error of the solution.

$$
\begin{aligned}
u(x) &= \sum_{n=0}^{\infty} a_n L_n(x) \\
&= \sum_{n=0}^{N} a_n L_n(x) + \sum_{n=N+1}^{\infty} a_n L_n(x) \\
&= \sum_{n=0}^{N} a_n L_n(x) + \tau \\
&\approx \sum_{n=0}^{N} a_n L_n(x)
\end{aligned} \tag{5.2}
$$

The norm of the truncation error provides an estimate of the overall error of the solution. To find it, we will need the modes a_n beyond N, which can be obtained by extrapolation of the known modes $a_n, n = 0 \cdots N$. The polynomials we use are orthogonal, therefore the following simple relation holds for the inner product of two of the polynomials.

$$
\begin{aligned}
(L_n, L_m) &= \int_{-1}^{1} L_n(r) L_m(r) dr \\
&= \frac{2}{2n+1} \delta_{nm}
\end{aligned} \tag{5.3}
$$

with δ_{nm} being the Kronecker delta. We start by computing the known modes by taking the inner product of the approximate solution $u_h(x)$ and the corresponding Legendre polynomial $L_n(x)$.

$$
\begin{aligned}
(u_h, L_n) &= \int_{-1}^{1} u_h(x(r)) L_n(r) dr \\
&= \sum_{k=0}^{N} a_k \int_{-1}^{1} L_k(r) L_n(r) dr \\
&= \frac{2}{2n+1} a_n
\end{aligned} \tag{5.4}
$$

Rearranging, we get the modes a_n.

$$
a_n = \frac{2n+1}{2} \int_{-1}^{1} u_h(x(r)) L_n(r) dr \tag{5.5}
$$

If we can obtain the truncated modes by extrapolation of our known modes a_n, the L_2 norm of the truncation error can then be approximated.

$$
\begin{aligned}
\|\tau\| \approx \tau_{est} &= \sqrt{\int_{-1}^{1} \sum_{n=N+1}^{\infty} \left(\widetilde{a}_n L_n(r)\right)^2 dr} \\
&= \sqrt{\sum_{n=N+1}^{\infty} \widetilde{a}_n^2 \int_{-1}^{1} L_n(r) L_n(r) dr},
\end{aligned}
\tag{5.6}
$$

where $\widetilde{a}_n$ are the extrapolated modes. We use the orthogonality of the Legendre polynomials shown in Equation 5.3 to obtain a simpler form of the truncation error estimate.

$$
\tau_{est} = \sqrt{\sum_{n=N+1}^{\infty} \frac{\widetilde{a}_n^2}{\frac{(2n+1)}{2}}}
\tag{5.7}
$$

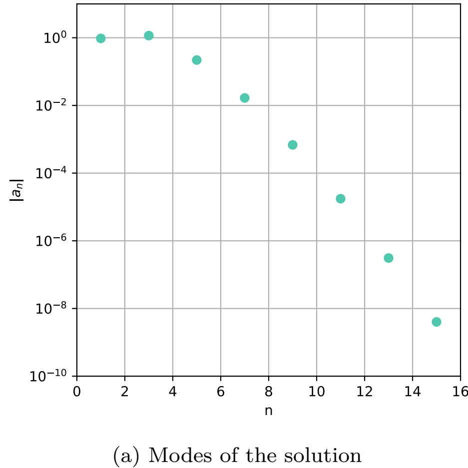
In 2D, the solution is a tensor product of polynomials, so the truncation error changes accordingly.

$$
\tau_{est} = \sqrt{\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}}}
\tag{5.8}
$$

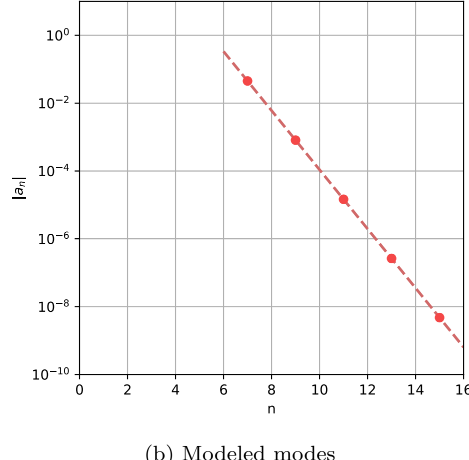
Furthermore, the decay rate of the modes provides an indication of the quality of the solution, in addition to a model for error estimation. For a smooth function, in this context this means a well-resolved solution, the modes a_n decay exponentially. As per [44], we model the exponential decay as follows

$$
a_n = Ce^{-\sigma n}
\tag{5.9}
$$

by least squares best fit to the existing calculated modes $a_n, n = 0 \cdots N$. Figure 5.6a shows an example with the modes of the smooth function $u = \sin(\pi x)$ with a polynomial order $N = 15$. It can be observed that the decay of the modes starts slowly, but becomes exponential after the first few modes. Figure 5.6b shows the modeled a_n, with the dashed line being the modeled exponential decay of slope σ in the log plot.



(a) Modes of the solution



(b) Modeled modes

Figure 5.6: Modes of the solution: We use the known modes and their decay rate to eventually extrapolate unknown modes. (a) Known modes (b) Modeled exponentially decaying modes

In order to fit a straight line to the data by linear least-squares, we want a linear relation between the decay rate σ and the modes. We take the logarithm of both sides of Equation 5.9 and get our linear relation.

$$\ln(a_n) = \ln(C) - \sigma n \tag{5.10}$$

We apply a linear least-squares fit to the last points of the known modes a_n. The slope of the line, σ, serves as the error refinement criterion. It will help choose a refinement strategy in Section 5.3. The linear least-squares fit finds a straight line by minimising the sum of the squares of the distances between the fitted points and the line. The resulting line is given by:

$$y = \alpha + \beta x, \tag{5.11}$$

and its coefficients are computed by:

$$\beta = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2}, \tag{5.12}$$

$$\alpha = \overline{y} - \beta\overline{x}, \tag{5.13}$$

with $\overline{x}$ and $\overline{y}$ being the averages of the x and y data respectively. The x data is the mode number $n = 0 \cdots N$, and the y data are the known modes in log form, $\log(a_n)$. The error decay slope becomes $\sigma = -\beta$, and the constant becomes $C = e^{\alpha}$.

Coming back to the error estimation, we can replace the sums of 5.7 and 5.8 by integrals

since the behaviour of the modes is known as an analytic function of n.

$$in\ 1D: \quad \tau_{est} = \sqrt{\sum_{n=N+1}^{\infty} \frac{\widetilde{a}_n^2}{\frac{2n+1}{2}}} \Rightarrow \sqrt{\int_{N+1}^{\infty} a_n^2 dn}, \tag{5.14}$$

$$in\ 2D: \quad \tau_{est} = \sqrt{\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\widetilde{a}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}}} \Rightarrow \sqrt{\int_{M+1}^{\infty} \int_{N+1}^{\infty} (a_{nm})^2 dndm} \tag{5.15}$$

By using the exponential decay model of Equation 5.9, the 1D case can be evaluated directly.

$$\begin{aligned} \tau_{est} &= \sqrt{\int_{N+1}^{\infty} a_n^2 dn} \\ &= \sqrt{\int_{N+1}^{\infty} \left(Ce^{-\sigma n}\right)^2 dn} \\ &= \sqrt{\frac{C^2}{2\sigma} e^{-\sigma(N+1)}}. \end{aligned} \tag{5.16}$$

The same method cannot be applied to the 2D case, as the modes in Equation 5.15 are defined in two dimensions. We will need to fit these modes in 1D to find the error estimate as in the 1D case. We use the same polynomial order in both x and y directions, and create 1D equivalent modes $\overline{a}_n$.

$$\overline{a}_n = |a_{n,n}| + \sum_{i=0}^{n-1} \left(|a_{i,n}| + |a_{n,i}|\right) \tag{5.17}$$

We use the equivalent modes to estimate the error.

$$\begin{aligned} \tau_{est} &= \sqrt{\int_{M+1}^{\infty} \int_{N+1}^{\infty} (a_{nm})^2 dndm} \\ &= \sqrt{\int_{N+1}^{\infty} \overline{a}_p^2 dp} \\ &= \sqrt{\int_{N+1}^{\infty} Ce^{-2\sigma p} dp} \\ &= \sqrt{\frac{C}{2\sigma} e^{-2\sigma(N+1)}} \\ &= \sqrt{\frac{C}{2\sigma}} e^{-\sigma(N+1)} \end{aligned} \tag{5.18}$$

The error estimator constants σ and C are computed from the 1D equivalent modes $\overline{a}_n$.

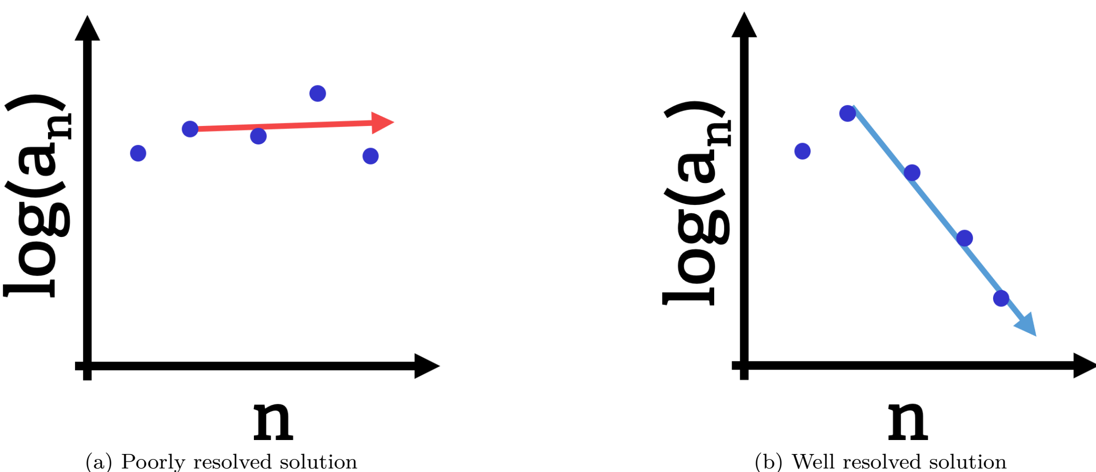
It is now possible to refine when the estimated error exceeds a certain threshold. We still

need to choose one of the refinement strategies from Section 5.1. Section 5.3 describes a method to choose the most appropriate strategy using the error decay rate σ computed previously.

5.3 Refinement Criteria

Once we compute the error estimate on every element and know which elements we want to refine, we must choose how to refine them. Recall from Section 5.1 that we can choose p-refinement, increasing the polynomial order N of an element, or h-refinement, splitting the element into smaller elements to increase the total number of elements K.

The method [44] used here is to look at the decay rate of the modes a_n of the solution. The error decay rate σ computed in Section 5.2 is the negative slope of the last few modes of the solution. A well-resolved solution should have a fast decay of the modes, with $\sigma > 1$. On the other hand, $\sigma \leqslant 1$ implies the solution is not well resolved as is. Figure 5.7 compares a poorly resolved solution whose modes do not decay fast to a well resolved solution whose modes decay fast.



(a) Poorly resolved solution (b) Well resolved solution

Figure 5.7: Modes decaying: Decay of spectral modes within an element and their best fit line. (a) $\sigma \leqslant 1$ (b) $\sigma > 1$

The truncation error is related to the value of the extrapolated modes. If the slope of the modes is steep, each added mode is expected to reduce the error significantly. This is because the truncated modes have successively much smaller values, therefore the truncated part of the solution is also small. On the other hand, a flat slope indicates that the value of the modes does not decay fast, and adding additional modes is not expected to reduce the truncation error much. This is often the case in non-smooth parts of the solution.

We choose the slope of the modes, σ, as our error refinement criterion. If $\sigma > 1$, adding polynomial modes is expected to reduce the error significantly, so we use p-refinement and

increase the polynomial order N of the element. If $\sigma \leqslant 1$, adding more modes is not predicted to reduce the error by much, therefore we use h-refinement and split the element into smaller elements, increasing K.

5.4 The Mortar Element Method

One issue that can arise from AMR is the presence of non-conforming interfaces. The flux calculations from Chapter 4 involve taking the extrapolated values of the solution of two elements at their interface, and computing the flux at points along the interface from the two adjacent elements. The elements can then use these fluxes to compute their derivative for that row or column. This works well when the interfaces are conforming, because the extrapolated solution values, at the intersection of the thin and bold black lines on Figure 5.8, line up between elements.

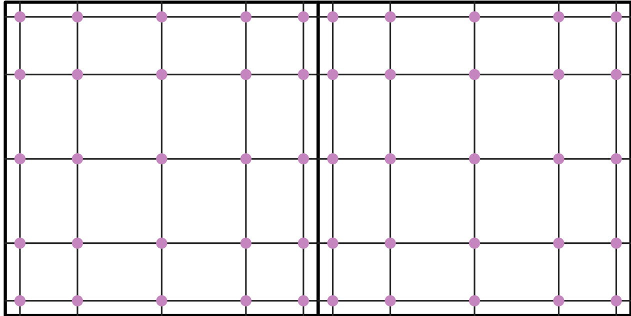


Figure 5.8: Conforming elemental interface: The extrapolated solution lines up between elements.

Once the mesh has been refined, the extrapolated values may not line up anymore, as seen in Figure 5.9. This makes it impossible to compute fluxes directly, as the flux computations need a left and right value at every point. These are non-conforming interfaces. There are three types of non-conforming interfaces. Geometrically non-conforming interfaces appear when two elements of the same polynomial order do not line up, either because they are not the same size or because they are offset. Functionally non-conforming interfaces appear when two elements line up but have a different polynomial orders. Finally, an interface can be both geometrically and functionally non-conforming.

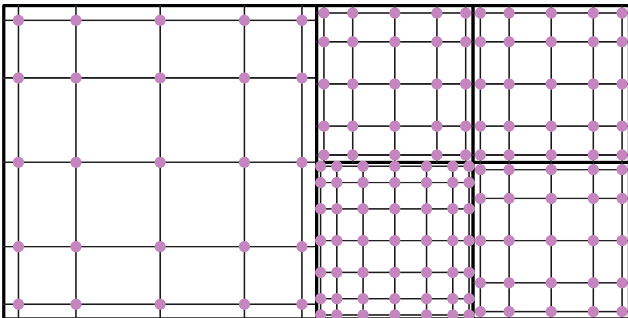


Figure 5.9: Non-conforming elemental interfaces: The extrapolated solution does not line up between elements.

We could simply interpolate values on the points that do not line up, but we want to conserve the fast convergence of the higher-order methods. We therefore use the mortar element method from [43]. This method adds mortar, or faces, between the elements. These faces have their own collocation points, unto which the extrapolated solution from the elements is projected. The flux computations can then be performed on the face's collocation points and be projected back to the elements afterwards.

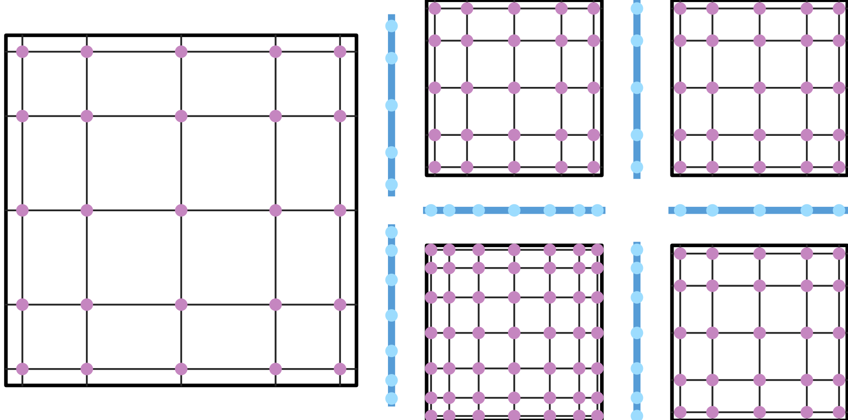


Figure 5.10: The mortar element method: Mortar is added between the elements, shown in blue.

We define the faces as the smallest of the element sides of a connection. This ensures each face only has one element on either side, their *left* and *right* elements. Elements can therefore have multiple faces on each of their sides. The functional of the face has the polynomial order of the highest-order of its neighbours. The next sections describe how to project to and from these new faces. The flux calculations are performed on the faces as described in Section 4.

5.4.1 Projection from Element to Mortar

Consider the projection between two elements and a face as shown in Figure 5.11.

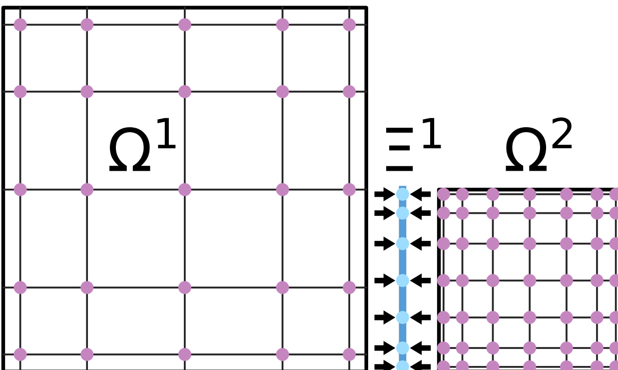


Figure 5.11: Element to mortar projection: The elements' boundary values are projected to the face.

Elements are denoted Ω and faces are denoted Ξ. We will use two coordinates, s for the element side coordinate and z for the face side coordinate. We project to the faces, therefore the face space will be used as a reference, $z \in [-1, 1]$. The coordinate change from element to face has the form $s = a + bz$, where a and b are scalars computed for each face side when generating the mesh. We can then define the element boundary solution U and its projection on the face Ψ.

$$U(s) = \sum_{k=0}^{N} U(s_k) l_k(s), \tag{5.19}$$

$$\Psi(z) = \sum_{k=0}^{J} \Psi(z_k) l_k(z) \tag{5.20}$$

with l being the Lagrange polynomials, N being the element polynomial order, and J being the face polynomial order. We then require that the integral of the error between the solution and its projection weighted by the Lagrange polynomials be zero.

$$\int_{-1}^{1} \left(\Psi(z) - U(a + bz)\right) l_j^{\Xi} dz = 0, \quad j = 0, \ldots, J \tag{5.21}$$

Incorporating Equation (5.19) and (5.20), we get a relation for the boundary and projected solutions.

$$\int_{-1}^{1} \sum_{m=0}^{J} \Psi(z_m) l_m^{\Xi}(z) l_j^{\Xi}(z) dz = \int_{-1}^{1} \sum_{k=0}^{N} U(a + bz_k) l_k^{\Omega}(a + bz) l_j^{\Xi}(z) dz, \quad j = 0, \dots, J \tag{5.22}$$

Or, in matrix form:

$$M\Psi = SU. \tag{5.23}$$

We can isolate the projected solution.

$$\Psi = M^{-1}SU \tag{5.24}$$

We use *Gaussian quadrature* to evaluate M and S. The orthogonality of the Legendre polynomials will be used to simplify the matrices.

$$l_m(z_k) l_j(z_k) = \delta_{mk}\delta_{jk} = \begin{cases} 1, & if \quad m = k = j \\ 0, & if \quad m \neq k \ or \ j \neq k \end{cases} \tag{5.25}$$

$$\begin{aligned} M_{jm} &= \int_{-1}^{1} l_m^{\Xi}(z) l_j^{\Xi}(z) dz \\ &= \sum_{k=0}^{J} l_m^{\Xi}(z_k) l_j^{\Xi}(z_k) w_k^{\Xi} \\ &= \sum_{k=0}^{J} w_k^{\Xi} \delta_{mk}\delta_{jk}, \qquad j, m = 0, \dots, J \end{aligned} \tag{5.26}$$

where w_k are the same Gauss-Legendre weights used in Chapter 4. We use Equation (5.25) to simplify the matrix M to a diagonal matrix.

$$M_{jm} = \begin{cases} w_j^{\Xi}, & m = j \\ 0, & m \neq j \end{cases} \tag{5.27}$$

The same can be done for S, with the following orthogonality relation:

$$l_j^{\Xi}(z_m) = \delta_{jm} = \begin{cases} 1, & if \quad m = j \\ 0, & if \quad m \neq j \end{cases} \tag{5.28}$$

$$
\begin{aligned}
S_{jk} &= \int_{-1}^{1} l_k^{\Omega}\left(a+bz\right) l_j^{\Xi}\left(z\right) dz \\
&= \sum_{m=0}^{J} l_k^{\Omega}\left(a+bz_m\right) l_j^{\Xi}\left(z_m\right) w_m^{\Xi} \\
&= \sum_{m=0}^{J} l_k^{\Omega}\left(a+bz_m\right) w_m^{\Xi} \delta_{jm}, \quad j=0,\ldots,J,\ k=0,\ldots,N.
\end{aligned} \tag{5.29}
$$

We use Equation (5.28) to simplify the matrix S.

$$
S_{jk} = l_k^{\Omega}\left(a+bz_j\right) w_j^{\Xi}, \quad j=0,\ldots,J,\ k=0,\ldots,N. \tag{5.30}
$$

With M and S, we can define a projection matrix from element boundary space to face space.

$$
\begin{aligned}
P_{jk}^{\Omega\to\Xi} &= M^{-1}S \\
&= \left(w_j^{-1}\right)^{\Xi} l_k^{\Omega}\left(a+bz_j\right)\left(w_j\right)^{\Xi} \\
&= l_k^{\Omega}\left(a+bz_j\right), \quad j=0,\ldots,J,\ k=0,\ldots,N.
\end{aligned} \tag{5.31}
$$

The projection operation from element to faces is therefore:

$$
\Psi = P_{jk}^{\Omega\to\Xi} U_k \tag{5.32}
$$

$$
= \sum_{k=0}^{N} l_k^{\Omega}(a+bz_j) U(a+bz_k), \quad j=0,\ldots,J. \tag{5.33}
$$

5.4.2 Projection from Mortar to Element

After projecting the element boundary solution to the faces, the fluxes can be computed on the faces. The result then needs to be projected back to the element boundaries to be useful in subsequent calculations. Figure 5.12 shows two faces projecting their fluxes to three elements.

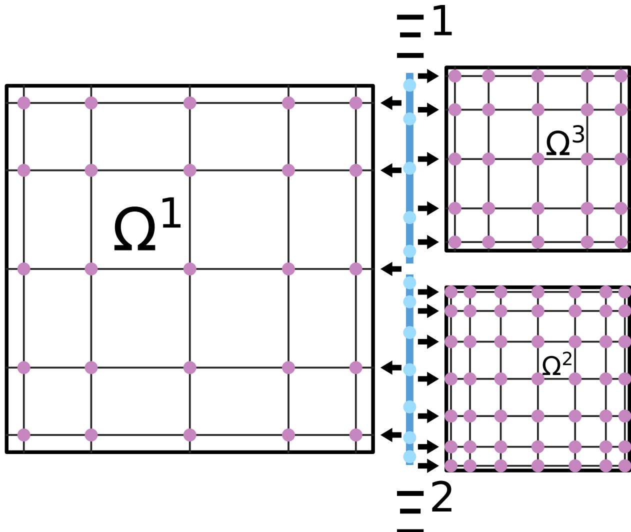


Figure 5.12: Mortar to Element projection: The faces' values are projected to the elements.

The projection from faces to elements is slightly more complicated than the other way around because we defined faces to have the size of the smallest elements they connect to. This means that while each face can only have one element on either side, elements can have multiple faces on each of their sides. The flux solution must then be projected from multiple faces onto each element edge.

This time, we take the element coordinates as a basis, $s \in [-1, 1]$. The coordinate change is the inverse of that of the element to face projection, $z = \frac{s-a}{b}$. Here Φ is the computed flux on the face, and F is the projection on the element side. As in the last subsection, we require that the integral of the error between the flux solution and its projection, multiplied by the test function, be zero. We also use the Lagrange polynomials as test functions.

$$\int_{-1}^{o'} \left(F(s) - \Phi(\frac{s-a}{b}) \right) l_j^{\Omega}(s) ds + \int_{o'}^{1} \left(F(s) - \Phi(\frac{s-a}{b}) \right) l_j^{\Omega}(s) ds = 0, \quad j = 0 \dots N \quad (5.34)$$

This looks very much like the condition we imposed for the element to face projection, except that the coordinate change is reversed and there are as many integrals as there are faces connecting to the element edge. In this case there are two, but there could be an arbitrary number. We can get the projection solution F out of each face's integral to form an integral over the whole element edge.

$$\int_{-1}^{1} F(s) l_j^{\Omega}(s) ds = \int_{-1}^{o'} \Phi(\frac{s-a}{b}) l_j^{\Omega}(s) ds + \int_{o'}^{1} \Phi(\frac{s-a}{b}) l_j^{\Omega}(s) ds. \quad (5.35)$$

We then rewrite the solutions in Lagrange form.

$$F(s) = \sum_{m=0}^{N} F(s_m) l_m^{\Omega}(s), \tag{5.36}$$

$$\Phi(\frac{s-a}{b}) = \sum_{m=0}^{J} \Phi(\frac{s_m - a}{b}) l_m^{\Xi}(\frac{s-a}{b}). \tag{5.37}$$

Incorporating those in Equation (5.35):

$$\begin{aligned} \int_{-1}^{1} F(s_m) l_m^{\Omega}(s) l_j^{\Omega}(s) ds = & \int_{-1}^{o'} \Phi(\frac{s_m - a}{b}) l_m^{\Xi}(\frac{s-a}{b}) l_j^{\Omega}(s) ds \\ & + \int_{o'}^{1} \Phi(\frac{s_m - a}{b}) l_m^{\Xi}(\frac{s-a}{b}) l_j^{\Omega}(s) ds. \end{aligned} \tag{5.38}$$

Or in matrix form:

$$MF = S\Phi. \tag{5.39}$$

We can again isolate the projected solution.

$$F = M^{-1} S\Phi \tag{5.40}$$

The M matrix is the same as for the element to face projection.

$$\begin{aligned} M_{jm} &= \int_{-1}^{1} l_m^{\Omega}(s) l_j^{\Omega}(s) ds \\ &= \sum_{k=0}^{N} w_k^{\Omega} \delta_{mk} \delta_{jk}, \qquad j, m = 0, \ldots, J \\ &= \begin{cases} w_j^{\Omega}, & j = m \\ 0 & j \neq m. \end{cases} \end{aligned} \tag{5.41}$$

The S matrix needs a change of reference from element space s to face space z in order to integrate on faces.

$$\int_{-1}^{o'} \Phi(\frac{s_m - a}{b}) l_m^{\Xi}(\frac{s-a}{b}) l_j^{\Omega}(s) ds = b \int_{-1}^{1} \sum_{m=0}^{J} \Phi(z_m) l_m^{\Xi}(z) l_j^{\Omega}(a + bz) dz \tag{5.42}$$

S then becomes:

$$
\begin{aligned}
S_{jm} &= b\int_{-1}^{1} l_m^{\Xi}(z) l_j^{\Omega}(a+bz)dz = b\sum_{k=0}^{J} l_m^{\Xi}(z_k) l_j^{\Omega}(a+bz_k) w_k^{\Xi}, \\
&= b l_j^{\Omega}(a+bz_m) w_m^{\Xi}, \quad j=0,\dots,N, \quad m=0,\dots,J.
\end{aligned} \tag{5.43}
$$

With M and S, we can finally define a projection matrix from face space to element boundary space.

$$
\begin{aligned}
P_{jm}^{\Xi\to\Omega} &= M^{-1}S \\
&= \left(w_j^{-1}\right)^{\Omega} b\, l_j^{\Omega}(a+bz_m) w_m^{\Xi} \\
&= b\, l_j^{\Omega}(a+bz_m)\frac{w_m^{\Xi}}{w_j^{\Omega}}, \quad j=0,\dots,N,\ m=0,\dots J.
\end{aligned} \tag{5.44}
$$

The projection operation from faces to elements is therefore:

$$
F = P_{jm}^{\Xi\to\Omega}\Phi_m \tag{5.45}
$$

$$
= \sum_{m=0}^{N} b\, l_j^{\Omega}(a+bz_m)\frac{w_m^{\Xi}}{w_j^{\Omega}}\Phi(a+bz_j), \quad j=0,\dots,J. \tag{5.46}
$$

5.5 Mesh Pre-condition

One type of error that cannot be alleviated by AMR is error caused by the inaccurate application of initial conditions. If the mesh is too coarse at the start of the computation, error is introduced in the solution since the mesh does not have enough resolution to adequately represent the initial conditions. Since this error is part of the solution, it does not reduce with refinement, it is simply projected to the refined elements and convected through the domain.

To alleviate this problem, we aim to refine the mesh before the main computation starts to a resolution high enough to capture the initial conditions correctly. We implement a simple “pre-condition” algorithm based on our AMR implementation.

5.5.1 Algorithm

The algorithm applies the initial conditions as usual, and runs the simulation for a set number of timesteps, usually the same as the AMR interval. At that point, the AMR algorithm is run. If the application of the initial conditions has introduced significant error to the solution, the error estimate will be higher in these regions and the mesh will be refined there by the AMR algorithm. To remove this error, the initial conditions are then applied again to this new refined mesh which should capture the initial conditions better. Care should be taken to perform a low number of iterations before refining, so as to refine the mesh close to the problematic areas at

the time of initial conditions application. This is one pre-condition step.

It is possible that the application of initial conditions is still not satisfactory after a single pre-condition step. In this case, multiple pre-condition steps can be performed, each advancing the solution a set number of timesteps, refining the mesh, and applying the initial conditions to that refined mesh. The flow of the algorithm is illustrated in Figure 5.13. Section 7.4 discusses the performance and solution quality of this algorithm.

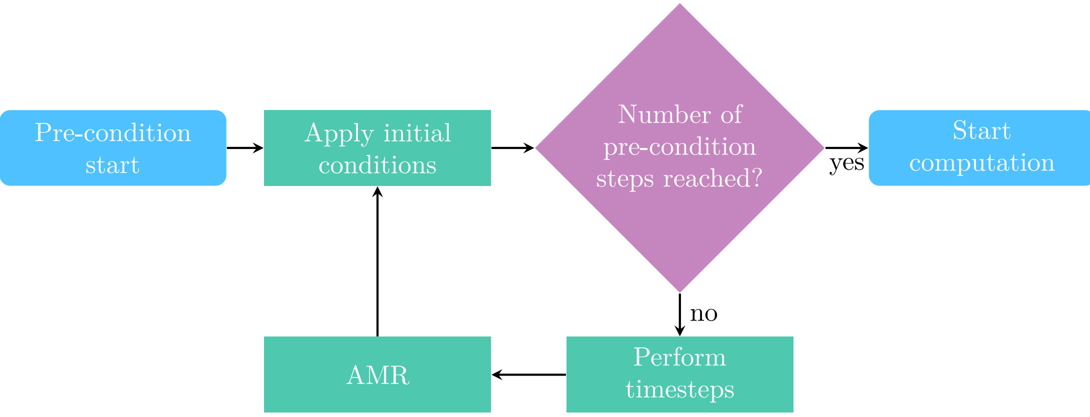


Figure 5.13: Flowchart of the pre-condition algorithm: Multiple iterative passes of the algorithm can be performed to ensure the initial conditions are well resolved.

5.6 Implementation

As detailed in Chapter 3, the grid is stored on the GPU as a flat vector of elements, each element having pointers to its solution arrays, stored in GPU dynamic memory. This ensures the elements have a fixed size regardless of their polynomial order. The element vector is allocated in GPU memory by the CPU using the CUDA runtime. Only this memory can be transferred between the CPU and GPU. Elemental solution arrays cannot be transferred between the CPU and GPU since they are allocated on the GPU. The faces are also stored in a vector on the GPU by the CPU using the CUDA runtime. Since the grid is unstructured, the elements and faces store the index of their neighbour faces and elements, respectively. This is only the position of the neighbours in their respective vector, acting as references so that multiple elements can refer to the same face and vice versa. The faces have one element on either side, and the elements can have one or multiple faces on each of their four sides.

With that structure in mind, adaptive mesh refinement must be able to add elements to the mesh, change the polynomial order of elements, replace elements that have split, create, replace and change the polynomial order of faces, and keep the element and face references coherent. The whole adaptivity process happens in parallel on each GPU, each on its mesh block. The adaptivity process starts by calculating the error in parallel on each element on the GPU. The elements store their estimated error τ_{est} and their error refinement criterion σ.

5.6.1 Counting refining elements

We start by finding out the total number of h-refining elements. A reduction [27] is computed on the GPU on a per-block basis, as described in Subsection 3.4.3. In this reduction each thread examines the estimated error τ_{est} of an element against a set threshold and marks it as refining or not. If refining, the error refinement criterion σ is read to mark it as h- or p-refining. If h-refining, the thread sets to 1 the value of an intermediate array at the thread's index within the block. Then, through a sequence of synchronisations and additions, the threads add up those values until the block total is known. The block total is then written to an output vector at the block's index within the mesh.

Once this is done, the array with each block's total number of h-refining elements is copied to the CPU and the total number of splitting elements is computed by adding all the values. Although it would be possible to launch this kernel again with fewer blocks on the resulting vector to reduce further to fewer values, this would not fill the GPU. Having the sum be defined with the same blocks as the elements will help us later when we want to know where to place new elements in the element vector.

If no elements split in the mesh, we can skip the next steps and only p-refine the elements in place. In this case, a kernel is launched on the GPU and p-refining elements will resize their solution arrays and project their solution onto the new array. The faces can then increase their polynomial order if either of their neighbour elements has increased their polynomial order beyond that of the face.

5.6.2 Moving elements

If any element h-refines, it will create more elements and the element vector will need to be resized. A new vector is created in GPU memory by the CPU with the new increased size. A prefix sum vector is created for each block of threads, indicating how many elements have split before it. This tells threads where to put existing and new elements in the new vector. A kernel is launched on the GPU, and all elements are assigned to threads and will either move, p-refine or h-refine. Figure 5.14 shows a mesh with six elements refining, with two elements being h-refined in red.

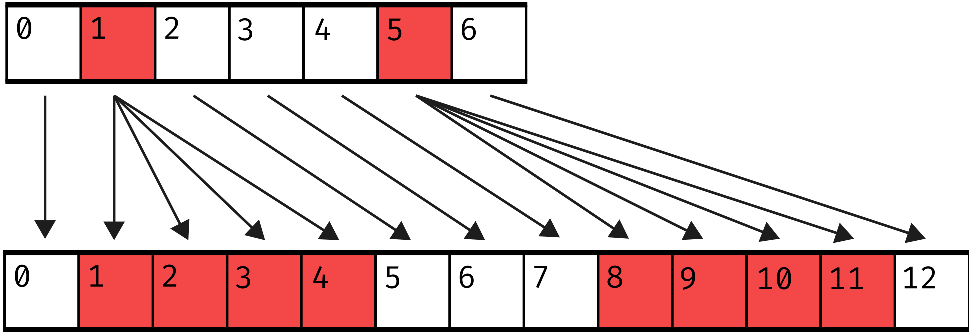


Figure 5.14: Moving elements from the old vector to the new: Two elements are h-refined (in red), the others are moved.

Non-refining elements are simply moved to their new index in the new vector. This is not a very costly operation because the element structure only contains geometric data, error data and pointers to solution data. The solution data itself, representing most of the element's size, is stored in dynamic memory on the GPU. It does not need to be copied, only the pointer to it needs to be copied in the new element vector.

p-refining elements act a bit like non-refining elements, except that they need to create new solution arrays with their new polynomial order and project their solution to it.

h-refining elements create four smaller elements in the new vector. They are placed consecutively at the new index of the splitting element, in an order determined by the space-filling curve (SFC) used, detailed in Chapter 6. The solution of the splitting element is projected onto the smaller elements. Four associated faces have to be created to connect the smaller elements to each other, as well as a node in the center.

Once the elements have been moved, their index has changed. All faces need to correct their neighbour elements' indices to reflect their new indices. The faces use the same block prefix sum as the elements to find the new index of their neighbours.

This section is prone to race conditions. The new element indices must be computed in parallel to be efficient, and cannot loop over all elements to find their offset. This would be very slow, and data residing in other blocks of threads is not as easily accessible. This is where the prefix sum vector enters. Each thread will know the offset of its block and only need to add the additional number of elements splitting before it within its block, which is accessible in block-shared memory. This ensures no threads read or write the same memory location, and can all move their elements directly to the correct location concurrently.

5.6.3 Moving faces

Faces are refined similarly to elements. If an element on either side of a face splits while the face covers the whole element side, the face will split too. This ensures that the face always

has the length of the smallest of the two elements it connects to.

Unlike elements, faces do not need any specific ordering as described in Section 6.1. This simplifies the refinement process for faces, as moving faces keep the same index and splitting faces add a face at the end of the vector. Faces also increase their polynomial order if needed. Figure 5.15 shows an array of faces being refined, with the splitting faces in red adding faces to the end of the new array. A prefix sum for each block of threads is also computed for faces to permit them to split in parallel without race conditions.

The splitting faces become two faces each, one at their initial index and another at the end of the array. Elements must then check their neighbouring faces for splitting, and update the face indices as needed.

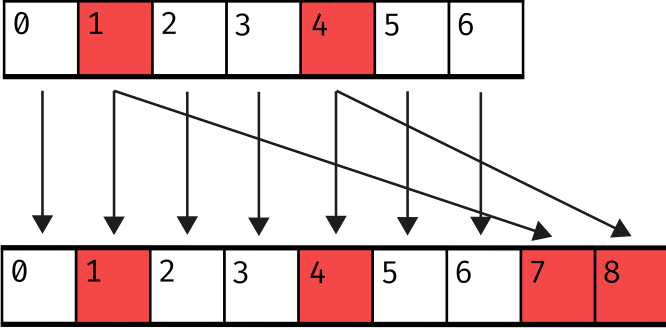
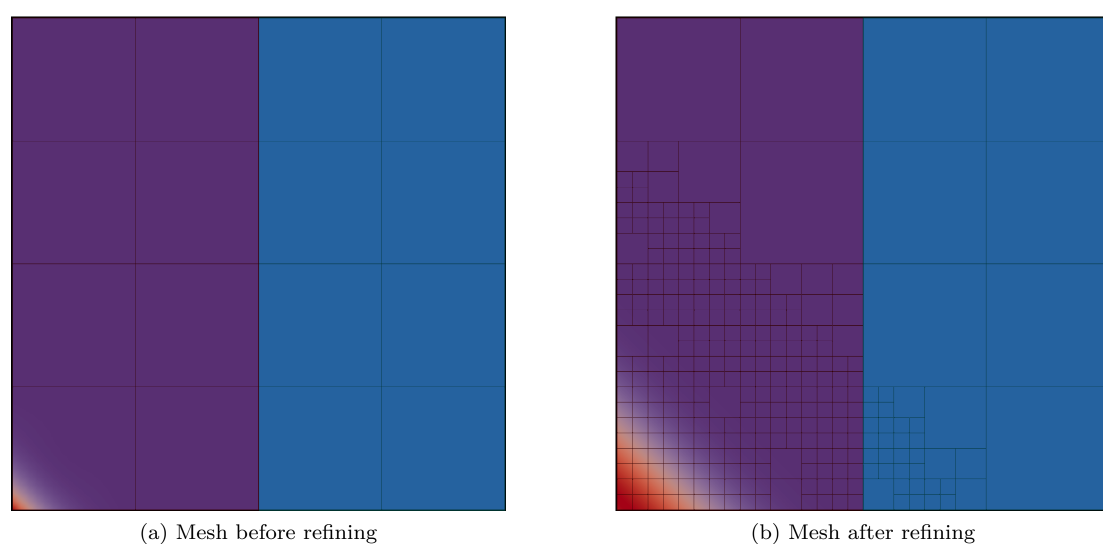


Figure 5.15: Moving faces from the old vector to the new: Two faces split (in red), the others are moved.

Chapter 6

Load Balancing

A well made multi-block mesh can distribute the work evenly between the different worker processes without needing runtime adjustment if the topology and areas of more expensive computation stay the same during the whole computation. This is complicated by the fact that, as seen in Chapter 5, the mesh elements can increase their polynomial order and split into multiple smaller elements in areas where the solution accuracy is estimated to be unsatisfactory. Unless the regions of interest happen to be evenly distributed between the different processes, this will lead to an imbalance as some processes are left with more elements, or higher-order elements, within their domain as illustrated in Figure 6.1.



(a) Mesh before refining

(b) Mesh after refining

Figure 6.1: Load imbalance: The elements have split unequally in the two worker GPUs, purple and blue, one having a higher computational load. (a) Before refining (b) After refining

The wall clock time of the simulation will be driven by the most heavily loaded process,

as the processes have to synchronise at each time step. The more lightly loaded processes will simply wait for the other to finish computing.

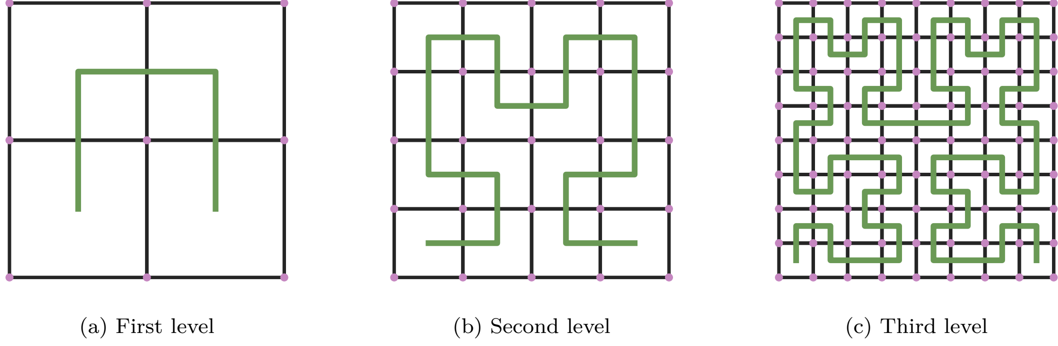
In order to maintain good performance as the mesh is refined, we will need to perform dynamic load balancing. Dynamic load balancing seeks to even out the computational load between the different processes. The algorithm for load balancing needs to be fast, to be executed often and to keep the mesh optimal for a longer period without overshadowing the solution time. The worker processes in this work use GPUs for their computations. Because transfers between GPUs are expensive, the algorithm needs to limit transfers both during the load balancing process and during computation afterwards. The workers are made up of one CPU core and one entire GPU. This means that the worker's processing power is mostly generated by the GPU, and the algorithm needs to use it as much as possible. The algorithm will need to have as many parts as possible running on the GPU, in parallel. Finally, the algorithm needs to use as little additional GPU memory as possible, as these processors have limited memory, especially compared to CPUs.

The algorithm needs to select which elements to send from one GPU to another. Many such algorithms exist, notably graph-based algorithms [34] and one-dimensional algorithms [61], often called *chains-on-chains partitioning (CCP)*. Chains-on-chains partitioning will be used here for its relative speed. This will require a scheme to transform our domain from two-dimensional space to one-dimensional space, as required by the algorithm.

Here, the repartitioning scheme uses *space-filling curves (SFCs)*, more specifically the Hilbert Curve. SFCs map multi-dimensional space to one-dimensional space. This space can be partitioned in segments, and upon change of the size of these segments elements can then be sent or received from the ends of these segments. In addition, these curves are fast to generate, use low memory, and maximise locality. Elements that are close in curve space will also be close by in the space used for computations. This limits the number of interfaces between processes.

6.1 Hilbert Curve

The Hilbert curve, developed by D. Hilbert [32], is a specific kind of space-filling curve as described by G. Peano [59]. This curve works for 2D domains with equal and power of two resolution in x and y. Some newer research [28] shows how such curves can be expanded to three dimensions, and arbitrary domains.



(a) First level

(b) Second level

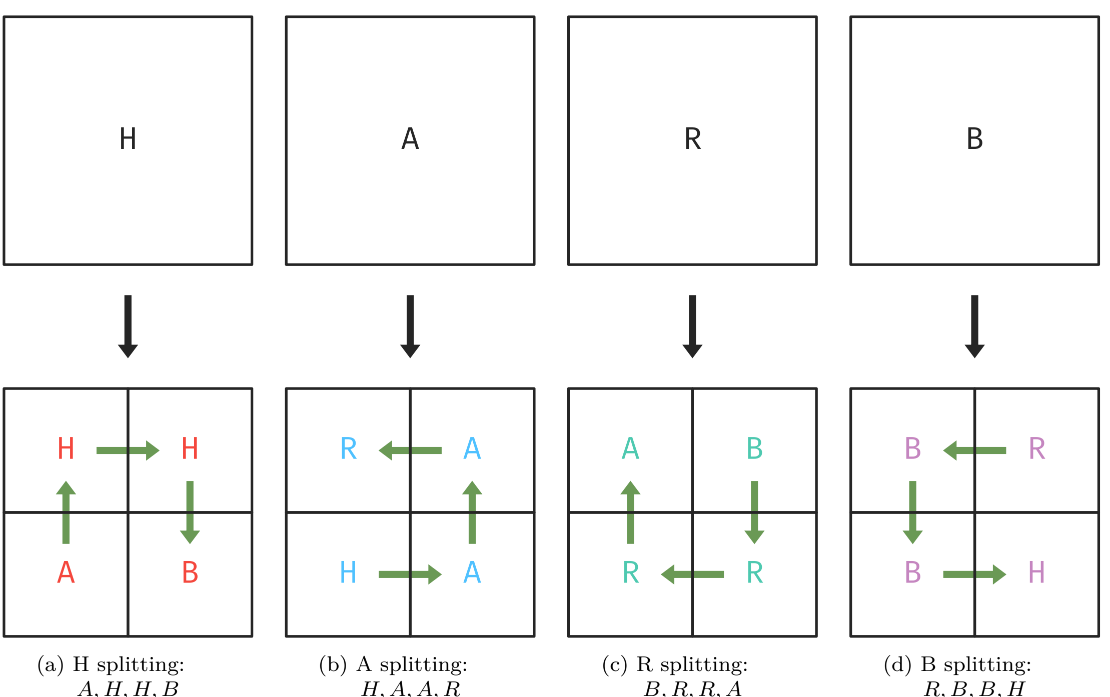
(c) Third level

Figure 6.2: Hilbert curve: The first thee Hilbert curves. (a) 2×2 (b) 4×4 (c) 8×8

Figure 6.2 shows the first three levels of the Hilbert curve. The curve successfully maps our 2D domain to a 1D one along the curve. That 1D domain can then be partitioned between the different worker GPUs. As illustrated in the figure, the elements have good locality and no jumps exist. Wherever the curve is cut, elements on the resulting segments are close together. This is the first desirable property of the curve for this program, as we aim to reduce the contact area between the mesh blocks dispatched to each GPU. The iterative nature of those curves is also apparent when put side by side. Each increasing level of the curve follows the general path of the previous curve. Iteration is one of the possible ways to generate this curve. The Hilbert curve is used to generate the initial meshes used by the program, as well as to re-number elements when the mesh refines.

6.1.1 Generation

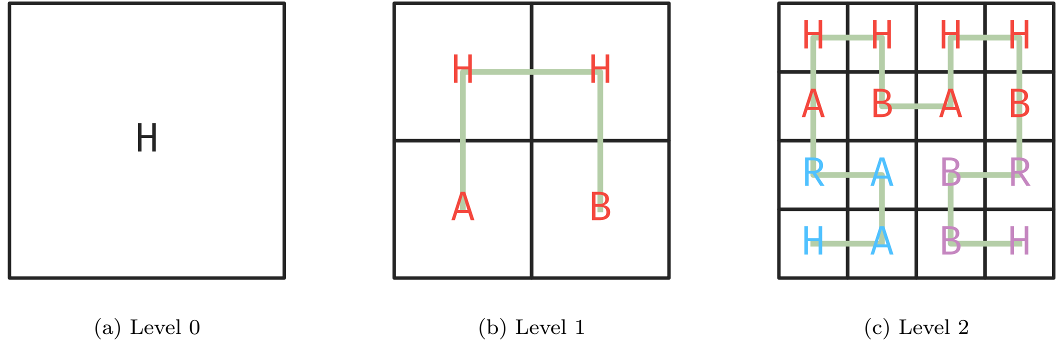
We choose a table-driven algorithm to generate meshes. Each element has one of four possible *states* s: $s \in \{H, A, R, B\}$. The state of an element distinguishes the local pattern of the curve inside the element. This state determines the state and ordering of the four children elements obtained when increasing the level of the curve. The four states and their resulting children are shown in Figure 6.3.



(a) H splitting: A, H, H, B (b) A splitting: H, A, A, R (c) R splitting: B, R, R, A (d) B splitting: R, B, B, H

Figure 6.3: The four states splitting, with their children's state and ordering.

When put together, and assigning H as the first state of the mesh, the mesh can be iteratively constructed to the required level as shown in Figure 6.4.



(a) Level 0 (b) Level 1 (c) Level 2

Figure 6.4: First levels of the Hilbert curve with statuses s ($s \in \{H, A, B, R\}$).

This construction can be summarised in Tables 6.1 and 6.2. The ordering of the children elements is from the bottom left element, in counter-clockwise order as shown in Figure 6.5.

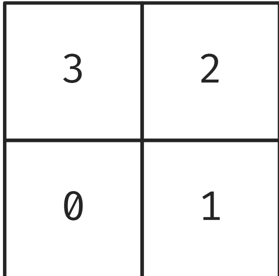


Figure 6.5: Child numbering: The $j = 0, 1, 2, 3$ children of an element, used to assign states and ordering.

These tables are implemented directly in the code, with the state defined as an enumeration from 0 to 3 and the tables as arrays. The state is used as an index for the arrays to get the resulting ordering and states. This gives the state $S(s, j)$ of the jth child element of a parent element as a function of the state s of the parent element. The same goes for the ordering p of the children, $P(s, j)$.

$S(s,j)$	$j = 0$	$j = 1$	$j = 2$	$j = 3$
$s = H$	A	B	H	H
$s = A$	H	A	A	R
$s = R$	R	R	B	A
$s = B$	B	H	R	B

Table 6.1: Children element state table: child state S, with s being the state of the parent element and j being the jth child.

$P(s,j)$	$j = 0$	$j = 1$	$j = 2$	$j = 3$
$s = H$	0	3	2	1
$s = A$	0	1	2	3
$s = R$	2	1	0	3
$s = B$	2	3	0	1

Table 6.2: Children element order table: child order P, with s being the state of the parent element and j being the jth child.

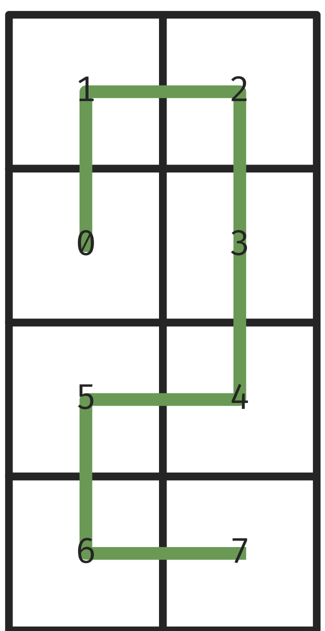
This table-driven approach has the added benefit of working seamlessly in parallel, as each element can generate its children using nothing more than its own order.

Without adaptive mesh refinement, no more work is needed. The elements do not even need to store their state, as the compact numbering of the elements is the result of the Hilbert curve. The mesh generator included with the program works this way, creating a uniform power of two sized 2D mesh numbered according to the Hilbert curve. The mesh generator uses the standard CGNS format. The meshes generated in this way can be used directly by the program

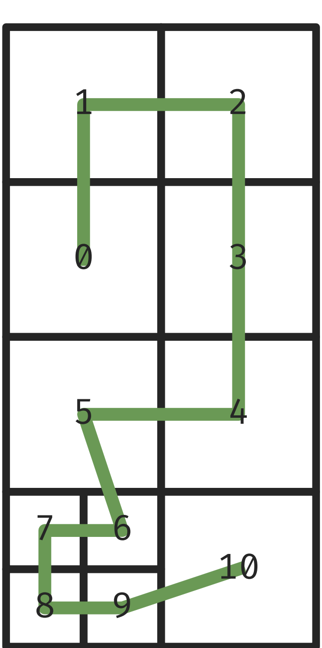
on a single GPU, or split into multiple blocks using the provided mesh partitioner. The mesh partitioner can split any single-block CGNS mesh into a multi-block mesh, which can then be used by the main program running on multiple GPUs.

6.1.2 Refinement

The same iterative process is used to refine elements. The parent state dictates the state and ordering of its children. Correct ordering is critical to maintaining good locality between elements as the mesh refines and non-conforming interfaces are created. Figure 6.6 shows a mesh block refining, where the element 6 splits into four smaller elements. The ordering of the children elements must follow the Hilbert curve to maintain the locality of the mesh.



(a) Before refining



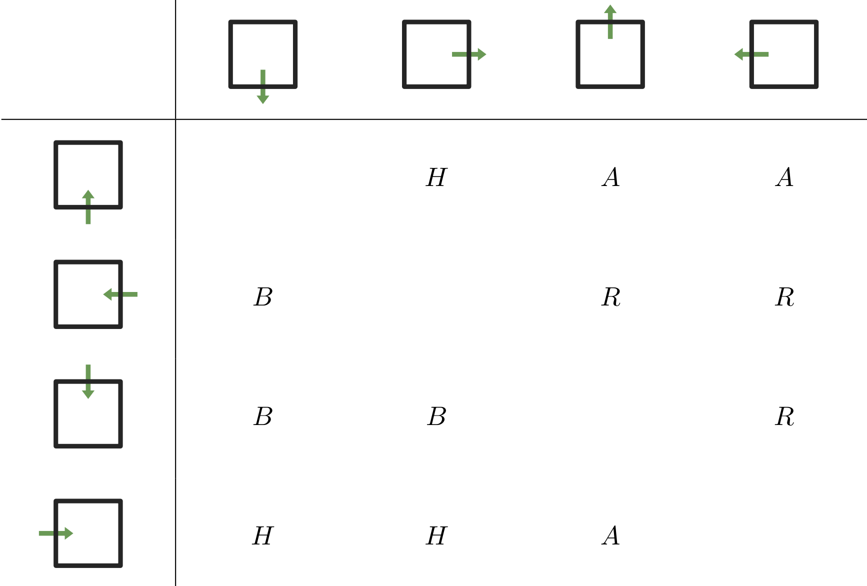
(b) After refining

Figure 6.6: Mesh refinement Hilbert curve: One element refining. (a) Before splitting (b) After splitting

This process assumes knowledge of each element's status. However, the main solver program is designed to accept standard CGNS meshes as an input. The CGNS format does not provide

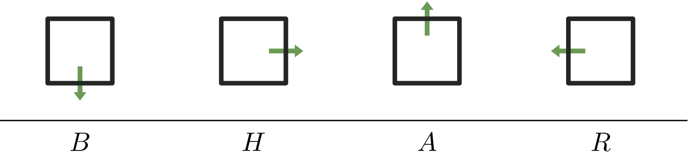
status information. Furthermore, the program can accept any 2D unstructured mesh using the format, which may not be numbered according to a Hilbert curve.

It is therefore necessary to deduce the status of elements when reading a mesh file. Figure 6.4c gives a hint of a possible way to do this. For an element k, the curve goes from element $k-1$ to element k, and then from element k to element $k+1$. When entering element k from element $k-1$, the curve will pass through one of the four sides of element k. The same goes for the curve leaving element k towards element $k+1$. Following the curve, each combination of entrance side and exit side corresponds to a single state. For example, the curve entering from the left side of the element and exiting through the top side always has the state A. This is confirmed by examining other levels of the Hilbert curve. The same can be deduced for the first and last elements, using only the exiting and entering side, respectively. Using this information, we can create a 2D matrix of state as a function of entering and exiting side for elements (Table 6.3), and 1D matrices for the first and last elements (Tables 6.4 and 6.5).



		H	A	A
	B		R	R
	B	B		R
	H	H	A	

Table 6.3: Element state deduction table: Element state as a function of curve entrance and exit side.



B	H	A	R

Table 6.4: First element state deduction table: Element state as a function of curve exit side.

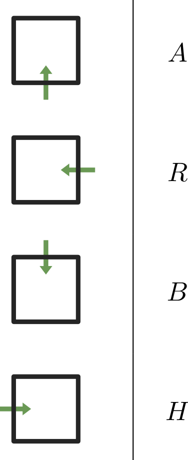
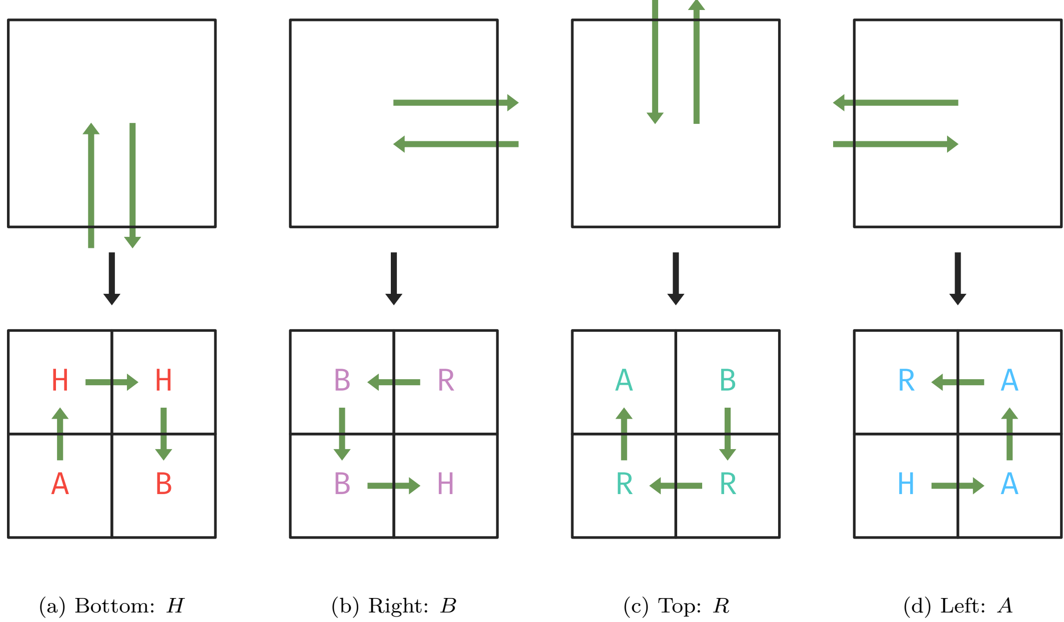


Table 6.5: Last element state deduction table: Element state as a function of curve entrance side.

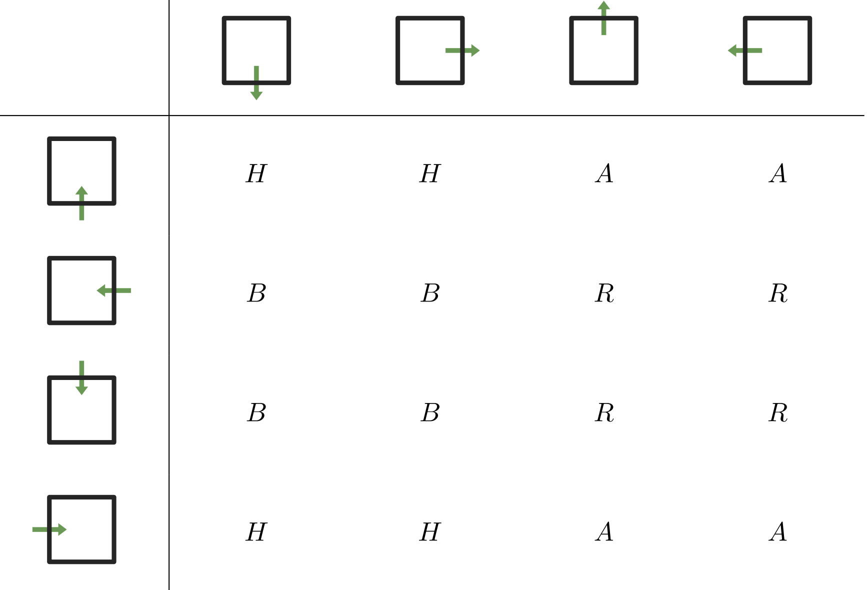
A few combinations are still undefined in the matrix. They represent the cases where the curve enters and leaves through the same side of the element. These cases should not occur with correct Hilbert curves. However, nonconforming geometries created while refining, or meshes created without using a Hilbert curve, may generate those geometries. Examining Figure 6.3, we observe that for each status the first and last child elements occupy the same side. We match that side in Figure 6.7 with the incoming and exiting sides of the undefined cases.



(a) Bottom: H (b) Right: B (c) Top: R (d) Left: A

Figure 6.7: The four missing combinations: Each corresponds to a single state.

We can then obtain the full matrix, given in Table 6.6.

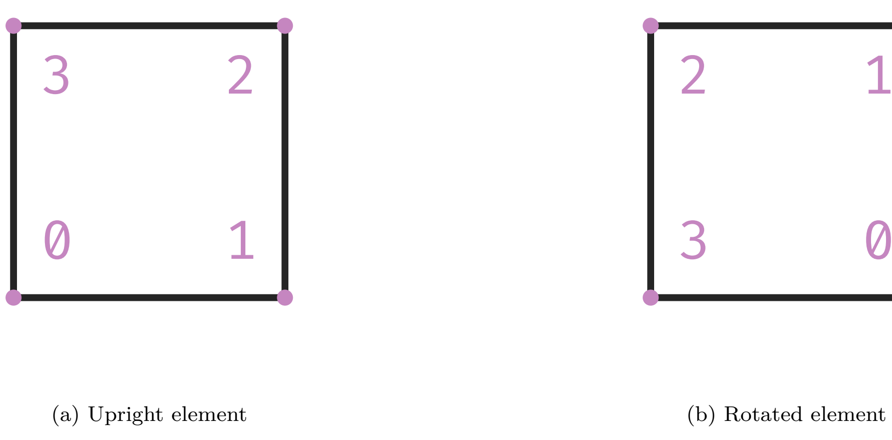


	H	H	A	A
	B	B	R	R
	B	B	R	R
	H	H	A	A

Table 6.6: Element state deduction table: Element state as a function of curve entrance and exit sides.

Until now, we assumed the elements were axis-aligned, and numbered such that their first

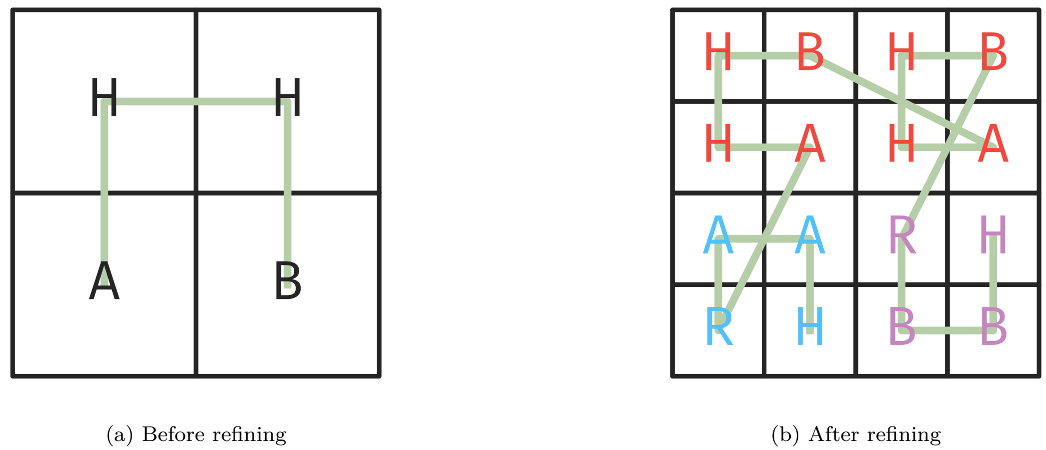
edge is at the bottom. This is important, as the elements number their children from their bottom left, in a counter-clockwise order, shown in Figure 6.8a. If an element's first edge is not at the bottom as in Figure 6.8b, the wrong ordering will be given to refining elements, as in Figure 6.9b. Figure 6.9b does not match the correct Hilbert curve in Figure 6.4c. Since elements can be oriented in any direction in meshes, this situation is expected to happen. The following examples use elements numbered such that their first side is not at the bottom, but at the right, as shown in Figure 6.8b.



(a) Upright element

(b) Rotated element

Figure 6.8: Element rotation: The numbering of the nodes of an element can affect its refinement. (a) Expected orientation (b) Unexpected orientation



(a) Before refining

(b) After refining

Figure 6.9: Refinement with rotated elements: The numbering is wrong when compared to the correct Hilbert curve from Figure 6.4c. (a) Initial mesh (b) Refined mesh

One might be tempted to deduce the element status in the local element referential instead.

Figure 6.10 shows the same rotated element in the global referential and in its local referential. Note that the entering and exiting sides of the Hilbert curve are not the same, which will change the deduced state. Figure 6.11b shows the result of one step of refinement using the local referential instead of the global referential. Figure 6.10b shows that even though there are no jumps, the correct Hilbert curve shown in Figure 6.4c is not attained.

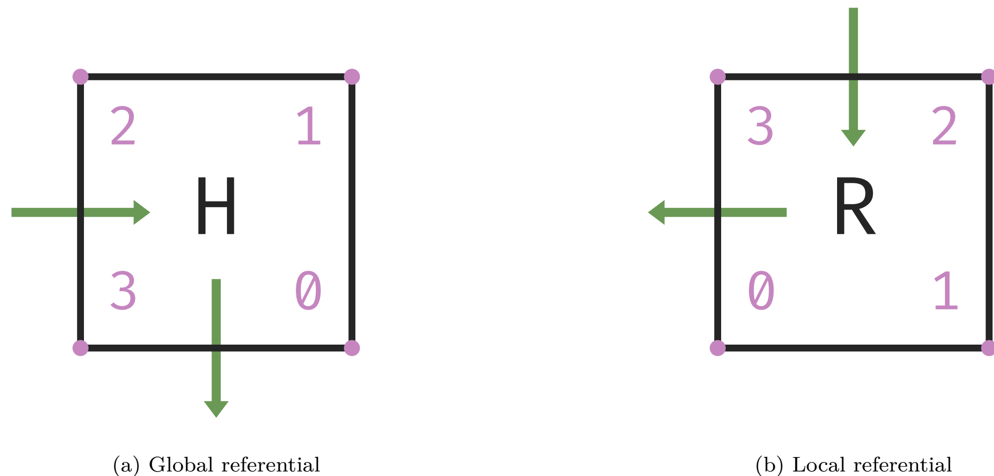


Figure 6.10: Different referentials: The different referentials will change the deduced state. (a) H (b) R

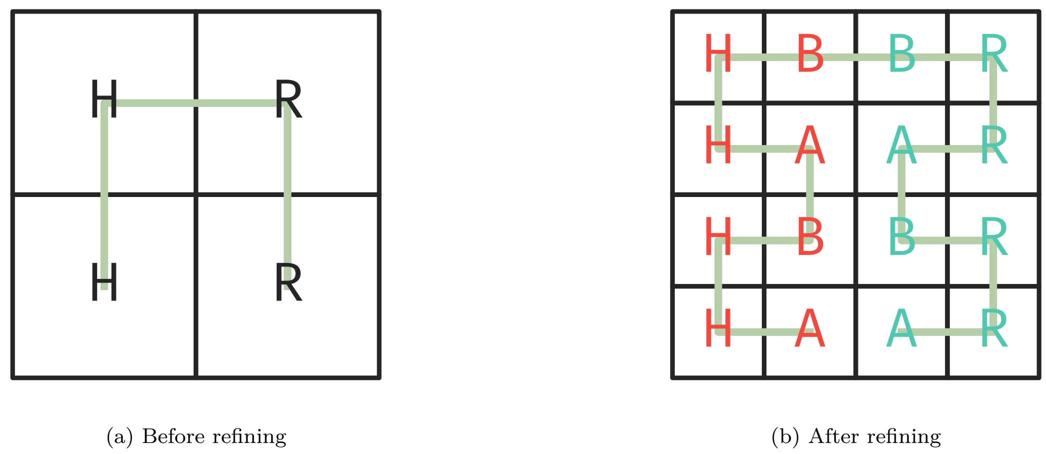
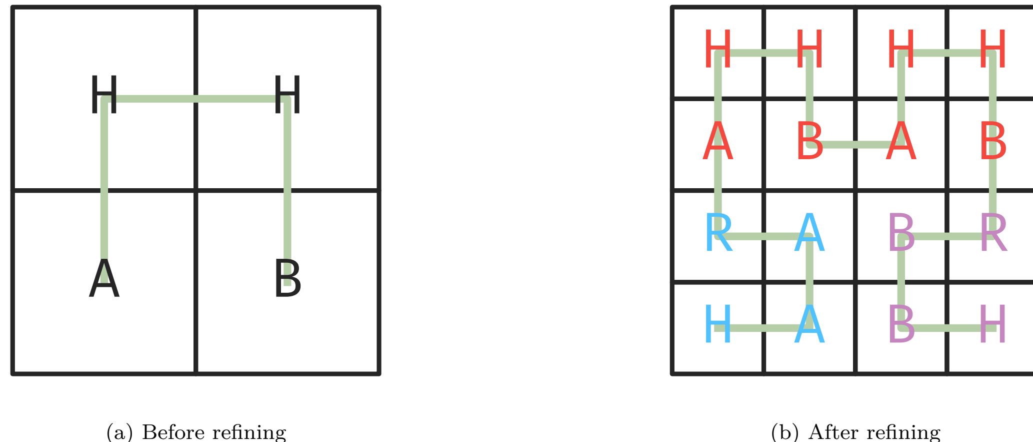


Figure 6.11: Refinement with rotated elements using local referential: This is not the next level curve, as seen in Figure 6.4c. (a) Initial mesh (b) Refined mesh

Both methods give wrong results when the elements are rotated compared to the first edge at the bottom case. The solution is to compute the entering and exiting sides in global coordinates, depending on the direction the curve goes in x and y. When splitting, the elements

use their rotation as an offset when numbering their children elements. The rotation of the elements is defined as the number of quarter turns the element is rotated compared to the normal case, with the first edge at the bottom. The refined mesh using this method is shown in Figure 6.12b and now has the correct Hilbert curve as in Figure 6.4c.



(a) Before refining

(b) After refining

Figure 6.12: Refinement with rotation offset: The next level Hilbert curve is correct. (a) Initial mesh (b) Refined mesh

With this out of the way, it is possible to compute the rotation when the mesh is read, store it as a member of the element class, and use it whenever the element splits to get the correct ordering. The different elements split independently in parallel, and the Hilbert curve keeps the ordering coherent. An element gets the new indices of its children by knowing how many elements split before it on the curve to give a starting index and then using the table to order the individual children.

The mesh can be refined multiple times that way and the locality of the Hilbert curve can be preserved. Even non-confirming interfaces do not create any jumps in the curve. Figure 6.13 shows a mesh of initially $K = 4 \times 4$ elements refined and load balanced three times. The Hilbert curve is shown in green to assess the quality of the ordering.

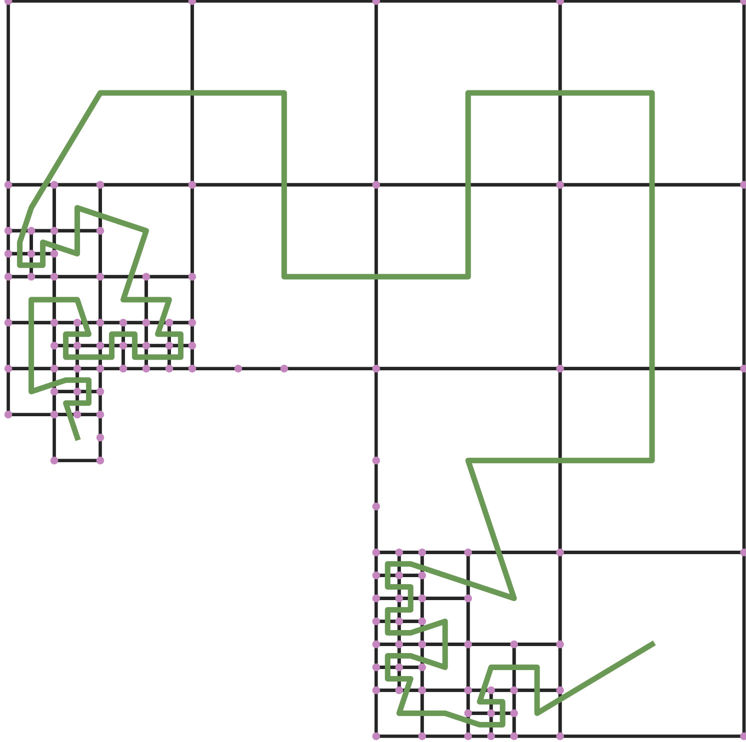


Figure 6.13: Mesh refining: The elements follow mixed levels of the Hilbert curve, without jumps or discontinuities.

6.2 Workload Leveling

Now that the problem can be partitioned in one dimension, we can begin redistributing elements to level out the workload between the different worker GPUs. We use chains-on-chains partitioning, where a linear list of N tasks i have associated weights w_i. This list of tasks is then split into P parts, equal to the number of workers P. Each worker p has a capacity c_p, indicating the relative workload it can execute.

To repartition, each worker sums the total weight of the I_p tasks it contains, w_p.

$$w_p = \sum_{i=0}^{I_p} w_i \tag{6.1}$$

The total amount of work in the problem W is obtained by adding up the total weights of each worker.

$$W = \sum_{p=0}^{p} w_p \tag{6.2}$$

Similarly, the total capacity of the system is summed.

$$C = \sum_{p=0}^{P} c_p \tag{6.3}$$

The ideal workload $w_{ideal,p}$ of a worker is given by:

$$w_{ideal,p} = W \frac{c_p}{C}. \tag{6.4}$$

The load imbalance L of the system is defined as the maximum of each worker's load imbalance l_p. A worker's load imbalance is the ratio between its current workload and its ideal workload:

$$l_p = \frac{w_p}{w_{ideal,p}}, \tag{6.5}$$

$$L = \max_{0 \leq p \leq P} (l_p). \tag{6.6}$$

The parallel load efficiency E is the reciprocal of the load imbalance:

$$E = \frac{1}{L}. \tag{6.7}$$

The new starting task $i_{0,p}$ of the worker is given by the new total weight of the previous workers.

$$i_{0,p} = \sum_{j=0}^{p-1} w_{ideal,j} \tag{6.8}$$

Tables 6.7 and 6.8 provide an example of repartitioning between four workers using this method.

worker p	0				1				2				3			
capacity c_p	2				2				1				1			
global id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local id i	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
weight w_i	1	1	1	1	2	2	3	2	2	1	2	1	1	1	2	1
total weight w_p	4				9				6				5			
ideal weight $w_{ideal,p}$	8				8				4				4			

Table 6.7: Problem before repartition: The workers have uneven weight.

We repartition the problem, so that the workers have a weight closer to their ideal weight.

worker p	0						1				2			3		
capacity c_p	2						2				1			1		
global id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local id i	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
weight w_i	1	1	1	1	2	2	3	2	2	1	2	1	1	1	2	1
total weight w_p	8						8				4			4		
ideal weight $w_{ideal,p}$	8						8				4			4		

Table 6.8: Problem after repartition: The workers have a better workload distribution.

This is the more general approach to repartitioning. Knowing our problem, we can simplify in two ways. Tasks are elements, and workers are GPUs. Firstly, since we do not work in mixed systems, all workers have equal capacity. In typical HPC systems, nodes have multiples of the same model of GPU. Since the program only executes on GPUs, all the workers will therefore be the same. See Section 8.3 for proposals of a solver that has both GPU and CPU workers in order to fully utilise supercomputer nodes. Such a system would necessitate the use of different capacities, since these nodes have many more CPU cores than GPUs, while each GPU is much faster.

Secondly, each element will be given equal weight. This greatly simplifies the computing of total weight within a GPU, since it becomes directly related to the number of elements. It also simplifies transfers between processes, as now only the number of elements in each process is needed to compute which elements need to be sent or received. This assumes that each element has the same computational complexity regardless of any factor, like its polynomial order. Of course, this is not true. However, the computational cost of elements was not observed to vary significantly for relatively low order polynomials. When the polynomial order is very different between elements, the increased number of collocation points to compute and the increased memory needed to store the solution start increasing the computation time for those elements. If this proves to be a significant problem, Section 8.3 discusses using weights for elements, and a way to relate polynomial order to the weight.

6.3 Reconstruction

Now that we have a way to repartition work, the reconstruction of the mesh can begin. Each GPU worker shares its number of elements with *MPI_Allgather*, so that every worker knows how many elements each worker currently has. They can then calculate the global id of their first and last element. Using the total number of elements in the mesh divided by the number of workers, they can compute their new number of elements, and their new first and last global element id. Any element they have that is out of this new range, they send. Knowing the new number of elements per process, it is easy to compute to which process the elements need to be sent based on their global element id. Any element missing from their new range, they need to receive. Again, using the previous number of elements per process, it is easy to compute from which process these elements should come.

The knowledge of which elements to receive and send with which process is used to set up MPI transfers. All the data is sent and received in a non-blocking way with *MPI_Isend* and *MPI_Irecv*, and once these transactions are completed the mesh can be reconstructed in each process.

6.3.1 Data transfer

The element objects array is allocated from the CPU, therefore elements can be copied from the GPU to the CPU directly. The relevant data members are the polynomial order of the elements, their Hilbert curve status, their rotation and their split level. All other members are either not stored directly in the element structure and therefore cannot be copied directly, or can be computed again when received. An MPI datatype is created to be able to send and receive arrays of elements directly without copying that data into separate arrays. Subsection 6.5.1 details this process.

The solution data of an element is not so easy to transfer between processes. The solution arrays are stored in GPU dynamic memory and the elements only hold a pointer to them. Being allocated from the GPU, these arrays cannot be transferred to the CPU directly. They also have a different size depending on the element's polynomial order. To solve this, a buffer is allocated from the CPU on the GPU, and a kernel is launched to copy each element's solution into this buffer on the GPU. To do this in parallel, the buffer is of the maximum size of the solution, times the number of elements. This is so elements can easily compute where to copy their data without race conditions. This uses more memory than if the solution was packed, but then it would be harder for elements to find their data on the receiving end, and to find where to write it on the sending end.

This kernel allows us to fetch other data that is not readily available on the CPU. Processes only store the mesh block assigned to them, with no information on elements, nodes, faces or boundaries from other processes. Elements store the indices of their nodes, whereas the nodes

themselves are stored in an array of their own. These indices are meaningless to the receiving process, as it has no idea what nodes those indices correspond to. The kernel fetching the solution data from the GPU also copies the node values to an array, so that the four coordinates of the corners of each element can be sent.

This is all the data that is needed to send the elements themselves from one GPU to the other. Once it is received, it is stored using the inverse process. The nodes are checked for a match among the existing nodes within the receiving GPU, otherwise new nodes are appended to the node array. Then, elements need to be linked to their neighbours.

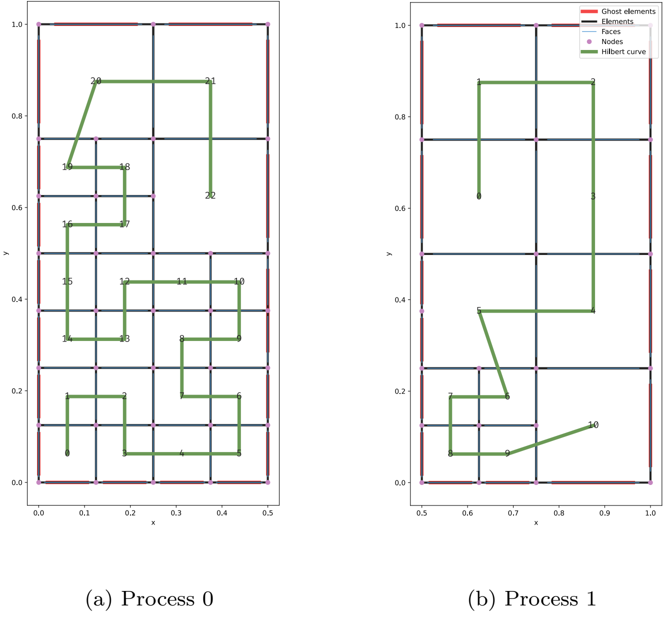
The nodes themselves are not enough to derive the connectivity of an element. The number of faces on a side of an element has no upper bound, so the data will need to be packed and offsets will need to be computed. A first kernel retrieves the number of faces, and therefore neighbours, on each side of elements to be sent. This information is used on both the sending and receiving sides to allocate arrays that can fit the total number of neighbours, as well as compute offsets to where each element has its neighbours in these arrays. Information to be fetched from neighbours and sent includes: the neighbours' local indices, process, side, polynomial order, and the two nodes on the connecting side. This information is used both to find the relevant neighbour if it is also in the receiving process, and to create MPI boundaries if the neighbour is in another process.

On the subject of MPI boundaries, another data exchange must take place during load balancing. Every process that has an MPI boundary with another process must send the new local index and new process of each element making up that boundary. This is so the other process knows to receive from another process if the element has been sent to a new process. This is also necessary to send elements that are adjacent to an MPI boundary, as these elements will have neighbours in another process whose local indices need to be updated.

6.3.2 Connectivity

Once all the data has been received, the mesh needs to be reassembled. Faces and boundary elements that link to elements that have been sent are removed. Faces around received elements are created, or reused if the element happens to have been the destination of an MPI interface. New boundary elements, also called ghost elements because they are not part of the domain, are created around received elements, and as a replacement for sent elements. Elements and faces that are kept are moved to new indices, using a similar prefix sum to the one used in Subsection 5.6.2. All references to these faces and elements are updated.

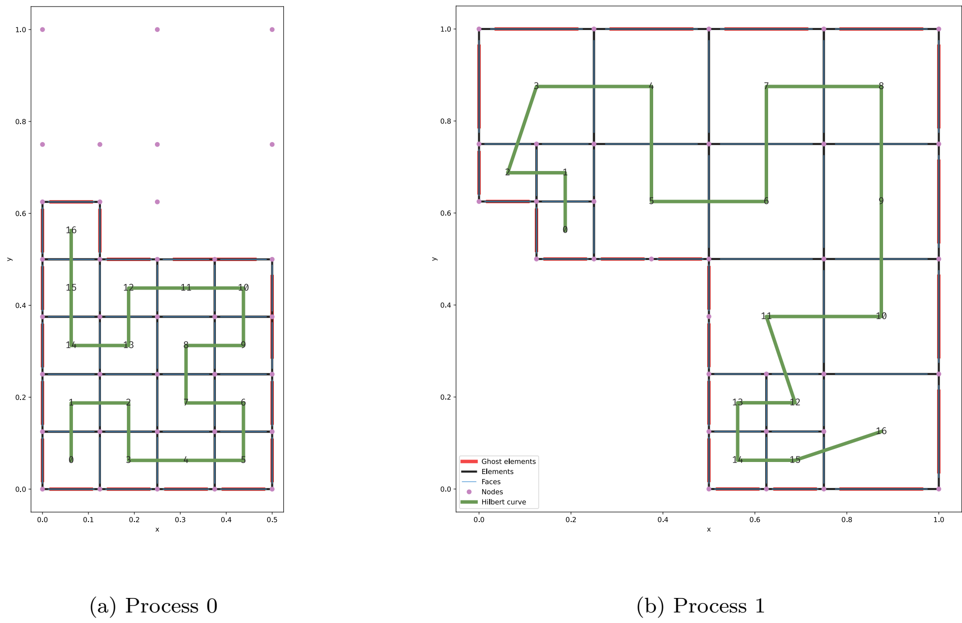
Figures 6.14 and 6.15 illustrate an example of the whole load balancing process, where two processes initially have an unbalanced number of elements because of a prior refinement. The domain spans $x, y \in [0, 1]$ and the elements making up that domain are split between two GPUs.



(a) Process 0

(b) Process 1

Figure 6.14: Mesh before load balancing: The two blocks have different numbers of elements.



(a) Process 0

(b) Process 1

Figure 6.15: Mesh after load balancing: The two blocks have equal numbers of elements.

6.4 Load Balancing Criteria

Dynamic load balancing is a costly process, especially when using GPUs. Transfers between GPUs are expensive, as is reallocating memory for arrays that change size. Load balancing at every occasion would needlessly waste resources than could be used to solve the program. A balance has to be established between having a good quality mesh for a longer part of the computation against load balancing less often and saving the load balancing processing time.

Two approaches are studied in this work to alleviate this problem. The first approach is to load balance the mesh at a regular interval. If the mesh is not refined after each AMR step, but every few AMR steps, computation time can be saved on the load balancing routine. On the other hand, the mesh may be unbalanced between these steps. If a particularly large number of elements are refined at a time, a significant load imbalance can persist until the next load balancing step.

The second approach is to set an acceptable load imbalance, under which the mesh will not be load balanced. After refining, if the load imbalance L is below that threshold, the mesh will not be load balanced. This means that the mesh will only be load balanced when it is really needed. The disadvantage is that the load imbalance could stay slightly below that threshold indefinitely, leaving the program with a slightly imbalanced mesh for large parts of its runtime.

The two approaches are compared in Section 7.5.

6.5 Implementation

Dynamic load balancing presents significant challenges when GPUs are used. Mesh data that resides on the GPU is not readily accessible by the CPU to be transferred from one GPUto another, and as much of the algorithm as possible should run on the GPU in parallel. Also, reconstructing the mesh and the connectivity between blocks is difficult when no worker has a global view of the mesh.

6.5.1 Element exchange

Computing which elements to exchange from one GPU to another is fairly straightforward. The workload levelling from Section 6.2 informs each worker of which elements it should send to which other worker, and which elements it should receive from which other worker.

As mentioned previously, the element solution is stored in dynamic GPU memory, and cannot be copied directly to the CPU. A kernel is launched to copy that data to an array which can then be copied to the CPU. Algorithm 6.1 shows how that data is packed in the solution array to be sent between workers. The values of the four nodes of each element are also stored, as well as the number of neighbours on each side. The data is aligned to the maximum

polynomial order used by the program, so that elements can store their solution in parallel without knowledge of other elements' polynomial order. This wastes space as most elements do not use the entire space they are assigned to in the transfer arrays. Since this avoids computing an offset array to indicate where each element's solution should be placed if the data was packed, this tradeoff is accepted.

Algorithm 6.1 get_transfer_solution: The solution data of elements is stored in parallel in an array.

```
__global__
auto get_transfer_solution(size_t n_elements, const Element2D_t* elements,
        int maximum_N, const Vec2<deviceFloat>* nodes, deviceFloat* solution,
        size_t* n_neighbours, deviceFloat* element_nodes) -> void {

    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t i = index; i < n_elements; i += stride) {
        const size_t p_offset =  3 * i      * std::pow(maximum_N + 1, 2);
        const size_t u_offset = (3 * i + 1) * std::pow(maximum_N + 1, 2);
        const size_t v_offset = (3 * i + 2) * std::pow(maximum_N + 1, 2);
        const size_t neighbours_offset = 4 * i;
        const size_t nodes_offset = 8 * i;

        const Element2D_t& element = elements[i];

        n_neighbours[neighbours_offset]     = element.faces_[0].size();
        n_neighbours[neighbours_offset + 1] = element.faces_[1].size();
        n_neighbours[neighbours_offset + 2] = element.faces_[2].size();
        n_neighbours[neighbours_offset + 3] = element.faces_[3].size();
        element_nodes[nodes_offset]     = nodes[element.nodes_[0]].x();
        element_nodes[nodes_offset + 1] = nodes[element.nodes_[0]].y();
        element_nodes[nodes_offset + 2] = nodes[element.nodes_[1]].x();
        element_nodes[nodes_offset + 3] = nodes[element.nodes_[1]].y();
        element_nodes[nodes_offset + 4] = nodes[element.nodes_[2]].x();
        element_nodes[nodes_offset + 5] = nodes[element.nodes_[2]].y();
        element_nodes[nodes_offset + 6] = nodes[element.nodes_[3]].x();
        element_nodes[nodes_offset + 7] = nodes[element.nodes_[3]].y();

        for (int j = 0; j < std::pow(element.N_ + 1, 2); ++j) {
            solution[p_offset + j] = element.p_[j];
            solution[u_offset + j] = element.u_[j];
            solution[v_offset + j] = element.v_[j];
        }
    }
}
```

Elements can have an arbitrary number of neighbours, therefore neighbour data cannot be staggered in the same way as solution data. This is why we need to transfer the number of neighbours on each side of elements, so as to generate the neighbour information arrays with the correct size, and to generate offset arrays so each element stores its neighbour data in the correct memory location. The neighbour data fetch kernel is much more complicated than Algorithm 6.1, but operates in a similar manner. The main difference is that neighbour data is not readily available if a neighbour is a boundary element. In that case the different boundary condition and MPI interface arrays are searched to find which one the boundary element belongs to, and to find the required information that way. This is because boundary elements that are part of an MPI interface must present the local index and process of the element on the other side of the interface, and boundary condition boundary elements must encode which kind of

boundary condition they are.

Elements themselves are copied directly from the GPU to the CPU since they are stored in CPU-allocated CUDA GPU memory. Using a created MPI datatype, arrays of elements objects can be directly sent through MPI. Algorithm 6.2 shows how the datatype is created, and which data members from the element object are sent.

Algorithm 6.2 MPI_datatype: A MPI datatype is created to send and receive arrays of elements. Only some data members are sent.

```
Element2D_t::Datatype::Datatype() {
    constexpr int n = 4;
    constexpr std::array<int, n> lengths {
        1,
        1,
        1,
        1
    };
    constexpr std::array<MPI_Aint, n> displacements {
        offsetof(Element2D_t, N_),
        offsetof(Element2D_t, status_),
        offsetof(Element2D_t, rotation_),
        offsetof(Element2D_t, split_level_)
    };
    const std::array<MPI_Datatype, n> types {
        MPI_INT,
        MPI_INT,
        MPI_INT,
        MPI_INT
    };

    MPI_Datatype tmp_type;
    MPI_Type_create_struct(n, lengths.data(),
        displacements.data(), types.data(), &tmp_type);
    // We resize the datatype to be able to send arrays of elements
    MPI_Type_create_resized(tmp_type, 0, sizeof(Element2D_t), &datatype_);
    MPI_Type_commit(&datatype_);
}
```

6.5.2 Mesh reconstruction

Once all elements to be transferred and their neighbour information have been sent and received, the mesh must be reconstructed. This is the more difficult part of the process, as many edge cases exist. An example operation is the **find_faces_to_delete** kernel, showcased in Algorithm 6.3, which marks faces without any remaining domain elements as neighbours for deletion.

Algorithm 6.3 find_faces_to_delete: Faces with no remaining domain neighbour elements are marked for deletion.

```cpp
__global__
auto find_faces_to_delete(size_t n_faces, size_t n_domain_elements,
        size_t n_elements_send_left, size_t n_elements_send_right,
        const Face2D_t* faces, bool* faces_to_delete) -> void {

    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t i = index; i < n_faces; i += stride) {
        const Face2D_t& face = faces[i];
        const std::array<bool, 2> elements_leaving {
            face.elements_[0] < n_elements_send_left
                || (face.elements_[0] >= n_domain_elements - n_elements_send_right
                    && face.elements_[0] < n_domain_elements),
            face.elements_[1] < n_elements_send_left
                || (face.elements_[1] >= n_domain_elements - n_elements_send_right
                    && face.elements_[1] < n_domain_elements)
        };
        const std::array<bool, 2> boundary_elements {
            face.elements_[0] >= n_domain_elements,
            face.elements_[1] >= n_domain_elements
        };

        faces_to_delete[i] = elements_leaving[0] && elements_leaving[1]
                          || elements_leaving[0] && boundary_elements[1]
                          || elements_leaving[1] && boundary_elements[0];
    }
}
```

This algorithm provides part of the foundation of the part of load balancing concerning faces, but is only the start. Faces need to be moved to a new index to remove the gaps from deleted faces. We then need to update the face indices in all elements, and resize the elements' neighbour faces arrays if some of the faces have been deleted. New faces have to be added to the face array to connect the new received elements to the mesh.

Ghost elements can be marked for deletion if all the faces connecting them to the domain are to be deleted, except in two special cases. Ghost elements that are part of an MPI interface need to be deleted if the element on the other side of the interface is sent to the process containing the boundary element. Algorithm 6.4 details a kernel that is launched specially to deal with this edge case. Also, boundary elements can be marked for deletion because no domain element connects them anymore, but some received elements could connect to them. Another kernel, shown in Algorithm 6.5, is launched to deal with this case.

Algorithm 6.4 find_mpi_interface_elements_to_delete: MPI interface elements should be deleted if the element on the other side of the interface is moved to the same process.

```
__global__
auto find_mpi_interface_elements_to_delete(size_t n_mpi_interface_elements,
        size_t n_domain_elements, int rank,
        const size_t* mpi_interfaces_destination,
        const int* mpi_interfaces_new_process_incoming,
        bool* boundary_elements_to_delete) -> void {

    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t i = index; i < n_mpi_interface_elements; i += stride) {
        if (mpi_interfaces_new_process_incoming[i] == rank) {
            boundary_elements_to_delete[mpi_interfaces_destination[i]
                                          - n_domain_elements] = true;
        }
    }
}
```

Algorithm 6.5 find_mpi_interface_elements_to_keep: MPI interface elements should be kept if a received element now connects to it.

```
__global__
auto find_mpi_interface_elements_to_keep(size_t n_mpi_destinations,
        size_t n_neighbours, int rank, size_t n_domain_elements,
        const int* neighbour_procs, const size_t* neighbour_indices,
        const size_t* neighbour_sides, const int* mpi_destination_procs,
        const size_t* mpi_destination_local_indices,
        const size_t* mpi_destination_sides,
        const size_t* mpi_interfaces_destination,
        bool* boundary_elements_to_delete) -> void {

    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int stride = blockDim.x * gridDim.x;

    for (size_t i = index; i < n_mpi_destinations; i += stride) {
        const int mpi_proc = mpi_destination_procs[i];

        // The element is marked for deletion because either all linking elements are sent
        // Maybe we received an element that links to it, then we must keep it
        if  (mpi_proc != rank
          && boundary_elements_to_delete[mpi_interfaces_destination[i]-n_domain_elements]) {
            const size_t mpi_index = mpi_destination_local_indices[i];
            const size_t mpi_side = mpi_destination_sides[i];
            for (size_t j = 0; j < n_neighbours; ++j) {
                const int neighbour_proc = neighbour_procs[j];
                const size_t neighbour_index = neighbour_indices[j];
                const size_t neighbour_side = neighbour_sides[j];

                if  (mpi_proc  == neighbour_proc
                  && mpi_index == neighbour_index
                  && mpi_side  == neighbour_side) {
                    boundary_elements_to_delete[mpi_interfaces_destination[i]
                                                  - n_domain_elements] = false;
                    break;
                }
            }
        }
    }
}
```

This example is presented to show that the apparent simplicity of Algorithm 6.3 is complicated when all the edge cases of a dynamically changing mesh are factored in. Adding to the complexity is the fact that the GPU workers only have knowledge of their own mesh block,

as well as the neighbour information of the received elements. The different computations are also executed in parallel on the GPU, since each GPU is paired with a single CPU which would heavily bottleneck the process if it was used for all calculations.

Many more operations remain, such as moving elements that are not deleted to their new position in the array, adding received elements, adding boundary elements around received elements, and updating the element indices in all faces. The MPI interfaces must also be rebuilt and ordered since elements may have moved from one process to another. The methods used are similar to those used for faces, and those seen in Section 5.6.

Chapter 7

Results

This chapter presents results of the implementation of the adaptive DG-SEM wave equation solver on GPU architectures and studies its effectiveness. Three main components are studied. First, the performance of the core spectral element method code on GPUs is presented in Section 7.3 with a comparison with the same code running on CPUs for a simple wave problem. Then, the performance of the adaptive mesh refinement is studied in Section 7.4. Dynamic load balancing is studied in Section 7.5. Finally, more complex applications are shown in Sections 7.6 and 7.7.

7.1 Platforms

The implementation we have developed aims to perform large scale computations, and as such targets HPC platforms. The program has been designed to scale to multiple levels of parallelism, the highest being splitting the workload between several HPC nodes, each having multiple CPUs and GPUs. To showcase how the program works on such platforms, most of the following tests were run on clusters graciously offered by Compute Canada[1]. Two such clusters were used for our experiments, Béluga and Narval.

7.1.1 Béluga

Béluga is a heterogeneous cluster suitable for a variety of purposes, from traditional CPU computing to massively parallel GPU computing. It is located at the École de technologie supérieure in Montréal. Béluga is made up of 977 compute nodes of different types, totaling 39,120 CPU cores and 696 GPUs. For our testing, we use Béluga's general usage GPU nodes. Each one of the 172 GPU nodes contains 40 CPU cores and 186 GB of memory across two Intel Gold 6148 CPUs, and four Nvidia V100 SXM2 GPUs with 16 GB of memory each. One

[1] https://www.computecanada.ca/

such node totals about 2 TFlops of double precision CPU computing power, and 13 TFlops of double precision GPU computing power. The nodes are connected by Infiniband HDR (100 Gb/s) interconnections.

7.1.2 Narval

Narval is a heterogeneous cluster suitable for a variety of purposes, from traditional CPU computing to massively parallel GPU computing and artificial intelligence workloads. It is also located at École de technologie supérieure in Montréal. Narval is made up of 1301 compute nodes of different types, totaling 80,720 CPU cores and 636 GPUs. For our testing, we use Narval's GPU nodes. Each one of the 159 GPU nodes contains 48 CPU cores and 498 GB of memory across two AMD Milan 7413 CPUs, and four Nvidia A100 GPUs with 40 GB of memory each. One such node totals about 2.2 TFlops of double precision CPU computing power, and 38.8 TFlops of double precision GPU computing power. The nodes are connected by Infiniband EDR (100 Gb/s) interconnections.

7.1.3 Consumer hardware

Some smaller scale tests or tests that did not involve performance were run on consumer hardware. This is important, as it may not always be possible to access HPC systems when flow simulations have to be performed. The program should also have reasonable performance on regular systems. The system used in these cases is a single computer containing 16 CPU cores and 128 GB of memory across two Intel E5–2650 V2 CPUs, and one Nvidia GTX 1070 GPU with 8 GB of memory. This totals about 0.5 TFlops of double precision CPU computing power, and 0.25 TFlops of double precision GPU computing power. This is an interesting comparison, as consumer GPUs are not geared towards high precision compute loads. GPUs found in HPC systems typically have a 1:2 ratio between their double precision and single precision performance, whereas consumer GPUs usually have 1:32 ratio. This means that computations using double precision floating point numbers will execute at one sixteenth of the speed of compute-oriented GPUs, all other factors being equal.

7.2 Test Case

The test case used in this chapter is shown in Figure 7.1. We solve the 2D wave equation from Section 4.2 on a square domain of size 1 in each dimension. A linear Gaussian wave starts just outside of the domain, and traverses the domain diagonally. The wave solution is given by Equation 7.1.

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{k_x}{c} \\ \frac{k_y}{c} \end{bmatrix} e^{-\frac{[k_x(x-x_0)+k_y(y-y_0)-ct]^2}{d^2}} \tag{7.1}$$

$c = 1$ is the speed of sound. k_x and k_y are the components of the wavevector $\mathbf{k}$. We set both k_x and k_y and to $\frac{\sqrt{2}}{2}$ to normalise $\mathbf{k}$. We set $d = \frac{0.2}{2\sqrt{\ln 2}}$. x_0 and y_0 govern the initial location of the wave. We use $(x_0, y_0) = (-0.2, -0.2)$ so that the wave starts just outside of the domain.

The solution is computed until a time of $t = 2s$, with a Δt satisfying CFL $= 0.5$. Figure 7.1 shows the pressure distribution at $0.5s$. The initial conditions are set using Equation 7.1. The boundary conditions at the edges of the domain are constrained to the analytical solution. The number of elements K varies depending on the problem, as does the polynomial order N. This problem should help demonstrate key aspects of the program, as the solution is steeper near the wave and needs refinement for accurate results.

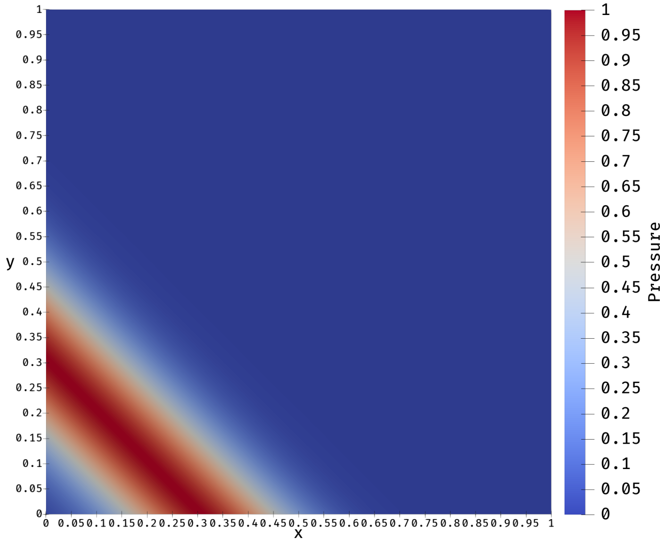


Figure 7.1: Test case: A wave travels through a square domain at a 45° angle.

7.3 Scaling Tests

Parallel scaling is an important aspect of the performance of a program. It describes how the execution time varies when more resources are assigned to the problem. This is an optimisation problem, as each problem will need a different amount of resources dedicated to it to attain the best scaling possible.

A completely parallelisable program, often called embarrassingly parallel, should scale linearly with the amount of resources. The speedup S of a workload split between P workers should be equal to the number of workers in Equation 7.2, with t_1 being the time to solve the

problem serially using a single worker, and t_p being the time to solve the problem in parallel using P workers. In this work, workers are a single GPU or a single CPU core for comparative tests.

$$S = \frac{t_1}{t_P} \tag{7.2}$$

Unfortunately, it is often not possible to make a program entirely parallel. In such cases, dividing a problem into more and more smaller tasks obeys the well-known Amdahl's law [2], as seen in Equation 7.3. It splits the program into a sequential part s and a parallel part l, with only the parallel part scaling with the number of workers.

$$S = \frac{1}{s + \frac{l}{P}} \tag{7.3}$$

This is called *strong scaling*, where a fixed problem is solved using varying amounts of resources. Results of strong scaling tests of the solver are discussed in Subsection 7.3.1.

On the other hand, increased available resources can enable working on bigger problems. Gustafson argues [25] that modern highly parallel computers will be used to solve more complex problems instead of solving smaller problems faster. Such a scaling is described with Equation 7.4.

$$S = s + l \times P \tag{7.4}$$

This is called *weak scaling*. It describes solving a problem whose size increases with the amount of resources, while the task size per worker stays constant. Weak scaling is tested in Subsection 7.3.2. With weak scaling a program will scale indefinitely, with the slope of the scaling being influenced by the parallel proportion of the task.

This section will also serve as a general comparison between performance of the code on CPUs and GPUs. All scaling tests have been performed on the Béluga supercomputer. Scaling is evaluated by varying the number of *nodes*, where a node in this context is a single computer from Béluga, sporting 40 CPU cores and 4 GPUs as stated in Subsection 7.1.1. The four-node execution time of those tests therefore represents the program running on 16 GPUs or 160 CPU cores, for the GPU and CPU computations respectively.

Both scaling tests evaluate the core solver part of the program, as the meshes start with enough elements to have a well resolved solution throughout the simulation. The adaptive mesh refinement and dynamic load balancing subroutines are still enabled, but the error is never estimated to be high enough to perform AMR. In turn, no load imbalance occurs and the load balancing routine never finds enough load imbalance to trigger.

7.3.1 Strong scaling

The test case for this section is described in Section 7.2. The domain is divided into 256 elements in each of the x and y directions, for a total of $K = 65536$ elements. The problem is then solved in parallel using a range of nodes, from one quarter node to eight nodes. One quarter node contains 1 GPU and 10 CPU cores, while eight nodes contain 32 GPUs and 320 CPU cores. The mesh is split into blocks accordingly, one block per GPU or one block per CPU core. The first test has a polynomial order $N = 4$ for all elements, and a block size of $B = 32$ threads per block.

The block size, as explained in Subsection 3.1.2, is the number of threads in a block of threads. Having a higher block size can speed up execution by reducing the number of instructions to dispatch by the control flow unit of the GPU, because the instructions are dispatched by block of threads. However, if the threads within a block diverge by taking different branching code paths, having a higher block size can actually lower performance. A single thread diverging will have all the threads in its block waiting until it is finished, therefore a higher block size will have more threads stalled when divergence occurs.

In Figure 7.2, we add two “ideal” lines both for the GPU and CPU calculations. These lines represent perfect scaling from a data point. The dashed and dotted line represents perfect scaling from the first data point, one GPU or 10 CPU cores. This is equivalent to halving the computation time every time the computing power doubles. The GPU curve is always below this ideal line. As we increase the number of GPUs, the performance scales better than linearly up to a point, and then decreases. This indicates that the most efficient point for GPUs is not the first. We therefore add a second dashed line, which represents the scaling from the best point. The best point is where the computation time multiplied by the number of GPUs, which gives the total computation time, is the lowest. This line is more representative, and will be used for the next figures.

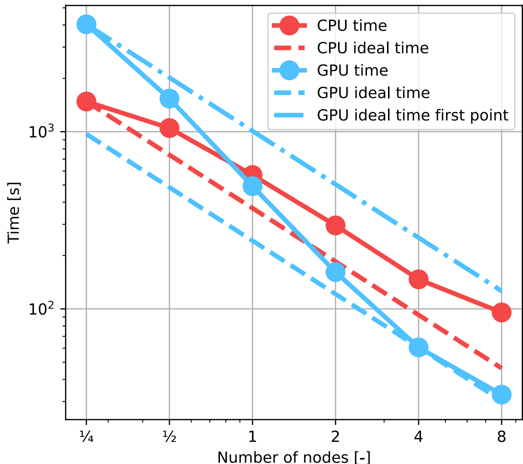


Figure 7.2: Strong scaling: The number of workers increases while the problem size stays fixed. $N = 4$, $K = 65536$, $B = 32$

Figure 7.2 shows that the GPU implementation, initially slower than the CPU implementation, overtakes it when one full node or more is used. The higher number of nodes show the GPU implementation to be 2.9× faster that the CPU implementation.

At low numbers of GPUs working on the same problem, the GPUs are more heavily loaded. It is possible that the higher memory usage, or the increased cache eviction rate lowers the performance of GPUs in that regime. With a lower number of elements being worked on by a GPU, cache has to be emptied less often to make room for more elements to use for computations.

At the highest number of GPUs working on the same problem, we see that the scaling is slightly reduced. This could be a hint that the GPUs are not sufficiently loaded for those cases. These GPUs have many CUDA cores, 5120 each in the case of Béluga. If the workload does not saturate the GPU, some of those cores will be idle, leading to worse performance. The best number of elements per GPU seems to be around 4096 for this case.

The next test, shown in Figure 7.3, shows the same test case but with an increased polynomial order of $N = 6$ for all elements. The rationale is that the GPU should spend more time on the computations, since the computations will be heavier, while spending the same time on the overhead of kernel calls and other housekeeping.

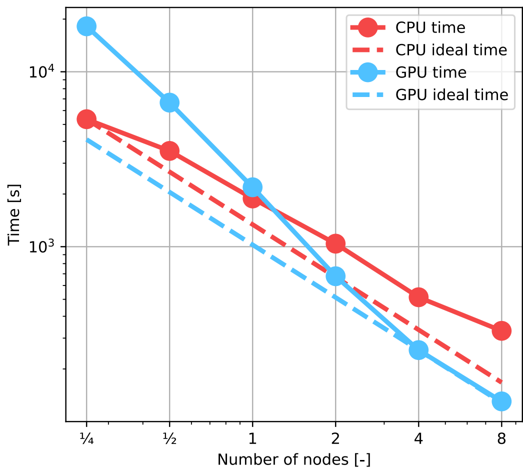
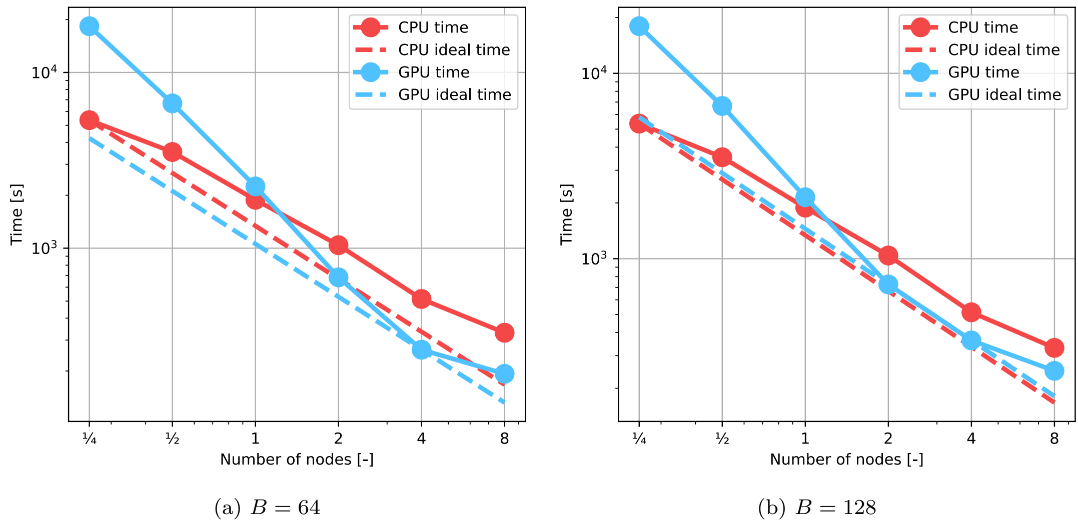


Figure 7.3: Strong scaling: The number of workers increases while the problem size stays fixed. $N = 6$, $K = 65536$, $B = 32$

Figure 7.3 shows that the increased polynomial order has not increased the performance of the GPU implementation compared to the CPU implementation, rather shifting to the right the crossover point where the GPU implementation is faster. It is possible that the reduced relative performance is linked to the increased memory used by the solution, as the needed storage scales with $(2N + 1)^2$, where N is the polynomial order of elements.

Next, this last case is computed again with increased block size B, first 64 on Figure 7.4a, then 128 on Figure 7.4b. The CPU implementation is not modified. This should assess the incidence of block size on the solution time.



(a) $B = 64$

(b) $B = 128$

Figure 7.4: Strong scaling: The number of workers increases while the problem size stays fixed. $N = 6$, $K = 65536$ (a) $B = 64$ (b) $B = 128$

As the last figure suggests, increasing the block size B does incur a performance penalty. This is possibly caused by a high amount of thread divergence within blocks. This is consistent with how the program is written, for example Section 5.6 shows how the projection back and forth between the elements and faces has several branching code paths. This creates divergence between threads if the threads in a block happen to take different code paths, which reduces performance as these paths are then executed serially. A lot of changes in the architecture of the program would have to be made to reduce this branching, and switch to a more data-driven approach.

One thing to note is that the curve of the CPU implementation on Figures 7.2 and 7.3 is similar, whereas the GPU curves from all the figures differ more. This is a recurring theme of all the tests in this chapter: code running on GPUs seems to be harder to optimise and some factors have unexpected influences on the performance.

7.3.2 Weak scaling

The weak scaling test uses the test case from Section 7.2. Weak scaling increases both the size of the problem and the amount of processing power at the same rate. To achieve this, we increase the number of elements in the domain to keep the number of elements per supercomputer node constant. The domain is split up into elements such that each node's mesh block contains 16384 elements. Since each node is made up of four GPUs and 40 CPUs, this amounts to 4096 elements per GPU, or 410 elements per CPU core. To ensure that all these cases perform the same number of iterations, the CFL is adjusted to keep the time step size equal. An ideal result is a flat curve, indicating that by increasing the problem size and resources by the same amount, the simulation time stays the same. The problem is studied from ¼ node to 16 nodes.

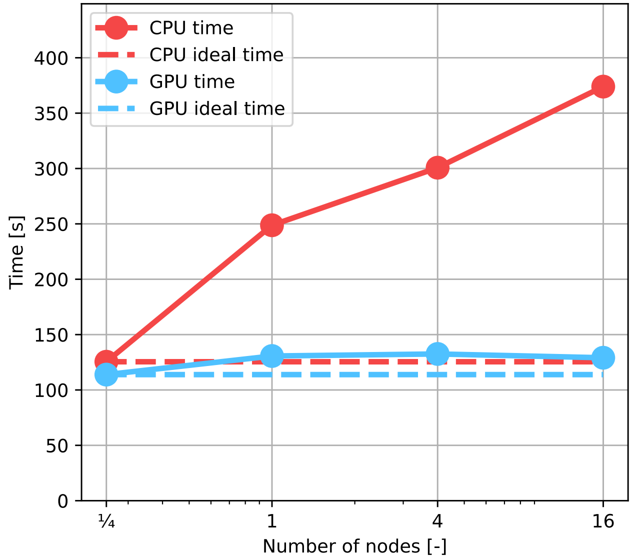


Figure 7.5: Weak scaling: The problem size increases with the number of workers. $N = 4$, $K = 16384/node$

Figure 7.5 shows good weak scaling from the GPU implementation. Keeping in mind that the quarter node and single node cases have a reduced workload by not having to do any MPI communication over the network, and the quarter node GPU case has no MPI communication to do at all, the GPU curve is reasonably flat. The CPU curve is not as good, indicating that there may be a bottleneck in the communication part of the program. The CPU implementation has a much higher number of workers than the GPU one, 40 per node instead of four, increasing the inter-process communication needed. The slope of the CPU curve is about 0.4 with respect to the number of CPU cores.

7.4 Adaptive Mesh Refinement Performance

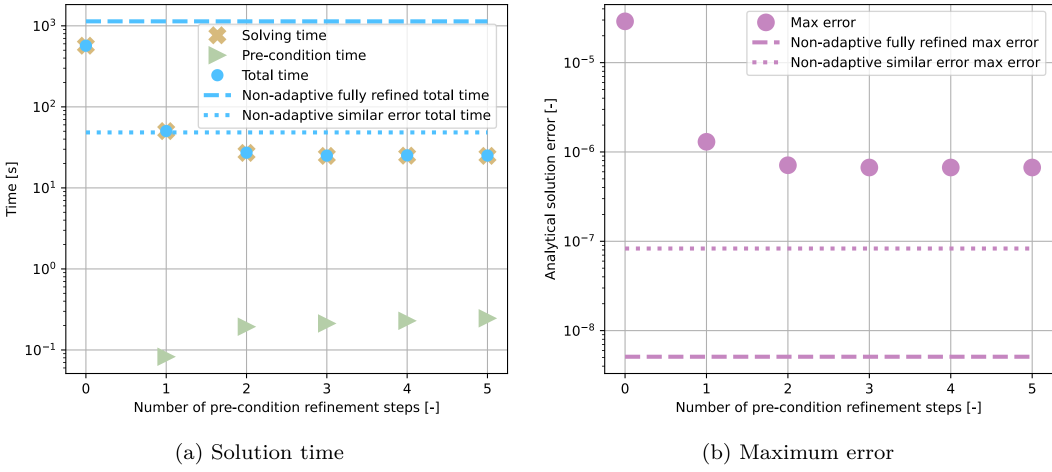
This section establishes the performance of the adaptive mesh refinement. The problem from Section 7.2 is used, initially split into four elements in each of the x and y directions, for a total of $K = 16$ elements. The initial polynomial order is $N = 4$. The system is allowed to refine up to a split level $S = 5$, meaning a cell can split five times and become $\frac{1}{32}$ of the size of the original cell, and up to a polynomial order $N = 16$. The target error threshold is set to 1×10^{-6}, meaning it will refine elements whose estimated error is greater than that number.

Adaptive mesh refinement is a costly process, as the CPU must reallocate arrays on the GPU for the different objects making up the mesh, and then schedule kernels to move objects to the new arrays. We test here two strategies to reduce the performance overhead of adaptive mesh refinement, while reducing error.

First, adaptive mesh refinement will be performed at a regular interval. Figures 7.6, 7.7, 7.8 and 7.9 show the results of refining the mesh every 5, 20, 100 and 500 timesteps, respectively.

Secondly, the simulations are performed with an increasing number of refinement pre-condition steps as described in Section 5.5. The analysis ranges from no pre-condition step to five steps. The total simulation time is shown in blue. It is the sum of the time taken up by the pre-condition of the mesh, shown in green, and the computation time to solve the problem, shown in yellow.

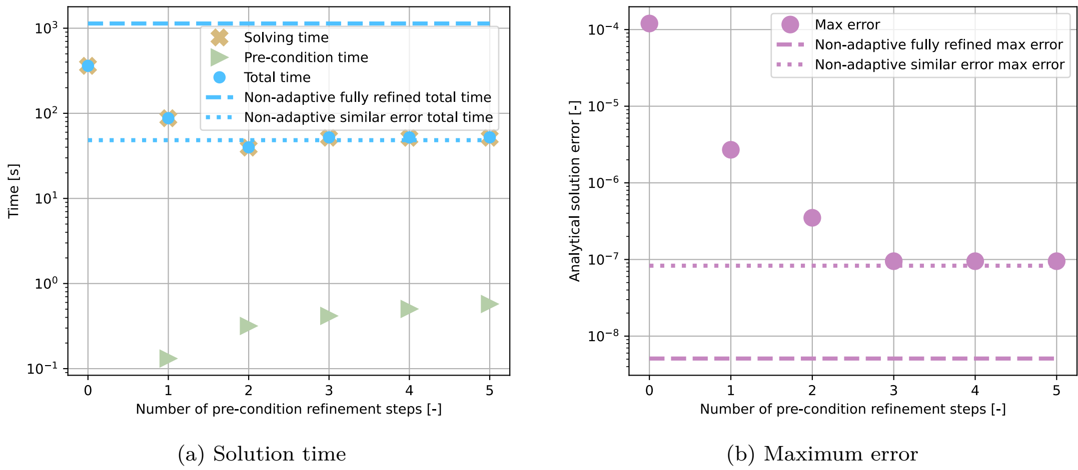
We compare those simulations to two non-adaptive cases. In the first case, the initial mesh is fully refined uniformly up to the maximum level attained by the adaptive cases. The mesh is split into 128 elements in each of the x and y dimensions, for a total of $K = 16384$ elements, with a polynomial order $N = 10$. This result is shown as a dashed line on the figures. The second non-adaptive case aims to obtain a maximum error similar to the adaptive case. The mesh is refined uniformly to 32 elements in each of the x and y dimensions, for a total of $K = 1024$ elements, with a polynomial order $N = 10$. This case is shown as a dotted line on the figures. Both the simulation time and the error relative to the analytical solution are plotted. The cases were computed using the Narval supercomputer on a single GPU.



(a) Solution time

(b) Maximum error

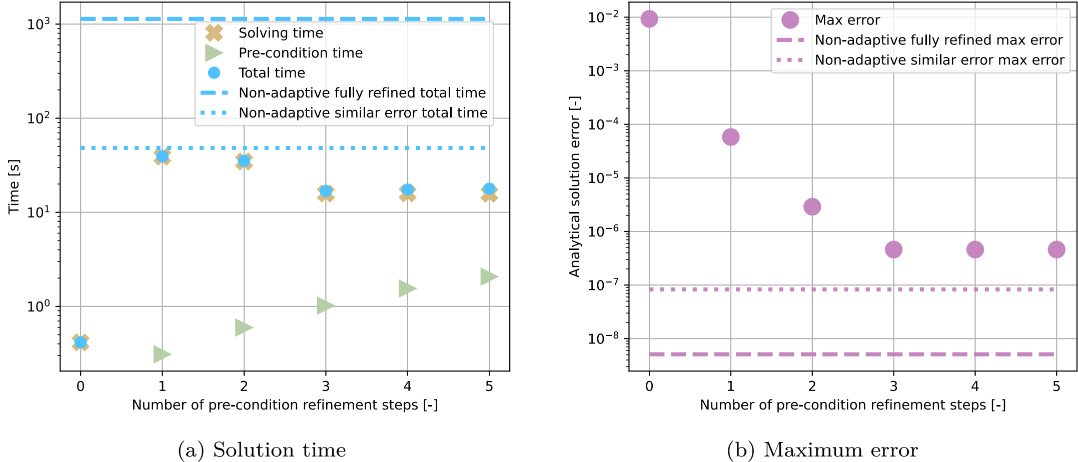
Figure 7.6: Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval = 5



(a) Solution time

(b) Maximum error

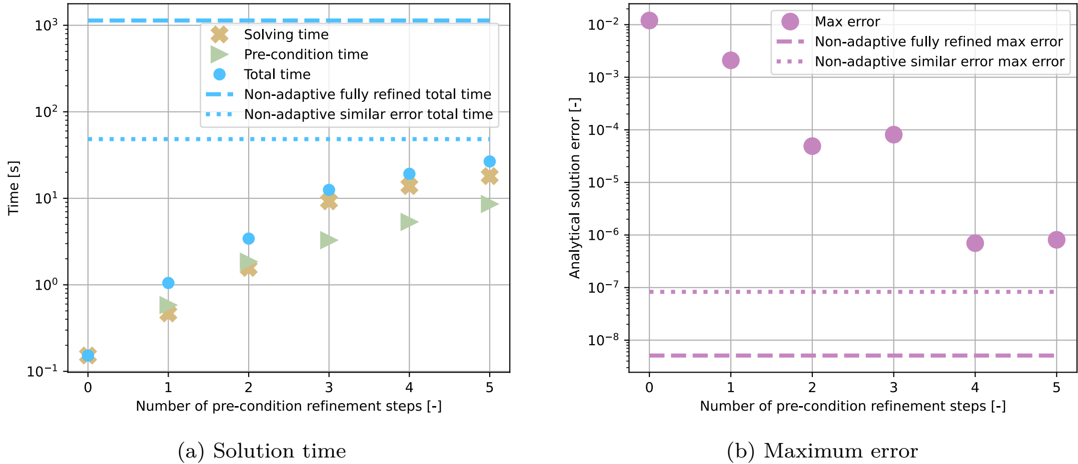
Figure 7.7: Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval $= 20$



(a) Solution time

(b) Maximum error

Figure 7.8: Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval $= 100$



(a) Solution time

(b) Maximum error

Figure 7.9: Adaptive mesh refinement performance: Simulation time and error for increasing number of pre-condition steps. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, refinement interval $= 500$

From Figures 7.6, 7.7, 7.8 and 7.9, we see that this is an optimisation problem, with tradeoffs between the simulation time and the error generated.

An important part of the error comes from the initial conditions being applied on too coarse a mesh. Even if the mesh is refined, this error propagates through the domain. Mesh pre-condition, as described in Section 5.5, aims to alleviate this problem. Indeed, with enough pre-condition steps all cases converge to an acceptable error. Adding more pre-condition steps does not improve the results, as the target error threshold of 1×10^{-6} is likely reached.

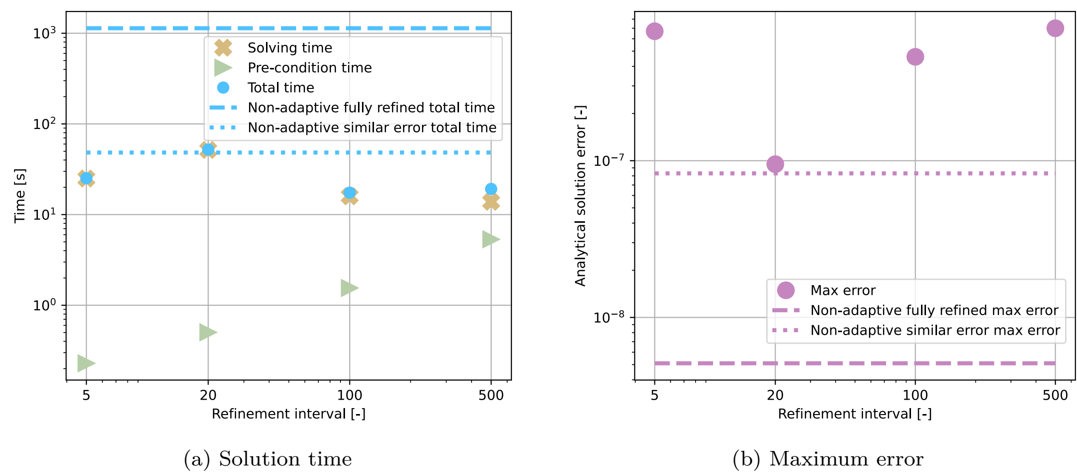
The fully refined non-adaptive case, shown in dashed lines on Figures 7.6, 7.7, 7.8 and 7.9, is prohibitively time consuming to compute. The solution is well resolved as shown on Figure 7.7b, with the analytical solution error being 19× smaller than in the adaptive case. On the other hand, it takes 22× as much time to compute. The difference is even more pronounced when the refinement interval is increased to 100, as in Figure 7.8a, where the adaptive solution with three to five pre-condition steps is around 67× faster, at the expense of a worse error compared to the 20 refinement interval case.

As for the similar maximum error non-adaptive case, shown with dotted lines, the results are closer. The case with a refinement interval of 20 on Figure 7.7 was used to set the target error for that non-adaptive case. The adaptive case starting from a coarse mesh is able to reach a similar maximum error and similar computation time when compared to the non-adaptive case. The other cases also attain the target error threshold of 1×10^{-6} while being faster.

The non-adaptive cases are closer to being ideal for the GPU architecture. The non-adaptive cases have less thread divergence than the adaptive case. There is a lot more branching in the adaptive case, with elements having a different polynomial order, non-conforming interfaces, and different numbers of neighbours. This increased divergence lowers the performance

since some threads are predicated off while others execute an instruction. The memory access patterns of the non-adaptive cases are also better, for example the different faces connecting to an element are closer in memory. Threads from a same warp accessing memory that is not contiguous must perform several memory fetches. The non-adaptive cases play to the strengths of the GPUs, yet the adaptive case shows performance improvements. This means that AMR is worth it even if the GPU architecture is not perfectly suited for it, and with improvements to fit it better it could perform even better. Section 8.3 discusses some improvements to make the program even better suited for the GPU architecture.

The best number of pre-condition steps seems to be four for this case, as the target error is reached with all refinement intervals. After this number of steps the solution does not change much as the target error threshold is likely reached. We use that number of steps to compare the effects of different refinement intervals in Figure 7.10.



(a) Solution time

(b) Maximum error

Figure 7.10: Adaptive mesh refinement performance: Simulation time and error for four pre-condition steps and increasing refinement interval. $N_{initial} = 4$, $K_{initial} = 16$, $S = 5$, pre-condition steps = 4

Figure 7.10 gives a good overview of the AMR performance. All refinement intervals attain the target error threshold of 1×10^{-6}, and obtain a real maximum error under 1×10^{-6} when compared to the analytical solution.

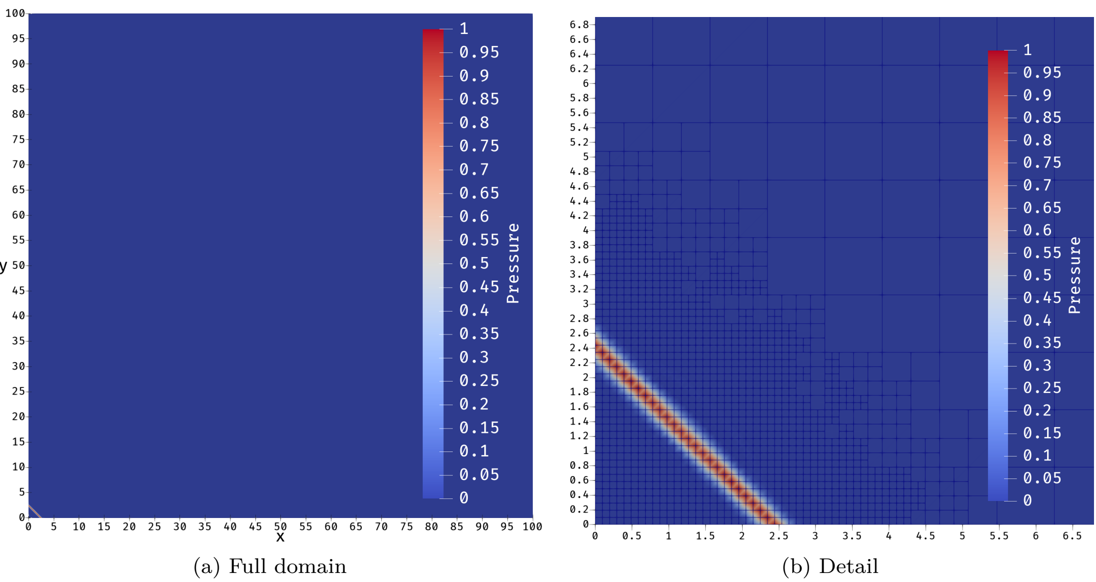
It is always faster to use AMR than to start with a uniform fully refined mesh, and it is faster to use AMR than a uniformly refined mesh that has the same maximum solution error. The adaptive case can refine elements where they are most needed, saving computing resources.

There is a balance to be made between refining often and spending more time in the AMR routine, and refining less often therefore using a less refined mesh for longer. Section 8.3 discusses ideas for alternative approaches to choose when to refine. In this case it seems that performing AMR every 20 time steps yields the lowest error, while waiting 100 timesteps between

refinements yields a faster runtime at the price of a worse error that still stays below the target error threshold we set.

7.5 Dynamic Load Balancing Performance

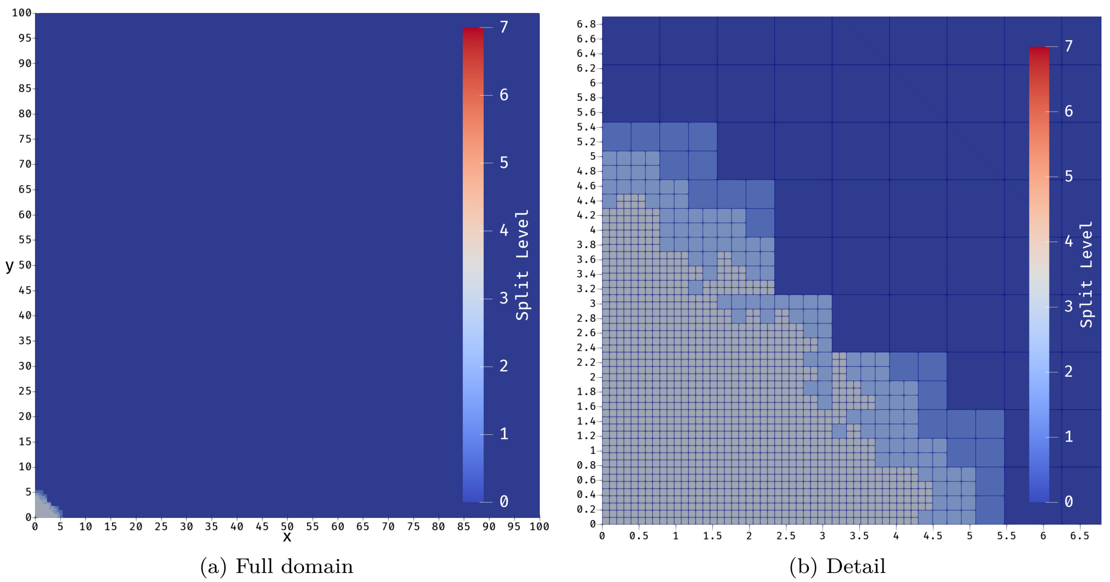
This section is dedicated to benchmarking the dynamic load balancing module of the program in adaptive cases where varying levels of load imbalance are generated. The sample problem is the same as described in Section 7.2, except that the domain size is increased from a size of 1 to 100 in both the x and y dimensions. The wave stays the same with $c = 1$, effectively only staying in the lower left corner during the program runtime. The program advances the solution to a simulation time of two seconds, with a Δt satisfying CFL $= 0.5$. The mesh is split into 128 elements in each of the x and y directions, for a total of $K = 16384$ elements. The initial polynomial order is $N = 4$. The problem is solved using four GPU nodes from the Narval supercomputer, for a total of $P = 16$ GPUs. This initially amounts to 1024 elements per GPU. The mesh is refined every 20 timesteps. Load balancing is performed according to two strategies described in Subsections 7.5.1 and 7.5.2 In order to prescribe a different load imbalance for different cases, a different maximum split level S is set. This parameter controls how many times an element can h-refine. Three cases are presented, a low load imbalance case with $S = 3$ shown in Figures 7.11 and 7.12, a medium load imbalance case with $S = 5$ shown in Figures 7.13 and 7.14, and a high load imbalance case with $S = 7$ shown in Figures 7.15 and 7.16. The different cases are allowed to refine up to their respective maximum split level S, and up to a polynomial order of $N = 12$.



(a) Full domain

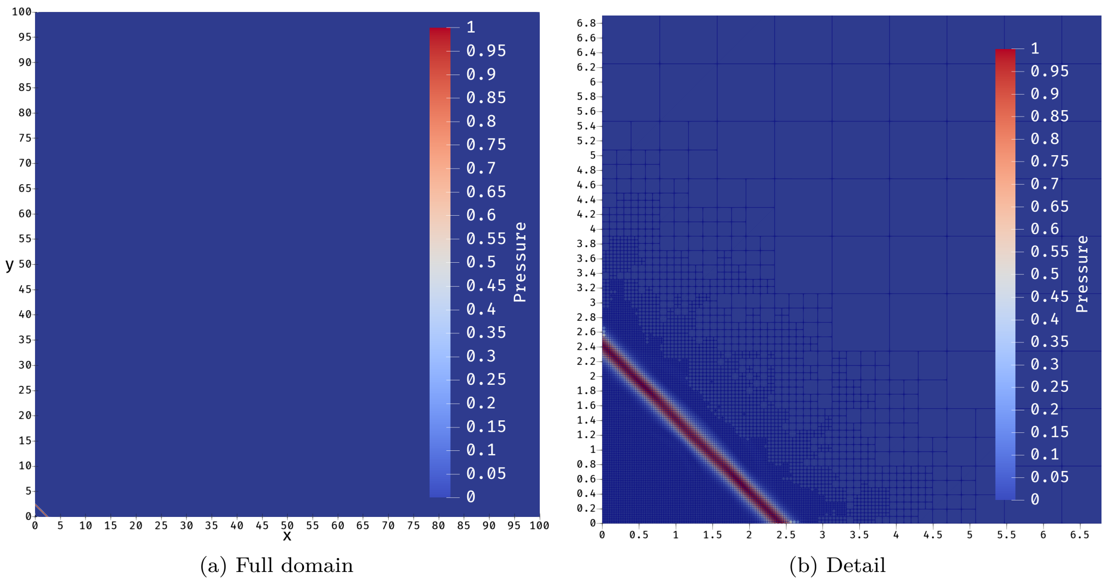
(b) Detail

Figure 7.11: Low load imbalance ($S = 3$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 17752$ (a) Complete domain (b) Area of interest



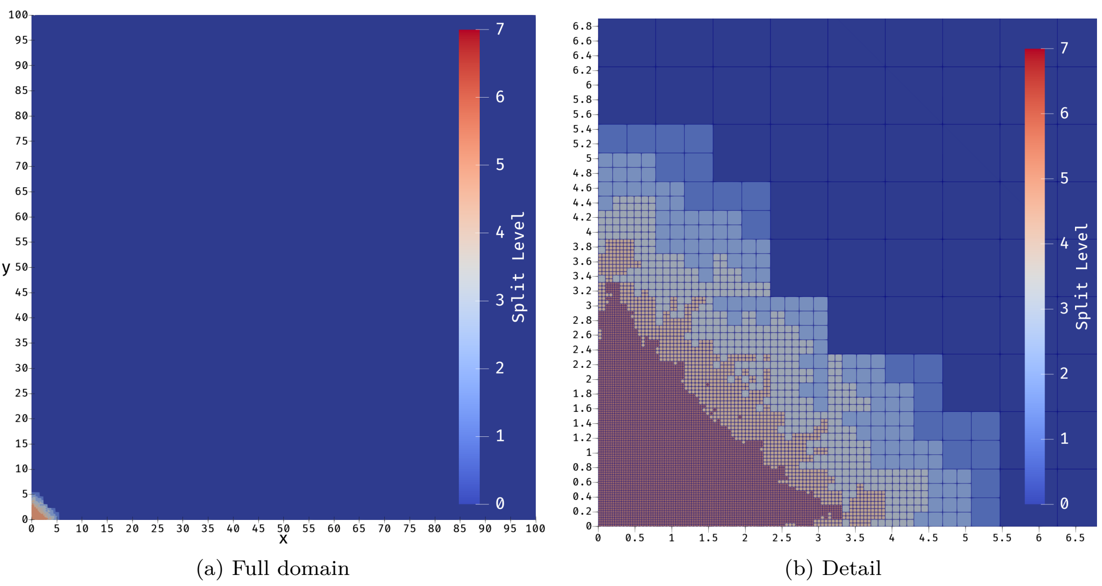
(a) Full domain (b) Detail

Figure 7.12: Low load imbalance ($S = 3$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 17752$ (a) Complete domain (b) Area of interest



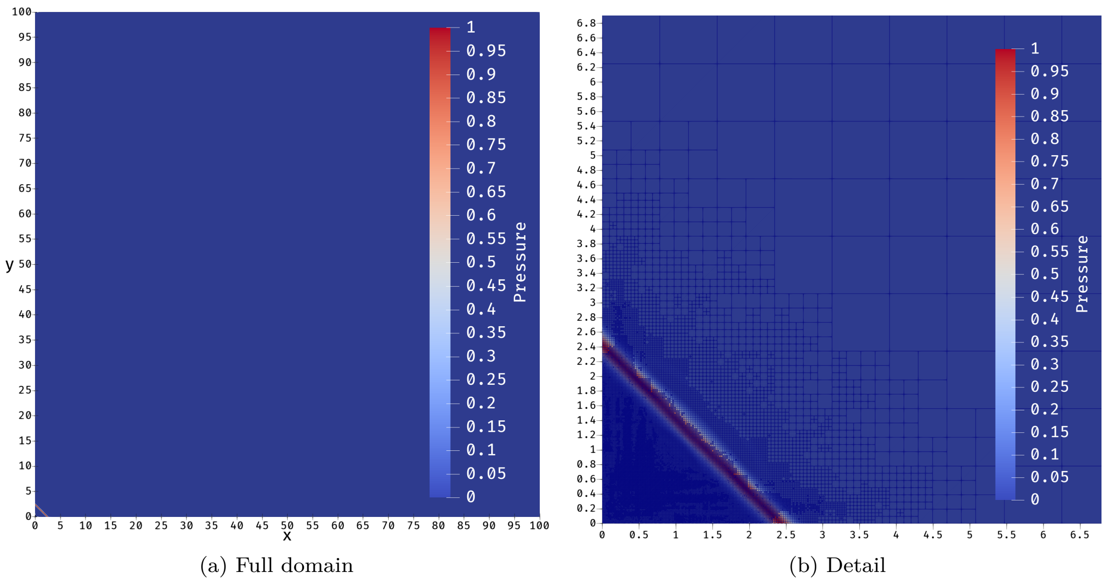
(a) Full domain (b) Detail

Figure 7.13: Medium load imbalance ($S = 5$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 26734$ (a) Complete domain (b) Area of interest



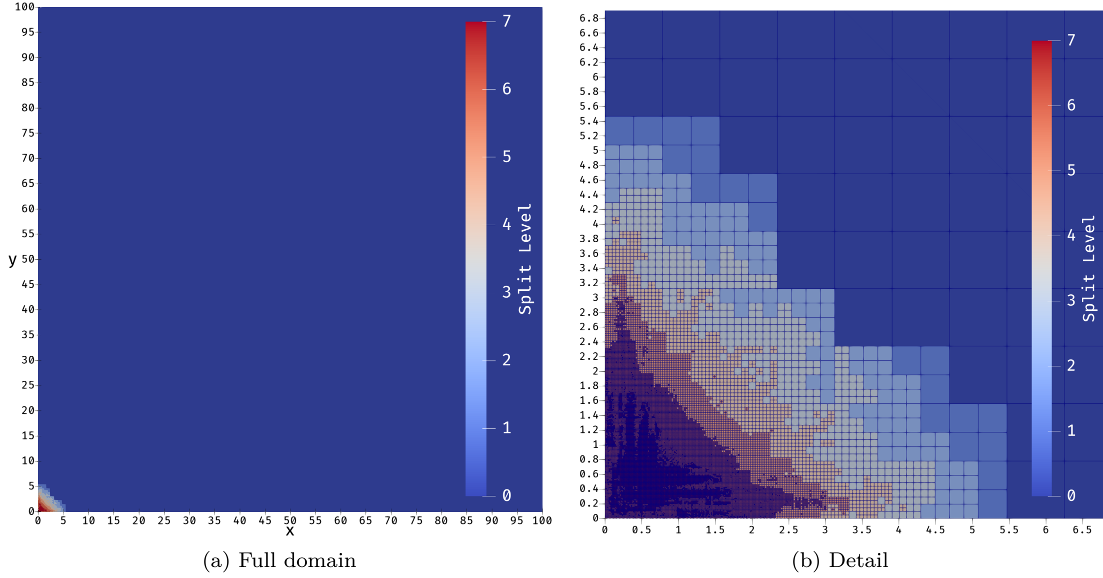
(a) Full domain (b) Detail

Figure 7.14: Medium load imbalance ($S = 5$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 26734$ (a) Complete domain (b) Area of interest



(a) Full domain (b) Detail

Figure 7.15: High load imbalance ($S = 7$) test case pressure: A wave passes through a very big domain. $K_{initial} = 16384$, $K_{final} = 54837$ (a) Complete domain (b) Area of interest



(a) Full domain (b) Detail

Figure 7.16: High load imbalance ($S = 7$) test case split level: Split level, indicating how many times the elements have split, only the bottom left refines. $K_{initial} = 16384$, $K_{final} = 54837$ (a) Complete domain (b) Area of interest

Table 7.1 describes the load imbalance of the three cases. S is the maximum number of times an element can split, K is the number of elements in the mesh, K_p is the number of elements in process p, w_{ideal} is the number of elements in a process if evenly distributed, and L is the load imbalance as described in Equation 6.6.

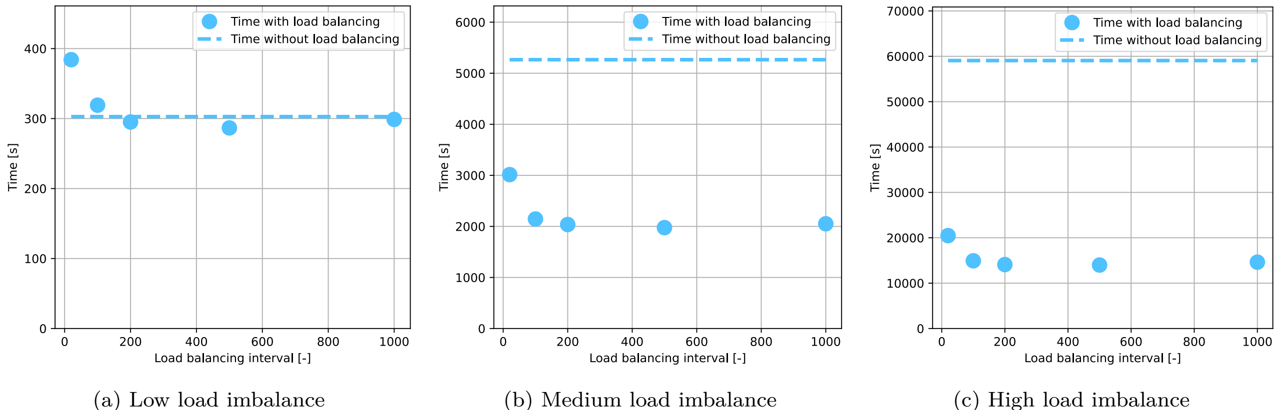
Case	S	K	Max K_p	w_{ideal}	L
Low	3	17752	2392	1110	2.16
Medium	5	26734	11374	1671	6.81
High	7	62490	47130	3906	12.07

Table 7.1: Load imbalance cases.

We compare two approaches to choose when to load balance the mesh, as described in Section 6.4. We compare the three cases above with and without load balancing.

7.5.1 Load balancing interval

We start by varying the load balancing interval for the three cases in Figures 7.17a, 7.17b and 7.17c. Since we refine the mesh every 20 time steps, a load balancing interval of 20 is equivalent to load balancing the mesh every time we refine it, up to every 50 times we refine for a load balancing interval of 1000. The dashed line represents the performance without load balancing.



(a) Low load imbalance

(b) Medium load imbalance

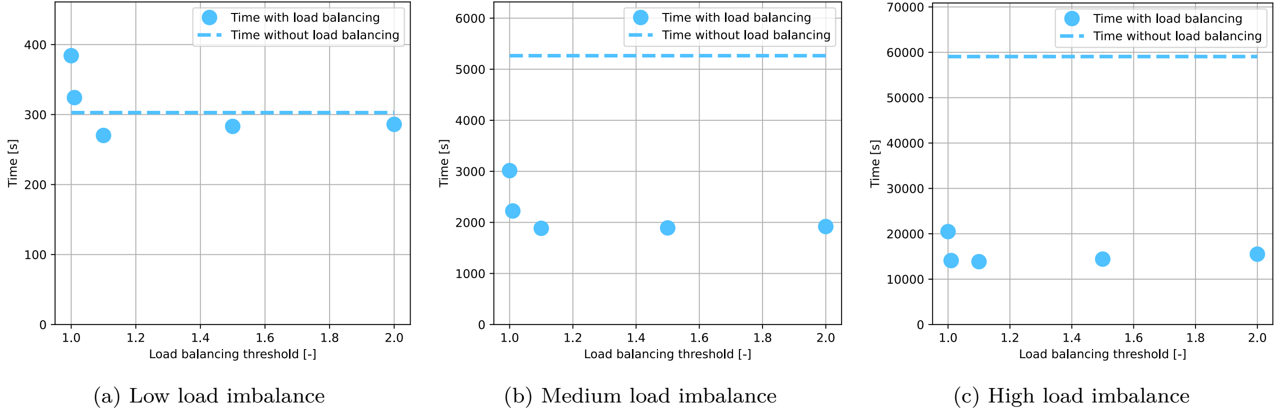
(c) High load imbalance

Figure 7.17: Load balancing performance interval test: Simulation time with refinement and load balancing while increasing load balancing interval. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval $= 20$, $P = 16$ (a) $S = 3$ (b) $S = 5$ (c) $S = 7$

From Figure 7.17, we see that load balancing diminishes the computation time of all three problems. The low load imbalance case needs a higher load balancing interval to offset the cost of the load balancing algorithm. In the other cases, the load imbalance is great enough that the cases with load balancing are always faster. It takes a certain load balancing interval to obtain the best performance, and that interval changes with the case. Next, we will study an alternative algorithm to improve on those two points.

7.5.2 Load balancing threshold

We now use an allowable load imbalance threshold to choose when to perform load balancing, as described in Section 6.4. This means we assess the load imbalance L of the mesh after each refinement step, and we only perform load balancing if it is above a certain threshold. We start with a threshold of 1, equivalent to load balancing after every refinement step, to 2, equivalent to load balancing only if a GPU has two or more times the workload it would have if the mesh was perfectly balanced. The same three load imbalance cases from Section 7.5 are shown in Figure 7.18.



(a) Low load imbalance

(b) Medium load imbalance

(c) High load imbalance

Figure 7.18: Load balancing performance threshold test: Simulation time with refinement and load balancing while increasing load balancing threshold. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval = 20, load balancing interval = 20, $P = 16$ (a) $S = 3$ (b) $S = 5$ (c) $S = 7$

From Figure 7.18, we observe that using a load imbalance threshold to control when we load balance is a much better solution than simply increasing the interval at which we perform load balancing. The performance improves quickly when increasing the threshold, and the threshold values that work well do not seem to change with the problem. A load imbalance threshold of around 1.1 seems to give good performance in all cases, and would not need to be adapted to different cases.

7.5.3 Overall load balancing performance

From the results of Subsections 7.5.1 and 7.5.2, we observe that load balancing at every occasion incurs a significant performance penalty, sometimes performing worse than the non load-balanced case in the low load imbalance case, in Figures 7.17a and 7.18a. The two other cases also show worse performance with a low load balancing interval or load imbalance threshold. This is because the dynamic load balancing process is expensive, especially since we are using GPUs, because of the numerous data transfers and memory allocations performed. We must choose judiciously our load balancing strategy.

The performance increases as we space out load balancing, either by increasing the load balancing interval, or by increasing the load imbalance threshold, up to a point. Past that point, an interval of 500 or a threshold of 1.5 in this case, the performance goes down due to the mesh being more imbalanced for longer periods of time.

The load imbalance threshold strategy from Subsection 7.5.2 gives better results: the performance improves quickly when increasing the threshold and stays high longer than the load balancing interval strategy from Subsection 7.5.1. The load balancing interval is also case dependent, and must be adjusted according to the scale of the problem and how often it is refined.

For these reasons we choose the load imbalance threshold strategy with a load imbalance threshold of 1.1 to assess the overall dynamic load balancing performance in Figure 7.19. It shows the speedup we get from load balancing the three problems. The x axis represents the equivalent load imbalance L of the problems if they were not load balanced.

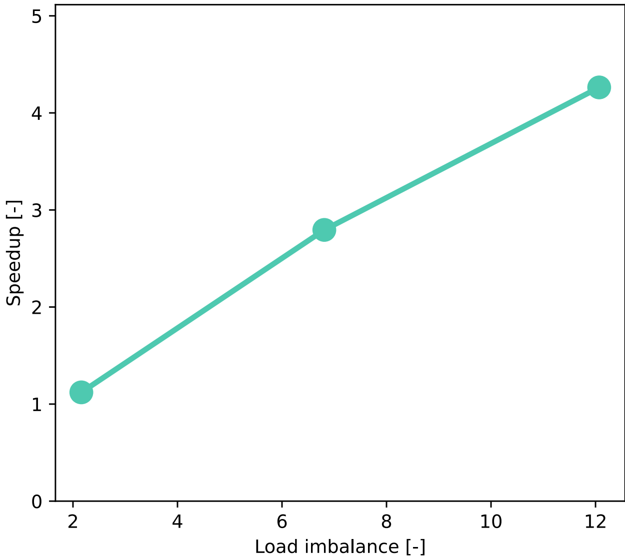


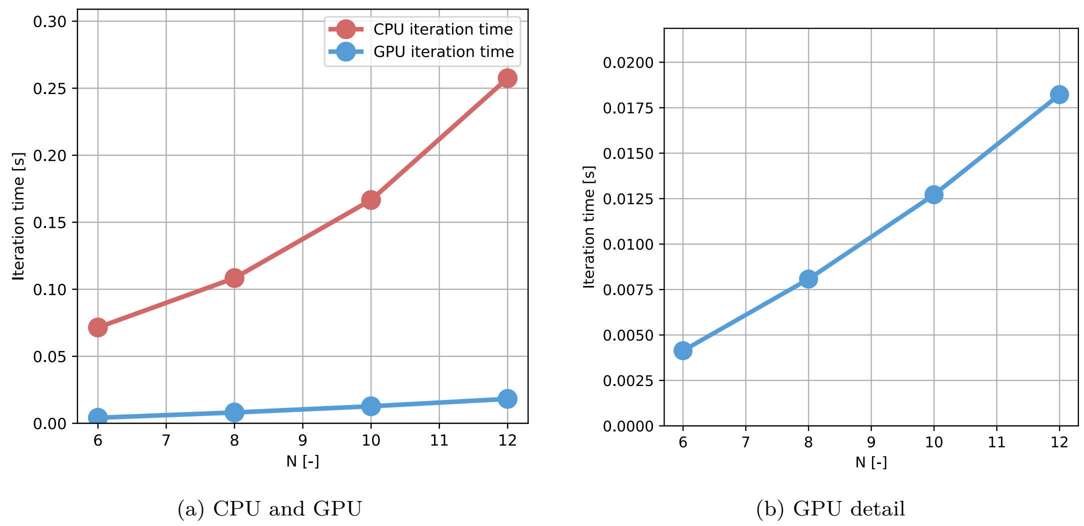
Figure 7.19: Load balancing performance: Simulation time with refinement and load balancing while increasing load imbalance. $N_{initial} = 4$, $K_{initial} = 16384$, refinement interval = 20, load balancing interval = 20, $P = 16$, load balancing threshold = 1.1

Figure 7.19 shows that dynamic load balancing performs well, and is able to regain performance lost to load imbalance. In the high load imbalance case, with a load imbalance of $L = 12.07$, dynamic load balancing can improve performance by $4.1\times$. It scales well with load imbalance, meaning it gives good results no matter the level of imbalance of the problem. The gains are especially crucial for high imbalance levels, where the load imbalance can degrade performance by an order of magnitude. In these cases, dynamic load balancing can regain most of the lost performance.

7.5.4 Polynomial order influence

Figure 7.20 shows a comparison of the iteration time as a function of the polynomial order N of a uniformly refined mesh, both on the CPU and GPU. This shows how computing time scales on the two types of processors with increasing N, and could guide how elements should be weighted individually when performing load balancing, instead of being all weighted equally as they are in this work. The test case was computed using the Narval supercomputer, using a grid with 64 elements in each of the the x and y directions. The initial polynomial order of the mesh is indicated on the x axis, and the problem is advanced by 1000 iterations in all cases.

AMR is performed every 100 iterations, but there are enough elements that the mesh never refines, it only estimates the error. The CPU case is computed on a single CPU core, and the GPU case is computed on a single GPU.



(a) CPU and GPU

(b) GPU detail

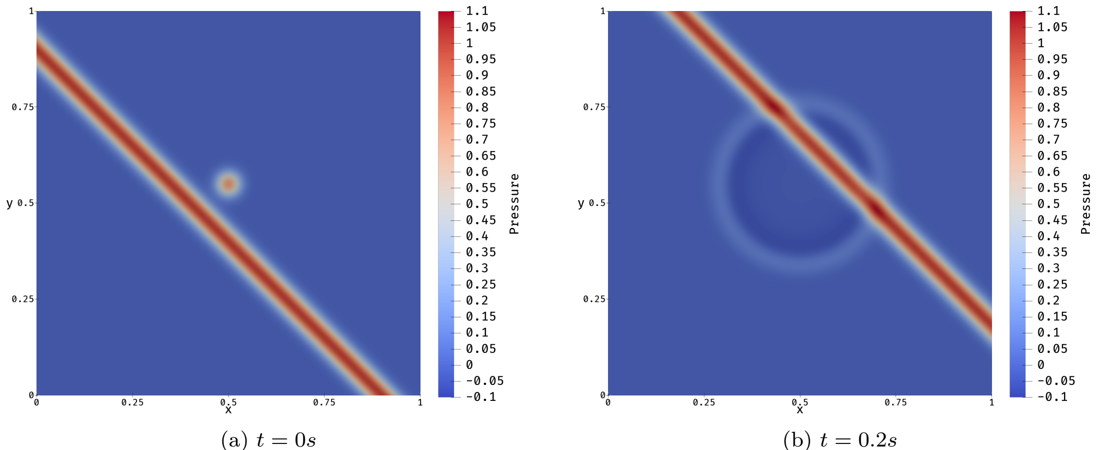
Figure 7.20: Polynomial order influence: Iteration time with increasing polynomial order N. (a) CPU and GPU. (b) GPU detail, emphasising the slope.

Figure 7.20 shows that the relation between iteration time and polynomial order N on the GPU is closer to being linear in this range. This is unexpected, as we expect the relation between polynomial order and computing time to scale with $(N+1)^2$ like on the CPU, as the number of operations to perform on every time step scales with $(N+1)^2$. This may be explained by the fact that GPUs favour more arithmetically intense computations. As N increases, there are more compute operations to perform relative to other operations like scheduling and launching kernels, which may increase the workload of GPUs and increase their performance. An increase in realisable GPU performance in GFLOPS has been observed when N increases, as for example by Chan et al. [12].

7.6 Complex Case

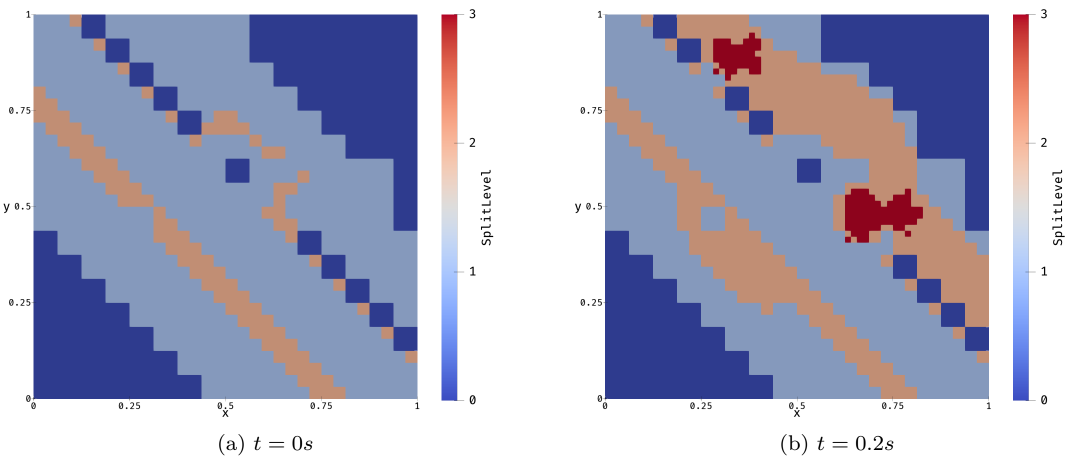
We now present a more complex case with all modules of the program enabled. Figure 7.21 shows the pressure distribution at two times. The problem depicts a diagonal wave moving at a wave speed $c = 1$ through a small circular "cloud" of higher pressure. The cloud expands and interacts with the wave. The Δt is chosen to satisfy $\text{CFL} = 0.5$, and the simulation is run until a solution time of $t = 0.2s$. The domain has a size of 1 in both dimensions, and has the analytical solution imposed to its outer edge as a boundary condition. The mesh is initially split into 16 elements in each of the x and y directions, for a total of $K = 256$ elements. The

polynomial order of the elements is initially $N = 4$. The solution is computed in parallel on 16 GPUs. The mesh is refined every 50 timesteps, and load balanced with an imbalance threshold of $L = 1.1$. It is allowed to refine up to $S = 3$ and $N = 16$. Three steps of mesh pre-condition are performed at an interval of 50 timesteps, which is why the $t = 0s$ mesh is already refined in Figures 7.22a and 7.23a. Figure 7.22 shows the split level S of the mesh, Figure 7.23 the polynomial order N of the mesh, and Figure 7.24 the rank of the elements. The rank indicates which of the 16 GPUs is responsible for which part of the mesh, and how this changes with time as the mesh is load balanced.



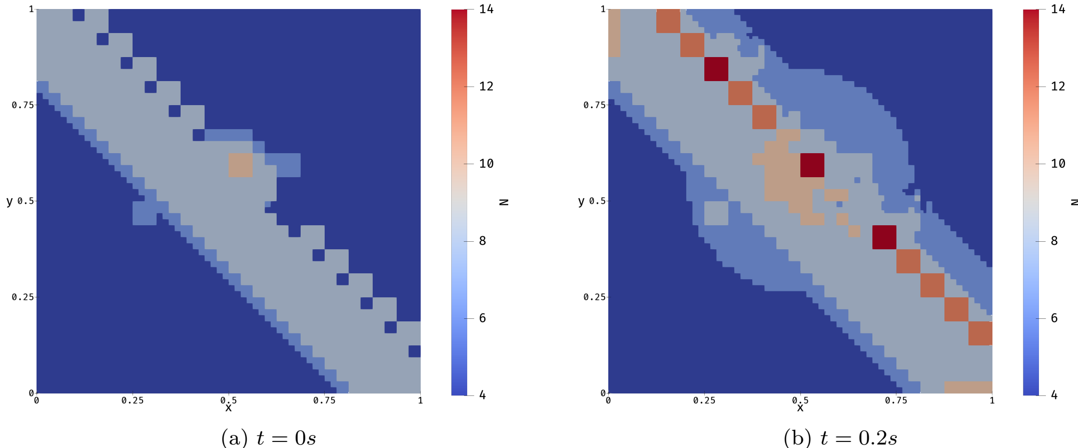
(a) $t = 0s$ (b) $t = 0.2s$

Figure 7.21: Complex case: A wave goes through a cloud of high pressure. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Initial conditions (b) After the wave and cloud collide



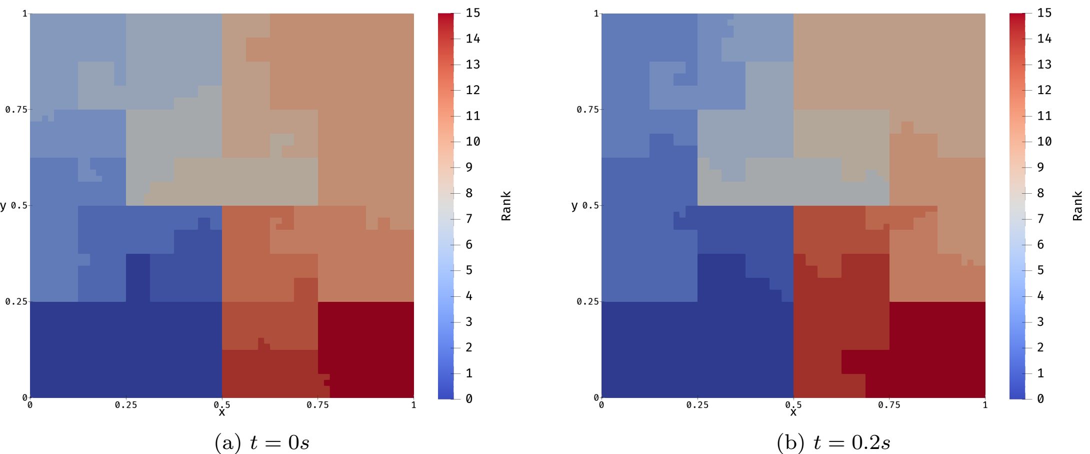
(a) $t = 0s$ (b) $t = 0.2s$

Figure 7.22: Complex case h-refinement: The elements split more where the cloud meets the wave. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide



(a) $t = 0s$ (b) $t = 0.2s$

Figure 7.23: Complex case p-refinement: The polynomial order increases more at the center of the cloud and where cloud meets the wave. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide



(a) $t = 0s$ (b) $t = 0.2s$

Figure 7.24: Complex case rank: As the mesh is refined load imbalance can occur, and elements are sent from one GPU to another when load balancing. $N_{initial} = 4$, $K_{initial} = 256$, $S = 3$, $P = 16$ (a) Start of time advancing (b) After the wave and cloud collide

These figures show that the pre-condition steps do a good job of refining the mesh to capture the initial conditions correctly in the $t = 0s$ figures (a). Afterwards, Figures 7.22b and 7.23b show that the mesh is refined along the movement of the linear wave and the expansion of the circular cloud, and is especially refined at their intersection. This indicates that the error estimation correctly identifies the more difficult areas of the solution. As more elements are created at the intersection of both waves, load imbalance occurs, and by the end of the computation there are $K = 5088$ elements. From Figures 7.22b and 7.24a, we can observe that there are more h-refining elements around GPUs 6 and 8, which are load balanced by $t = 0.2s$

in Figure 7.24b. The elements are moved around by dynamic load balancing, while keeping good locality and limiting the surface area between GPUs.

7.7 Complex Meshes

We now introduce a case on a complex geometry to show that the program works with more complex unstructured meshes that are not axis-aligned. A wave travels diagonally a wave speed $c = 1$ through a circular domain containing a NACA 0012 airfoil. The domain has a radius of 20 chords. The Δt is chosen to satisfy CFL = 0.5, and the simulation is run until a solution time of $t = 1.5s$. The boundary conditions are the analytical solution for the outer edge of the domain, and a reflection mimicking a wall for the airfoil surface. Figure 7.25 shows the initial mesh used for this case. Figure 7.26 shows the adapted mesh and Figure 7.27a depicts the wave moving across the airfoil. The mesh initially has $K = 2128$ elements, is allowed to h-refine up to $S = 3$, and by the end of the computation there are $K = 55680$ elements. The polynomial order of the elements is initially $N = 4$ and is allowed to p-refine up to $N = 16$. The mesh is refined every 1000 timesteps and load balanced with a imbalance threshold of $L = 1.1$. At the beginning of the computation, three pre-condition steps are performed at an interval of 1000 timesteps, which is why the mesh is already refined at $t = 0s$ in Figure 7.26a. The solution is computed in parallel on $P = 4$ GPUs.

The mesh is unstructured, and the original mesh file is numbered in a way that places elements with neighbouring indices very far apart in the 2D domain. This is not ideal, as it scatters memory accesses in a way that is detrimental to GPU performance, and increases dramatically the number of inter-GPU boundaries. To improve the numbering, the mesh was renumbered to a pseudo-Hilbert curve according to an algorithm presented in Appendix A. It is not a real Hilbert curve, as the Hilbert curve is only defined in 2D square domains with a power of two number of axis-aligned elements in both dimensions. Nonetheless, this pseudo-Hilbert curve resembles the Hilbert curve, and improves locality.

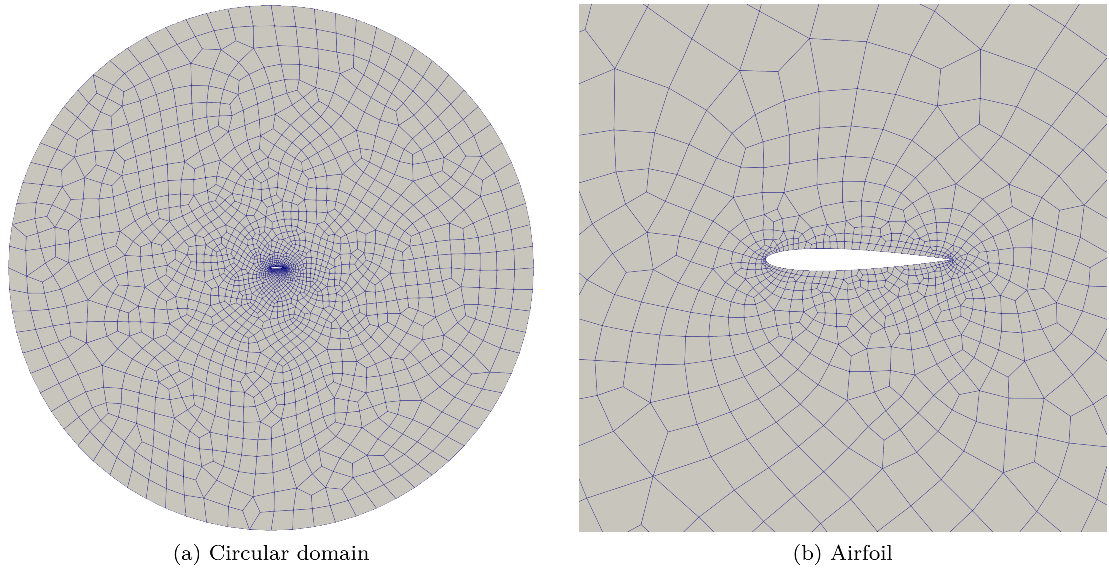


Figure 7.25: Complex mesh example: An airfoil in an unstructured circular domain with its initial mesh. $N_{initial} = 4$, $K_{initial} = 2128$, $S = 3$, $P = 4$ (a) Complete domain (b) Detail

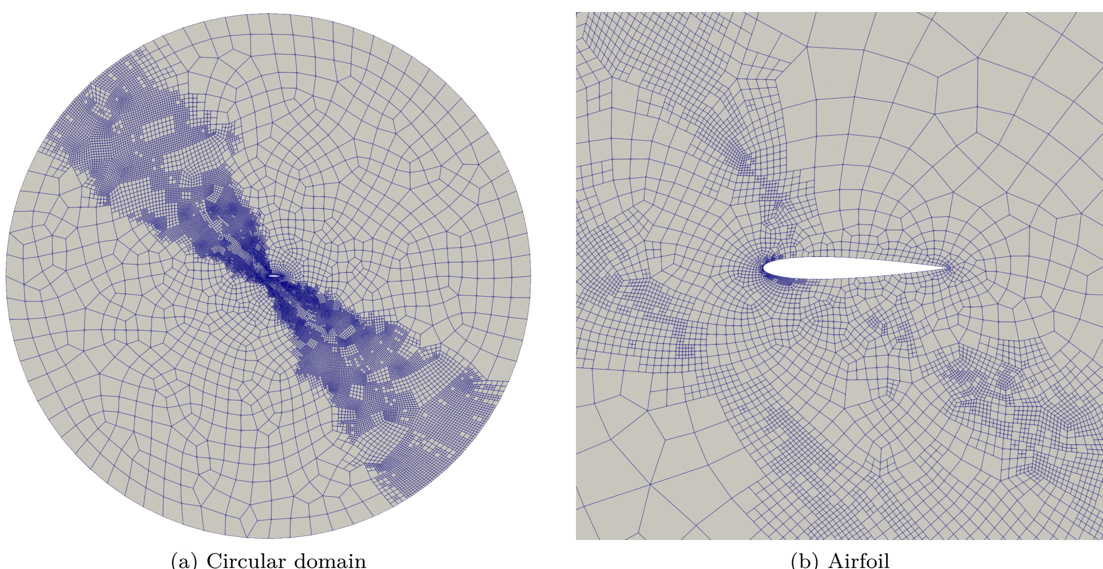
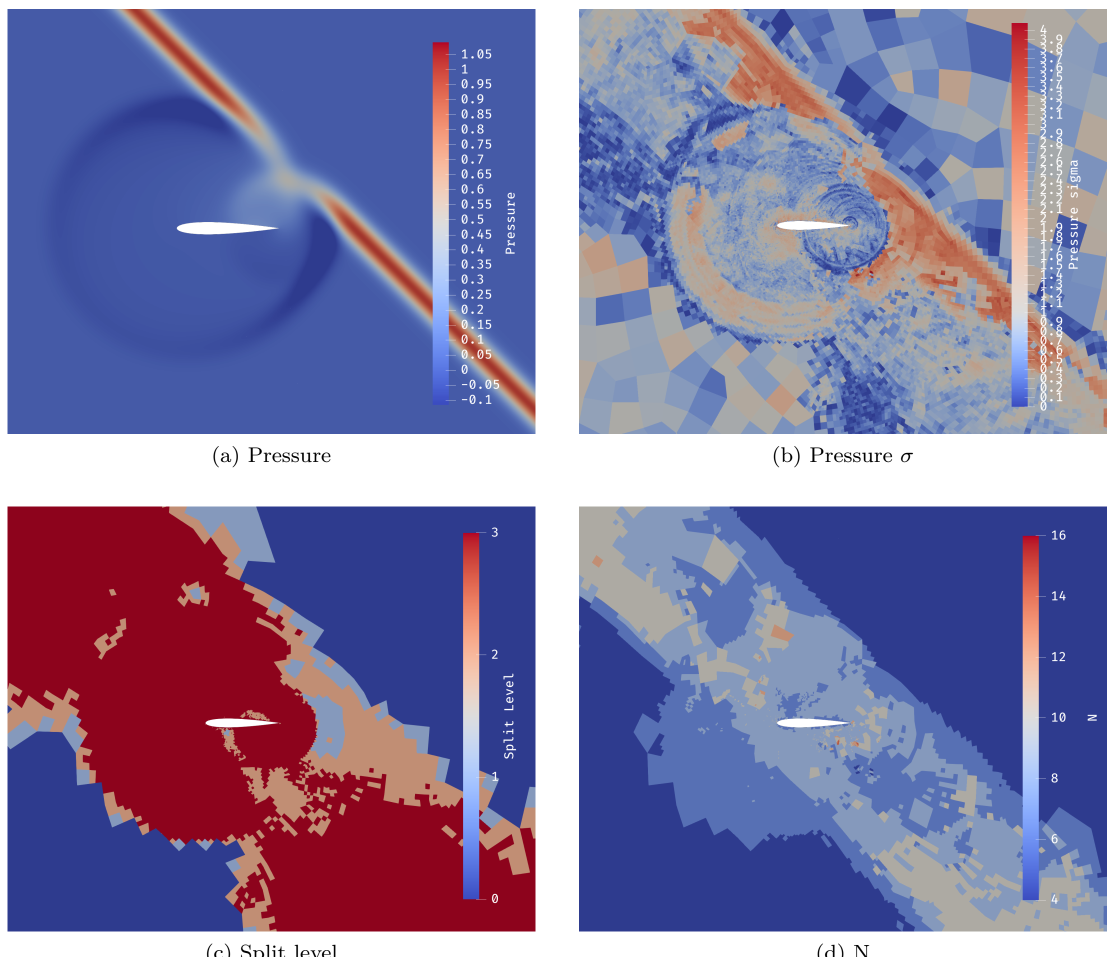


Figure 7.26: Complex mesh after pre-condition: Elements have been added to better apply initial conditions. $N_{initial} = 4$, $K = 24072$, $S = 3$, $P = 4$ (a) Complete domain (b) Detail

Figure 7.27 shows the problem after 1.5 seconds of solution time. Figure 7.27a shows the pressure distribution, the wave having passed around the airfoil and created a reflected circular wave. Figure 7.27b shows the pressure error decay rate σ, which informs the decision between h-refinement and p-refinement. An orange value, above 1, indicates p-refinement would be chosen, whereas a blue value, below 1, indicates h-refinement would be chosen. Figure 7.27c

shows the split level S, denoting how many times elements have h-refined. Figure 7.27d shows the polynomial order N, which increases when elements are p-refined.



(a) Pressure

(b) Pressure σ

(c) Split level

(d) N

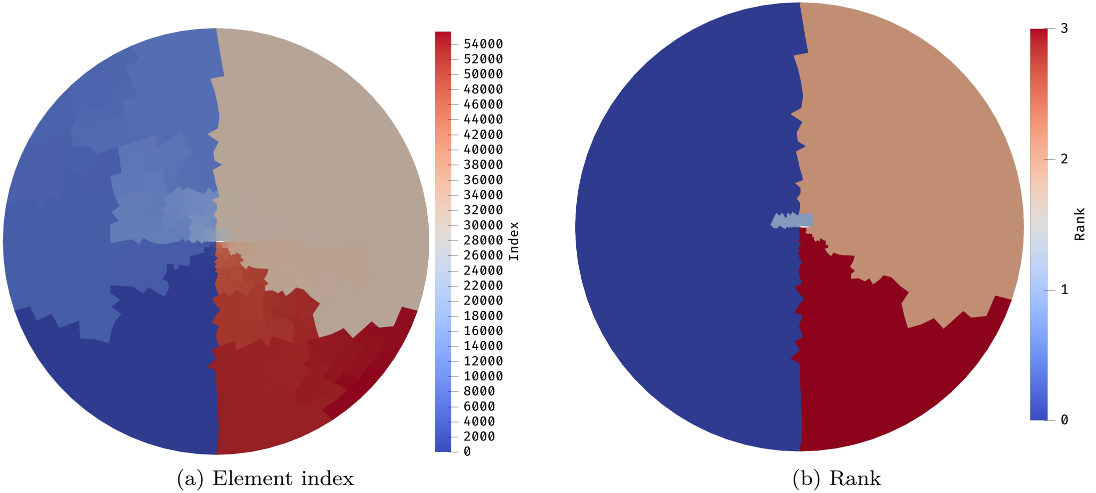
Figure 7.27: Complex mesh: A wave impinging on an airfoil at $t = 1.5s$. $N_{initial} = 4$, $K = 55680$, $S = 3$, $P = 4$ (a) Pressure (b) Pressure σ, prescribing h-refinement or p-refinement (c) h-refinement, the split level denotes how many times elements have split (d) p-refinement, polynomial order N

Figure 7.27 shows that the program has refined the mesh in more difficult areas of the domain, such as along the linear wave, close to the airfoil, and inside the circular wave. This is consistent with the fact that the interactions between the wave and the airfoil would create steeper areas in the pressure distribution.

This also shows that the program is able to compute solutions on complex geometries, including unstructured meshes where elements can be different from axis-aligned squares.

Figure 7.28 shows how the elements are arranged among the four GPUs. Figure 7.28a shows the indices of the elements, or how the elements are ordered along the pseudo-Hilbert

curve. Figure 7.28 shows the elements' rank, or in which GPU they are stored.



(a) Element index (b) Rank

Figure 7.28: Element distribution: How elements are distributed between GPUs. $N_{initial} = 4$, $K = 55680$, $S = 3$, $P = 4$ (a) Element indices, showing how they are ordered along the pseudo-Hilbert curve (b) Rank, showing which GPU contains which elements

In Figure 7.28, we can observe that the pseudo-Hilbert curve gives good locality to the elements, there are few discontinuities between elements, and the surface area between GPUs is small. The elements are well divided between the GPUs, as the GPUs in more heavily refined areas have a smaller footprint in the domain.

7.7.1 Profiling

This complex case was also profiled in order to assess which parts of the program were more computationally intensive. The simulation was run on four GPUs up to $t = 1s$ simulation time, refining every 1000 time steps and load balancing with a threshold of $L = 1.1$. Table 7.2 shows the result of this profiling session. This is a report of GPU time spent in the different kernels of the program. The report has a resolution of 0.1%, therefore most smaller kernels are not shown here. This could artificially reduce slightly the apparent time spent load balancing the mesh, as the load balancing module is made up of many smaller kernels, which may not show up in the report. The numbers are the average time spent between the four GPUs. There may also be profiler overhead, especially when memory is allocated and deallocated. This overhead could artificially inflate the time spent in memory allocation heavy parts of the program like AMR and load balancing.

Module	Time proportion (%)	Algorithm	Time proportion (%)
Solver	59.7	compute_derivative	26.68
		project_to_elements	19.2
		interpolate_to_edges	7.73
		rk3_step	3.75
		project_to_faces	1.35
		boundary_conditions	0.37
		compute_fluxes	0.35
		mpi_interfaces	0.2
		reduce_delta_t	< 0.1
AMR	38.5	hp_adapt	32.43
		split_faces	2.63
		split_mpi_interfaces	2.55
		split_boundaries	0.65
		p_adapt	0.15
		estimate_error	< 0.1
Load balancing	1.4	fill_received_elements	1.22
		create_mpi_boundaries	0.12
		create_neighbours	< 0.1
Other	0.7	empty_device_vector	0.73

Table 7.2: Program profiling: The relative GPU computation time spent in the three main modules, along with specific kernels and their relative GPU computation time.

This case ended up performing 281078 time steps, and took 5h04 to complete. The mesh was refined 281 times. These results show that the majority of the time is spent solving the problem and that the time spent load balancing the mesh is small. This demonstrates very good performance of the algorithm, with the only concerning result being the proportion of the computation time spent refining the mesh. The high time spent in the AMR routine could be reduced by refining the mesh less often, at the price of using a worse mesh for longer periods. Section 8.3 discusses a different approach to choose when to refine the mesh which could improve these situations by only refining the mesh when a global target error threshold is met.

Chapter 8

Conclusion

8.1 Summary

In this work, we presented a GPU-based wave equation solver using the discontinuous Galerkin spectral element method, as well as adaptive mesh refinement and dynamic load balancing. The main contributions of this work are: showing that AMR and load balancing on GPUs is possible and beneficial to solution quality and performance, showing that significant performance can be gained by performing computations using GPUs, and showing that GPUs accelerate spectral methods significantly.

The goal of using GPUs to perform these computations was to increase the available computing power, as spectral methods are notoriously expensive to compute. Spectral methods are useful when highly accurate solutions of complex flows are needed. Section 7.3 shows that a baseline case without AMR can be computed on a HPC platform up to three times faster using GPUs when they are sufficiently loaded. The program has also been shown to scale well on up to 64 GPUs. Chapter 3 discusses the architectural decisions taken to use the massively parallel GPU architecture. One such decision is to implement a data structure such that only high-level objects such as elements and faces are stored in flat arrays that can be transferred from the GPU to the CPU. The data arrays within those objects are instead allocated directly by the GPU in GPU code, and stored in dynamic memory. This allows for a more flexible data structure, where the GPU itself can perform some of the dynamic mesh changes necessary for AMR and load balancing, and reduces the overhead of moving objects around because the bulk of the data is stored outside of the objects themselves.

Some problems have localised areas where more precision is needed, such as shocks and boundary layers. Even with the increased processing power of GPUs, uniformly refining a mesh to the level needed by those areas makes the computing time increase to the point where it is not economic to solve these problems. We use adaptive mesh refinement to assess which parts

of the problem have a higher level of estimated error and to refine them, all while the program is running. Section 7.4 shows that a case with AMR can be more than 67× faster to execute than the same case initially uniformly refined to the same level as the AMR case. When compared to another case initially uniformly refined to the point that it reaches a similar maximum error to the AMR case, the AMR case is still generally faster. This is an excellent result, given that GPUs are not tailored to these kinds of dynamically changing computations.

The AMR process itself does not match well with the GPU architecture, as it reallocates and moves large amounts of memory, and is fundamentally sequential. Since each GPU is paired with a single CPU and is vastly faster, we want to execute as much as possible of the process in parallel on the GPU. We devised an algorithm to perform AMR in parallel, described in Chapter 5, offloading most of the computation to the GPU. Moving elements, h-refining, p-refining and the renumbering that comes with the process are all executed on the GPU. The only significant parts executed on the CPU are the allocation of the resized high-level arrays and the computing of offset arrays, so that the different GPU threads can operate in parallel without race conditions. Despite this, AMR can be a costly process on GPUs, for example in Subsection 7.7.1 where it makes up a significant portion of the total runtime.

We also implement a mesh pre-condition algorithm, described in Section 5.5, to refine the mesh before the computation up to a point where initial conditions are well resolved. Section 7.4 shows that performing a few pre-condition steps can significantly increase the solution quality when the starting mesh is very coarse. Combined with AMR, this means that meshes do not need to be very tailored to problems. A uniform coarse mesh can be used, and the pre-condition will refine the mesh to capture initial conditions correctly, while AMR refines the mesh as the solution is computed to capture the important areas of the flow.

To combat the load imbalance that can arise when the problem is solved in parallel using multiple GPUs, we implemented a dynamic load balancing algorithm. The algorithm uses the Hilbert curve, a space-filling curve that has good locality. This is especially important when using GPUs because data transfers between GPUs are more expensive, therefore we want to reduce the size of the boundaries between mesh blocks as much as possible. Section 7.5 shows that dynamic load balancing significantly increases performance when there is load imbalance. The performance increase scales well with load imbalance, meaning that dynamic load balancing should improve the performance no matter how much load imbalance is present. By judiciously choosing when we load balance, as explained in Section 6.4, the computing time spent in the load balancing algorithm can be very low, as reported in Subsection 7.7.1.

Load balancing is hampered by the same limitations as AMR, namely that it is not particularly well suited for the GPU architecture, and we want to execute as much of it as possible on the GPU. GPUs have less memory than CPUs, therefore we want each worker GPU to only have knowledge about the mesh block it is assigned in order to reduce its memory footprint. This complicates load balancing, as it is harder to reconstruct the mesh once elements are exchanged. Because of this, the load balancing algorithm is made up of several smaller dependent

functions, in order to catch every possible edge case. Most of those functions execute on the GPU, including the generation of the Hilbert curve ordering of the elements. This algorithm has a good performance, despite performing many data transfers between GPUs and reallocating a lot of memory. Chapter 6 details how the algorithm is designed, and what strategies were used to make it perform well on GPUs.

Another significant conclusion is that GPUs are particularly well suited to spectral methods, specifically. As described in Chapter 4, the number of operations to perform every time step scales with $(N+1)^2$, where N is the polynomial order of an element. This indicates that the computational complexity should scale with $(N+1)^2$. However, as reported in Subsection 7.5.4, it seems that increasing the polynomial order of elements increases the density of computations, which lends itself to better performance. The computation time relative to N seems closer to a linear relation than a quadratic one. This means that using higher order elements, with the benefit of higher accuracy, is more attractive on GPUs than on traditional platforms. It also points to the fact that spectral methods in general could have an increased speedup when using GPUs compared to other methods.

8.2 Concluding Remarks

The AMR and load balancing algorithms destined for GPUs are very different from those destined for CPUs only. Many different strategies must be used, such as optimising each and every part of the algorithm in terms of what can be executed in parallel on shared memory and what cannot. These parts should be executed on the GPU in order to improve performance and avoid unnecessary copies from the GPU to the CPU. Once these parallel parts have been identified, they must be modified until they operate on a single object type. For example, a kernel that is parallelised over elements and moves elements to a new index can modify the elements, arrays that are indexed per element, but it cannot modify faces. This is because multiple elements can link to the same face. If for example that kernel would update the element indices in the faces, a race condition would occur and the results would not be correct. In this case, the faces should update their element indices in another kernel, that parallelises over faces. There are many such issues that do not impact sequential code but start appearing when a few thousand threads execute in parallel.

Another noteworthy difference about GPU computing is that surprising things can have an impact on performance. One example is increasing the thread block size in Subsection 7.3.1. Increasing the block size should improve performance by reducing the number of instructions to dispatch. This is because there is a lot of divergence in the code, and with an increased block size more threads will be waiting for divergents threads. It is important to profile the code and look at which parts are taking up more time than they should. Sometimes some problematic functions can be made much faster, such as the error estimation routine that went from taking up 15% of the total execution time to 0.1% by precomputing a set of values.

The GPUs need to be saturated with work in order to have good performance. This can lead to peculiar situations like those in Subsection 7.3.1, where the relation between number of elements in a GPU and the computation time is not linear. Under a certain workload, the cores of a GPU do not all have work to perform, and the GPU is partly idle. On the other hand, if there is too much work in a GPU its memory may fill up, or the constant cache swapping from each core working on different elements successively may degrade performance. There is an ideal amount of work per GPU to be found, depending on the problem.

Thread divergence can also cause surprising results, such as a single if/else statement reducing the performance of a function by half. For example, the projection from faces to elements has four different paths, depending on if the face is forward or backward compared to the element, and if it is conforming or not. Since threads execute in groups of 32, threads that do not take a particular branch are stopped while the threads that took that branch execute, and then the next branch is evaluated. For our projection, if it happens that there are threads that take each of the four branches in a group, each of the branches will be executed one after the other with different threads inactive, taking up four times as much time as it should. This is a significant difference when programming for GPUs: branching must be minimised as much as possible.

It is also difficult to program dynamic meshes on the GPU. The fact that neither the CPU or GPU have the complete picture, and that all data has to be transferred from one to the other complicates some of the algorithms. Load balancing in particular required the most changes compared to a sequential CPU algorithm. In that algorithm, there are many dependencies between the different parts, which must be computed in a specific order. It is possible that it would be easier to re-create the mesh once elements have been exchanged, and rebuild all the connectivity from scratch. That approach was not chosen; instead the mesh is modified to remove unneeded parts and add new parts. Seeing that modifying the mesh with limited information is very difficult and error-prone, it is hard at this stage to say which approach is best.

8.3 Future Work

8.3.1 GPU Computing

There is still a lot of work to be done in order to better match the GPU architecture to the different modules of the program. For one, there is a lot of divergence in the code, due to AMR creating elements with different polynomial orders and non-conforming interfaces among other things. An avenue for future work would be to try to reduce divergence as much as possible. For example, sorting the faces by type and storing them in different arrays would allow different kernels to be launched for each type of face. In that case, each GPU thread in a kernel would execute the same instructions and there would be no divergence. Similarly, refining elements

per block would alleviate divergence, in that the elements assigned to a block of threads are always kept the same and there is no divergence.

Another approach would be to decompose the objects, like faces and elements, to a separate array for each member. This means changing from an “array of structures” to a “structure of arrays” data structure. This could help alleviate cache pressure, since GPUs have smaller caches than CPUs. This helps by reducing the amount of data that has to be loaded in cache since only the data members used can be loaded instead of whole objects. This could improve performance, especially when the GPU is very loaded.

There is also always work to be done by profiling the code and examining which parts of the code take up more execution time than they should, or are not optimal. Many tools exist to profile CUDA code. Two profilers were used to summarily examine the code for this work. The first is an overall profiler, the result of which is shown in Subsection 7.7.1. The other is a profiler for single kernels, which was used to improve the error estimation kernel. These tools give a lot of information, such as latency statistics, data dependencies, and optimal occupancy of the GPU.

There is no reason to use only CPUs or only GPUs. A future path to explore is to make a hybrid solver. If the program was modified to enable using both CPU and GPU workers together, it would be possible to use the complete processing power of computers. Since CPUs and GPUs have very different computing powers and capacities, this would make load balancing more difficult as elements would need to be split unequally between the workers. Since we implemented a CPU version of the program as a comparison, and both versions have the same interface between workers, it was possible to create a crude hybrid version to assess its potential. This tentative hybrid version is presented in Appendix B.

Finally, another area of work would be to add more fine-grained parallelism. As it stands, every process communicates with others via MPI. This happens regardless of if the other process is on the same machine or not. Workers on the same machine could access the same memory, and even transfer data between GPUs directly if they are connected together. We could use multithreading in addition to multiprocessing, where there is one process per machine and all workers on a machine execute via threads. This would completely remove necessary transfers between workers on the same machine, as they could all access the same data. This would also reduce the number of MPI transfers needed, as only one process per machine would need to communicate.

8.3.2 DG-SEM

In this work, we solved the wave equation. We use this simple equation in order to show the program works, but it is not very useful in itself. A possible avenue would be to upgrade the solver to solve a more interesting and complex equation. For example, solving the full Navier-Stokes equations would make the program useful for solving real-world problems.

Another possible improvement is using different shapes other than quadrilaterals, ideally within the same mesh. Many meshes use triangles, and some unstructured meshes use a mix of quadrilaterals and triangles. Some meshes even use elements with an arbitrary number of sides. Supporting these element types would make the program more general and able to work with more off-the-shelf meshes. This would mean changing the structure of the program to be generic over element types.

Speaking of elements, we often want to model curved surfaces, such as the airfoil from Subsection 7.7. In order to model curved surfaces with straight-edged elements, the elements need to be very small, and only approximate such surfaces. Implementing curved elements would help model these geometries better. An added advantage of curved elements is the possibility of smoothing the transition between very skewed elements, which becomes a problem with more complex equations.

Finally, it would be very interesting to transform the program from a 2D solver to a 3D solver. Many interesting problems appear only in 3D, and more complex geometries can be modeled. Also, 3D problems are an order of magnitude more computationally intensive to solve. These problems would benefit even more from the computational power of GPUs and would have no problem saturating them with work thanks to the increased complexity. With another dimension to fill, the number of elements that is possible to put in each dimension is reduced. This means that AMR is even more crucial in 3D, as computational resources are limited. The computational complexity increases from $(N+1)^2$ to $(N+1)^3$ in 3D, where N is the polynomial order. The scaling shown in Subsection 7.5.4 could help with that increased complexity.

8.3.3 AMR

There is also work to do to improve adaptive mesh refinement. Firstly, it takes up a large proportion of the total computation time. Maybe being more judicious about when we refine could alleviate this problem. There could be a global target error estimate below which the AMR routine is not launched, similar to the load balancing threshold described in Section 6.4. Care should be taken to not compromise efficiency by refining more often than necessary. It may also be possible to refine only some parts of the mesh, and avoid moving and reallocating the whole mesh.

As it stands now, we do not perform mesh coarsening in this work. This is an important avenue to explore, as it saves even more resources than refinement alone. In many cases, such as the one shown in Section 7.2, the areas that need refinement are not static in space. In that example, a wave goes through the domain diagonally. As it advances, the mesh is refined to better capture the wave. In its wake are left many very refined elements that do not contribute to the solution anymore, as the wave is not present in those areas. These elements take up computation power for no benefit. Mesh coarsening would do the inverse of h-refinement and p-

refinement, and either lower the polynomial order of elements or merge elements together. This would be a significant challenge to add to the program because of its flat arrays data structure. Many AMR programs store their elements in a tree data structure in order to keep track of which elements are the children of which and to be able to re-form elements that have split. In our data structure, elements resulting of h-refinement are just regular elements like any other, and have no knowledge of their previous topology. Coarsening would imply searching through nearby elements for other elements to be coarsened that can form a quadrilateral together. This is difficult, but brings the interesting ability of re-forming elements that did not exist previously.

8.3.4 Dynamic Load Balancing

The implementation of dynamic load balancing we present operates on the assumption that all elements have the same weight and are all equally as expensive to compute. With AMR, all elements can have a different polynomial order. As shown in Subsection 7.5.4, the polynomial order of elements influences the computation time. A way to load balance more fairly would be to give a weight to each element, and split that weight equally between workers. That weight could be influenced by an element's polynomial order and its number of neighbours. This is not trivial to add, as then all workers would need to be aware of the weight of all elements in order to know how to split the workload equally.

Since we would like to add a hybrid solver as stated in Subsection 8.3.1 and assign weights to elements of different complexity, the next logical step is to assign different capacities to workers. This is evident in the case of a hybrid solver, as a GPU has vastly more computing power than a CPU. By assigning a different capacity to each worker, GPU workers can be assigned more work than CPU workers. Even in the case of a GPU only solver, assigning capacity to workers would enable better load balancing on heterogenous platforms: for example, different computers with different kinds of GPUs working together, or full GPUs working with fractions of GPUs such as when sharing a system.

This capacity per worker could even be computed on the fly, with the program performing some iterations, and then examining how much time each worker took to perform such iterations. If we correlate time and the number of elements in each worker, the capacity of the workers could be computed. This would enable a hybrid solver to work well out of the box without doing trial runs or guessing the relative weight of specific CPUs and GPUs. This could even tailor the capacity of identical GPUs to unit-to-unit differences, or alleviate the load on thermally limited GPUs in systems where heat is an issue.

Bibliography

[1] A. AlOnazi, D. Keyes, A. Lastovetsky, and V. Rychkov. Design and Optimization of OpenFOAM-based CFD Applications for Hybrid and Heterogeneous HPC Platforms. *arXiv preprint arXiv:1505.07630*(), 2015. DOI: 10.25781/KAUST-9M51I.

[2] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. Association for Computing Machinery, 1967. DOI: 10.1145/1465482.1465560.

[3] M. Ameen, S. Patel, J. Colmenares, T. Chatterjee, and J. Chen. Direct Numerical Simulation (DNS) and high-fidelity large-eddy simulations for improved prediction of in-cylinder flow and combustion processes. *DOE Vehicle Technologies Office Annual Merit Review*():1–4, 2020.

[4] A. D. Beck, T. Bolemann, D. Flad, H. Frank, G. J. Gassner, F. Hindenlang, and C.-D. Munz. High-order discontinuous Galerkin spectral element methods for transitional and turbulent flow simulations. *International Journal for Numerical Methods in Fluids*, 76(8):522–548, 2014. DOI: https://doi.org/10.1002/fld.3943.

[5] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984. DOI: 10.1016/0021-9991(84)90073-1.

[6] G. L. Bryan, M. L. Norman, B. W. OShea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, and Y. Li. Enzo: An adaptive mesh refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 211(2):19, 2014. DOI: 10.1088/0067-0049/211/2/19.

[7] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011. DOI: 10.1137/100791634.

[8] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. D. Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin. Nektar plus plus : An open-source spectral/hp element framework. *Computer Physics Communications*, 192():205–219, 2015. DOI: 10.1016/j.cpc.2015.02.008.

[9] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999. DOI: 10.1109/4236.769420.

[10] 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. DOI: 10.1016/j.jcpx.2019.100023.

[11] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton. HipBone: A performance-portable GPU-accelerated C++ version of the NekBone benchmark(), 2022. DOI: 10.48550/ARXIV.2202.12477. URL: https://arxiv.org/abs/2202.12477.

[12] J. Chan, Z. Wang, A. Modave, J.-F. Remacle, and T. Warburton. GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *Journal of Computational Physics*, 318():142–168, 2016. DOI: 10.1016/j.jcp.2016.04.003.

[13] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010. DOI: 10.1109/IPDPS.2010.5470413.

[14] M. O. Deville, P. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2002. DOI: 10.1017/CBO9780511546792.

[15] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 47, 2004. DOI: 10.1109/SC.2004.26.

[16] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, and T. Warburton. NekRS, a GPU-Accelerated Spectral Element Navier-Stokes Solver. *CoRR*, abs/2104.05829(), 2021. URL: https://arxiv.org/abs/2104.05829 (visited on 02/21/2022).

[17] A. Garai, L. Diosady, S. Murman, and N. Madavan. DNS of Flow in a Low-Pressure Turbine Cascade Using a Discontinuous-Galerkin Spectral-Element Method. In *Turbo Expo: Power for Land, Sea, and Air*, volume 2B: Turbomachinery, 2015. DOI: 10.1115/GT2015-42773.

[18] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008. DOI: 10.1109/MM.2008.57.

[19] G. J. Gassner, A. R. Winters, and D. A. Kopriva. A well balanced and entropy conservative discontinuous Galerkin spectral element method for the shallow water equations. *Applied Mathematics and Computation*, 272():291–308, 2016. DOI: 10.1016/j.amc.2015.07.014.

[20] A. Giuliani and L. Krivodonova. Adaptive mesh refinement on graphics processing units for applications in gas dynamics. *Journal of Computational Physics*, 381():67–90, 2019. DOI: 10.1016/j.jcp.2018.12.019.

[21] N. Gödel, S. Schomann, T. Warburton, and M. Clemens. GPU Accelerated Adams–Bashforth Multirate Discontinuous Galerkin FEM Simulation of High-Frequency Electromagnetic Fields. *IEEE Transactions on Magnetics*, 46(8):2735–2738, 2010. DOI: 10.1109/TMAG.2010.2043655.

[22] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, and M. Min. Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016. DOI: 10.1007/s11227-016-1744-5.

[23] D. Gottlieb and S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. SIAM, Philadelphia, 1977. DOI: 10.1137/1.9781611970425.

[24] S. Gottlieb, C.-W. Shu, and E. Tadmor. Strong Stability-Preserving High-Order Time Discretization Methods. *SIAM Review*, 43():89–112, 2001. DOI: 10.1137/S003614450036757X.

[25] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, 1988. DOI: 10.1145/42411.42415.

[26] D. B. Haidvogel, E. Curchitser, M. Iskandarani, R. Hughes, and M. Taylor. Global Modelling of the Ocean and Atmosphere Using the Spectral Element Method. *Atmosphere-Ocean*, 35(sup1):505–531, 1997. DOI: 10.1080/07055900.1997.9687363.

[27] M. Harris et al. Optimizing parallel reduction in CUDA. *Nvidia Developer Technology*, 2(4):70, 2007.

[28] H. J. Haverkort. An inventory of three-dimensional Hilbert space-filling curves. *CoRR*, abs/1109.2323(), 2011. URL: http://arxiv.org/abs/1109.2323.

[29] S. He. *Dynamic Load Balancing for a hp-adaptive Discontinuous Galerkin Wave Equation Solver via Spacing-Filling Curve and Advanced Data Structure*, University of Ottawa thesis, 2021. DOI: http://dx.doi.org/10.20381/ruor-26256.

[30] J. S. Hesthaven and T. Warburton. Nodal High-Order Methods on Unstructured Grids: I. Time-Domain Solution of Maxwell's Equations. *Journal of Computational Physics*, 181(1):186–221, 2002. DOI: https://doi.org/10.1006/jcph.2002.7118.

[31] J. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, New York City, 2007.

[32] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38():459–460, 1891. URL: http://www.digizeitschriften.de/dms/img/?PPN=PPN235181684_0038&DMDID=dmdlog40.

[33] G. Karniadakis and S. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press on Demand, Oxford, 2005. DOI: 10.1093/acprof:oso/9780198528692.001.0001.

[34] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20():359–392, 1998.

[35] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Computer Science & Engineering (CS&E) Technical ReportS TR 97-061, University of Minnesota, 1997.

[36] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Computer Science & Engineering (CS&E) Technical Report TR 97-060, University of Minnesota, 1997.

[37] A. M. Khokhlov. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics*, 143(2):519–543, 1998. DOI: 10.1006/jcph.1998.9998.

[38] E. Kijsipongse and S. U-ruekolan. Dynamic load balancing on GPU clusters for large-scale K-Means clustering. In *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*, pages 346–350, 2012. DOI: 10.1109/JCSSE.2012.6261977.

[39] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009. DOI: 10.1016/j.jcp.2009.06.041.

[40] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, 1988. DOI: 10.1007/BF01887592.

[41] D. A. Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. Springer, New York City, 2009.

[42] 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. DOI: 10.1016/S0010-4655(99)00501-9.

[43] Y. Maday, C. Mavriplis, and A. T. Patera. Nonconforming mortar element methods - Application to spectral discretizations. In *Domain Decomposition Methods*, pages 392–418, Philadelphia. SIAM, 1989.

[44] C. Mavriplis. A Posteriori Error Estimators for Adaptive Spectral Element Techniques. In P. Wesseling, editor, *Proceedings of the Eighth GAMM-Conference on Numerical Methods in Fluid Mechanics*, pages 333–342. Vieweg+Teubner Verlag, 1990.

[45] D. S. Medina, A. St.-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages. *CoRR*, abs/1403.0968(), 2014. URL: http://arxiv.org/abs/1403.0968 (visited on 02/21/2022).

[46] G. Mengaldo, D. Moxey, M. Turner, R. C. Moura, A. Jassim, M. Taylor, J. Peiró, and S. J. Sherwin. Industry-Relevant Implicit Large-Eddy Simulation of a High-Performance Road Car via Spectral/hp Element Methods. *CoRR*, abs/2009.10178(), 2020. URL: https://arxiv.org/abs/2009.10178.

[47] E. Merzari, P. Fischer, M. Min, S. Kerkemeier, A. Obabko, D. Shaver, H. Yuan, Y. Yu, J. Martinez, L. Brockmeyer, et al. Toward exascale: overview of large eddy simulations and direct numerical simulations of nuclear reactor flows with the spectral element method in Nek5000. *Nuclear Technology*, 206(9):1308–1324, 2020.

[48] E. Merzari, A. Obabko, P. Fischer, N. Halford, J. Walker, A. Siegel, and Y. Yu. Large-scale large eddy simulation of nuclear reactor flows: Issues and perspectives. *Nuclear Engineering and Design*, 312():86–98, 2017. DOI: 10.1016/j.nucengdes.2016.09.028. 16th International Topical Meeting on Nuclear Reactor Thermal Hydraulics.

[49] A. Molcard, N. Pinardi, M. Iskandarani, and D. B. Haidvogel. Wind driven general circulation of the Mediterranean Sea simulated with a Spectral Element Ocean Model. *Dynamics of Atmospheres and Oceans*, 35(2):97–130, 2002. DOI: 10.1016/S0377-0265(01)00080-X.

[50] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015. DOI: 10.1137/140980260.

[51] B. A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997. DOI: 10.1109/2.612253.

[52] Nvidia. CUDA C++ Programming Guide. 2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 11/29/2021).

[53] A. Obabko, E. Tsyrulnykov, R. Rainsberger, A. V. Torreira, H. Nagib, A. Agarwal, and P. F. Fischer. Large-Eddy Simulation of Flows Through a Novel Vascular Access Device for Hemodialysis Access. In *APS Division of Fluid Dynamics Meeting Abstracts*, A11–008, 2017.

[54] N. Offermans, O. Marin, M. Schanen, J. Gong, P. F. Fischer, P. Schlatter, A. Obabko, A. Peplinski, M. Hutchinson, and E. Merzari. On the Strong Scaling of the Spectral Element Solver Nek5000 on Petascale Systems. *CoRR*, abs/1706.02970(), 2017. URL: http://arxiv.org/abs/1706.02970 (visited on 02/21/2022).

[55] N. Offermans, A. Peplinski, O. Marin, P. F. Fischer, and P. Schlatter. Towards Adaptive Mesh Refinement for the Spectral Element Solver Nek5000. In *Direct and Large-Eddy Simulation XI*, pages 9–15, New York City. Springer International Publishing, 2019.

[56] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. DOI: 10.1109/JPROC.2008.917757.

[57] J. Parkhurst, J. Darringer, and B. Grundmann. From Single Core to Multi-Core: Preparing for a New Exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pages 67–72. Association for Computing Machinery, 2006. DOI: 10.1145/1233501.1233516.

[58] A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984. DOI: 10.1016/0021-9991(84)90128-1.

[59] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36():157–160, 1890. URL: http://www.digizeitschriften.de/dms/img/?PPN=PPN235181684_0036&DMDID=dmdlog13.

[60] A. Peplinski, P. F. Fischer, and P. Schlatter. Parallel Performance of h-Type Adaptive Mesh Refinement for Nek5000. In *Proceedings of the Exascale Applications and Software Conference 2016*. Association for Computing Machinery, 2016. DOI: 10.1145/2938615.2938620.

[61] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004. DOI: 10.1016/j.jpdc.2004.05.003.

[62] T. Plewa, T. Linde, and V. G. Weirs. *Adaptive Mesh Refinement - Theory and Applications*. Springer, New York City, 2003.

[63] M. Rahman, R. Agarwal, M. Lampinen, and T. Siikonen. An Improved Version of One-Equation RAS Turbulence Model. In *45th AIAA Fluid Dynamics Conference*, 2015. DOI: 10.2514/6.2015-2786.

[64] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. LA-UR-73-479, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[65] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457–484, 2010. DOI: 10.1088/0067-0049/186/2/457.

[66] H.-Y. Schive, J. A. ZuHone, N. J. Goldbaum, M. J. Turk, M. Gaspari, and C.-Y. Cheng. gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability. *Monthly Notices of the Royal Astronomical Society*, 481(4):4815–4840, 2018. DOI: 10.1093/mnras/sty2586.

[67] Siemens. Whistle while you mesh: Simcenter STAR-CCM+ model-driven adaptive mesh refinement (AMR). 2020. URL: https://blogs.sw.siemens.com/simcenter/whistle-while-you-mesh-simcenter-star-ccm-model-driven-adaptive-mesh-refinement-amr (visited on 02/11/2022).

[68] SimFlow. RapidCFD GPU - OpenFOAM on GPU. 2020. URL: https://sim-flow.com/rapid-cfd-gpu/ (visited on 02/21/2022).

[69] J. Slotnick, A. Khodadoust, J. Alonso, W. Gropp, and D. Mavriplis. CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences. CR–2014-218178, NASA, 2014. URL: http://www.sti.nasa.gov.

[70] F. Stern, R. V. Wilson, H. W. Coleman, and E. G. Paterson. Comprehensive approach to verification and validation of CFD simulations—part 1: methodology and procedures. *J. Fluids Eng.*, 123(4):793–802, 2001.

[71] E. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer Science & Business Media, 2009. DOI: 10.1007/b79761.

[72] M. Vadsola, G. G. Agbaglah, and C. Mavriplis. Slat cove dynamics of low Reynolds number flow past a 30P30N high lift configuration. *Physics of Fluids*, 33(3):33607, 2021. DOI: 10.1063/5.0036088.

[73] F. Wang, N. Marshak, W. Usher, C. Burstedde, A. Knoll, T. Heister, and C. R. Johson. CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data. *Computer Graphics Forum*(), 2020. DOI: 10.1111/cgf.13958.

[74] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993. DOI: 10.1109/71.243526.

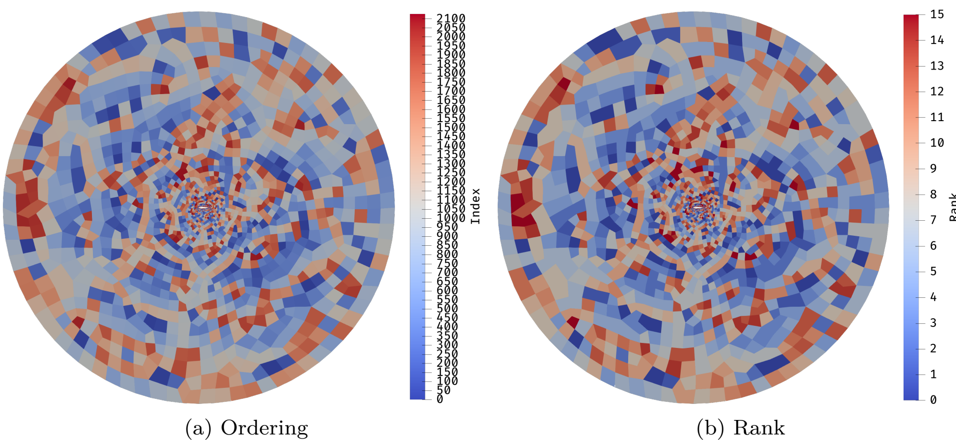
[75] J. H. Williamson. Low-storage Runge-Kutta schemes. *Journal of Computational Physics*, 35(1):48–56, 1980. DOI: 10.1016/0021-9991(80)90033-9.

[76] S. Wu, S. S. Patel, and M. M. Ameen. Investigating the Origins of Cyclic Variability in Internal Combustion Engines Using Wall-Resolved Large Eddy Simulations. In *Internal Combustion Engine Division Fall Technical Conference*, volume 85512, V001T06A003, 2021.

Appendix A

Mesh Renumbering

The DG-SEM solver described in this work can process unstructured meshes, which makes it more general and allows computations on pre-constructed meshes. The program utilises the Hilbert curve, a space-filling curve, to preserve locality and reduce the size of inter-GPU boundaries. The curve is described in Section 6.1. These two capabilities are at odds, as the overwhelming majority of meshes created for other purposes will not be numbered according to the Hilbert curve. Also, the Hilbert curve is only defined for square domains with n axis-aligned quadrilateral elements in the x and y directions, where n is a power of two. Therefore, most off-the-shelf meshes will not inherit the good locality and small boundaries between mesh blocks that we put a lot of effort to achieve in Chapter 6. Moreover, there is a performance penalty when using a single GPU if elements that are close in the ordering are not geometrically close, as memory access patterns are compromised. An extreme example is the original mesh used for Section 7.7. Figure A.1 shows the ordering of the elements and how the mesh is split into mesh blocks if partitioned into 16 blocks as-is.

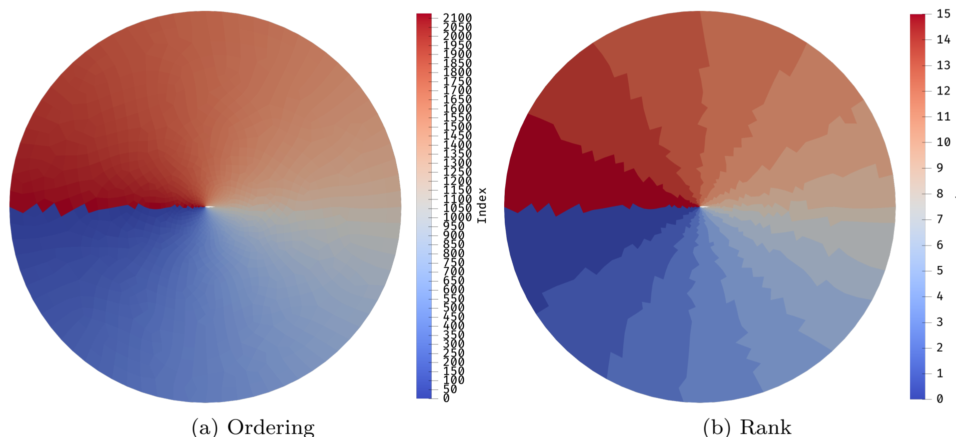


(a) Ordering (b) Rank

Figure A.1: Unsorted mesh: The mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned

The resulting mesh blocks are scattered around the mesh, leading to unnecessarily large boundaries between blocks. In order to get good performance, we will need to renumber the mesh.

Since this mesh is circular, and the hole representing the airfoil is in the middle, a simple sorting algorithm could represent the elements in polar coordinates, and sort them according to their angle around the center. The resulting ordering and partitioning are shown in Figure A.2. The index represents how the elements are ordered in the mesh, and the rank indicates which mesh block they are part of.



(a) Ordering (b) Rank

Figure A.2: Circular sorted mesh: The elements are sorted according to their angle and the mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned

This method seems to give reasonable results, but is only useful for circular meshes. Also,

the partitions it generates have good grouping for a low number of blocks, but as the number of blocks increases the blocks become shaped like thin slices. This is not ideal, as it increases the size of inter-block boundaries. Figure A.3 shows the ordering of elements more clearly.

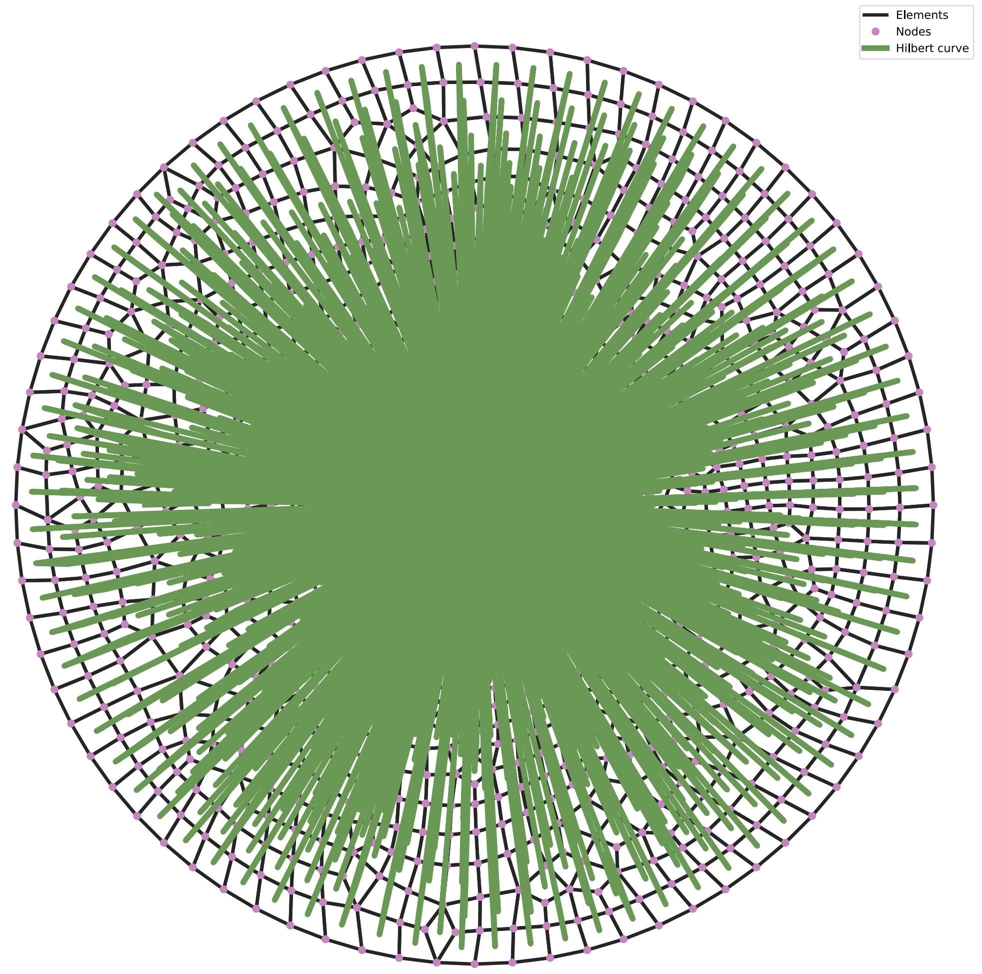


Figure A.3: Circular sorted mesh element ordering: The elements are very far from their neighbours.

This is evidently not a good result. Even if this method creates reasonable partitions, the elements are not necessarily close to their neighbours, which degrades performance.

What we really want is to number the elements according to the Hilbert curve, which gives good results. As the Hilbert curve is not defined for unstructured geometries like this one, we will have to approximate it. First, we perform an elliptic mapping to transform the circular geometry to a square. We then generate a Hilbert curve over that square, and number the elements by the 1D index of where their center falls on the Hilbert curve when rounded to

the closest point on the curve. By sorting the elements by that index, they will be numbered according to the Hilbert curve. If two elements fall to the same index, they are sorted by their initial index. To avoid this we generate a very fine, or high level, Hilbert curve. Figure A.4 shows the underlying curve along which the elements are sorted.

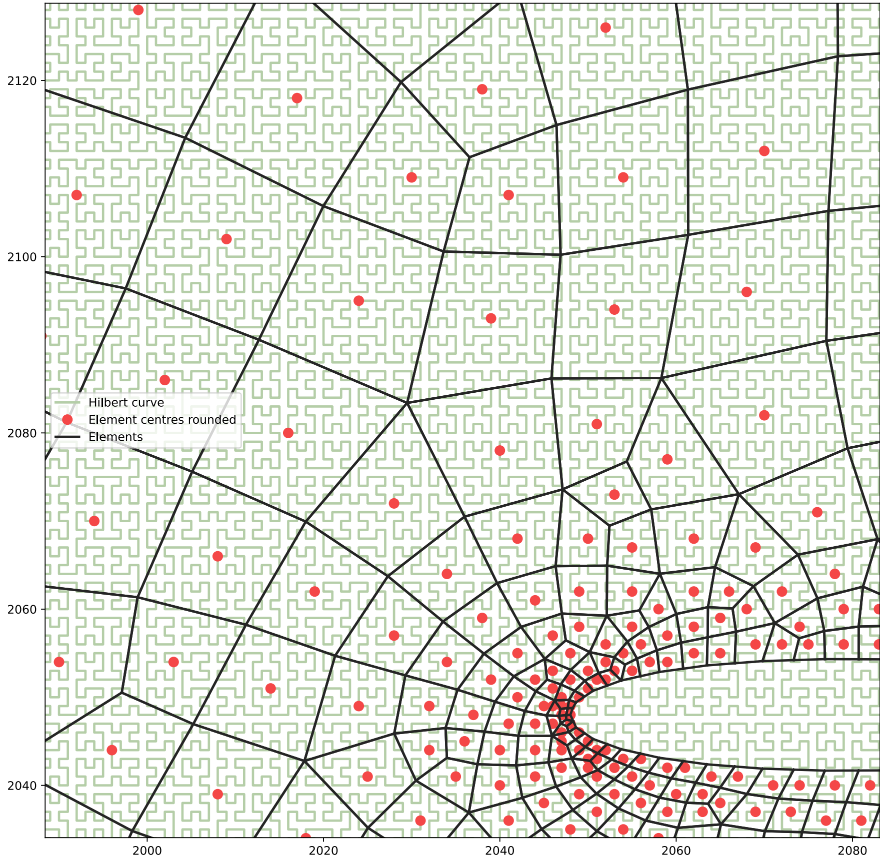
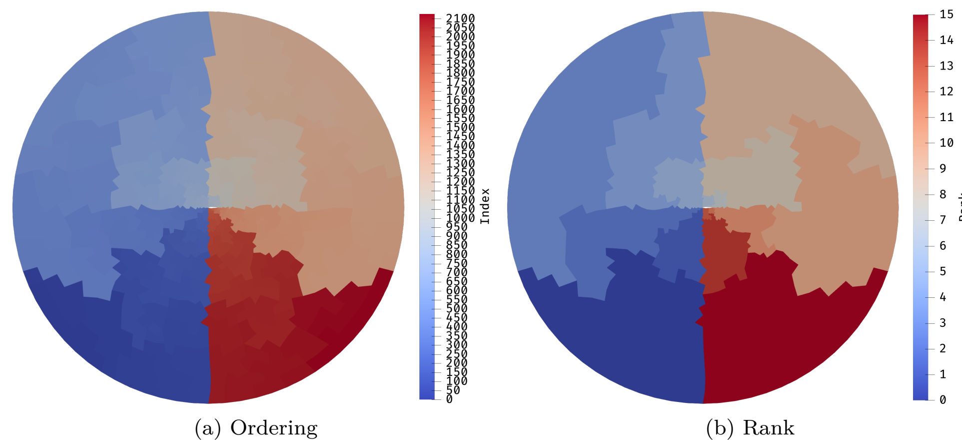


Figure A.4: Hilbert renumbering: The element centres are rounded to an underlying Hilbert curve.

When we use this method on the same mesh as Figure A.1, we obtain a mesh sorted along a pseudo-Hilbert curve. Figure A.5 shows the ordering of the elements as well as the partitions generated.



(a) Ordering (b) Rank

Figure A.5: Pseudo-Hilbert sorted mesh: The elements are sorted according to their index along a Hilbert curve, and the mesh is partitioned into 16 mesh blocks. (a) Ordering of the elements within the mesh (b) Mesh blocks once partitioned

Figure A.5a shows an ordering that looks a lot like the correct Hilbert curve, shown in Figure A.6. The grouping of mesh blocks in Figure A.5b is good, and has short inter-block interfaces.

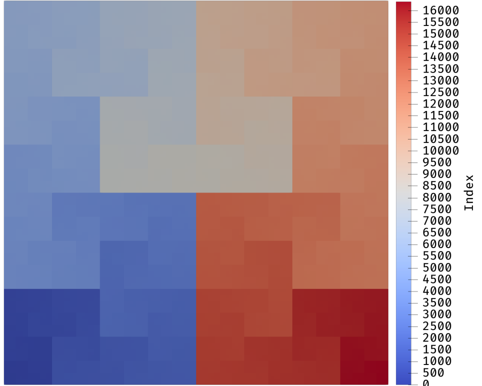
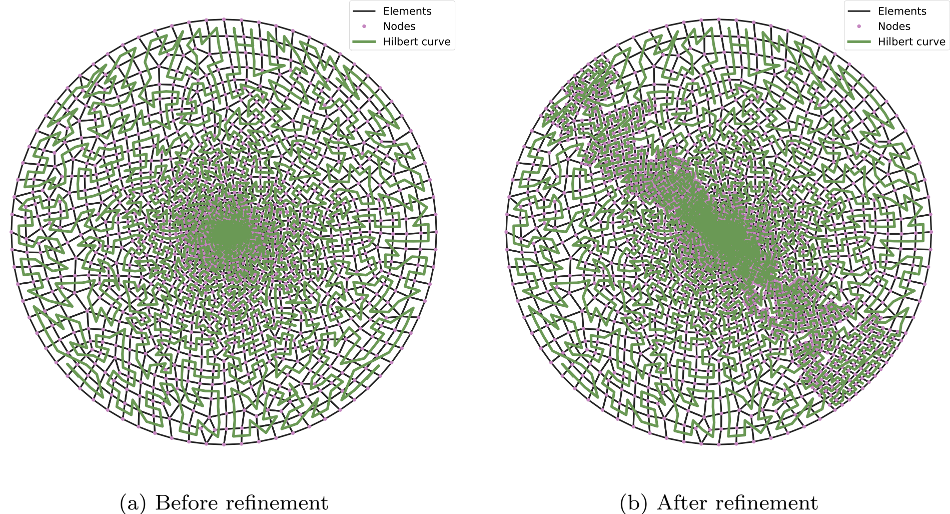


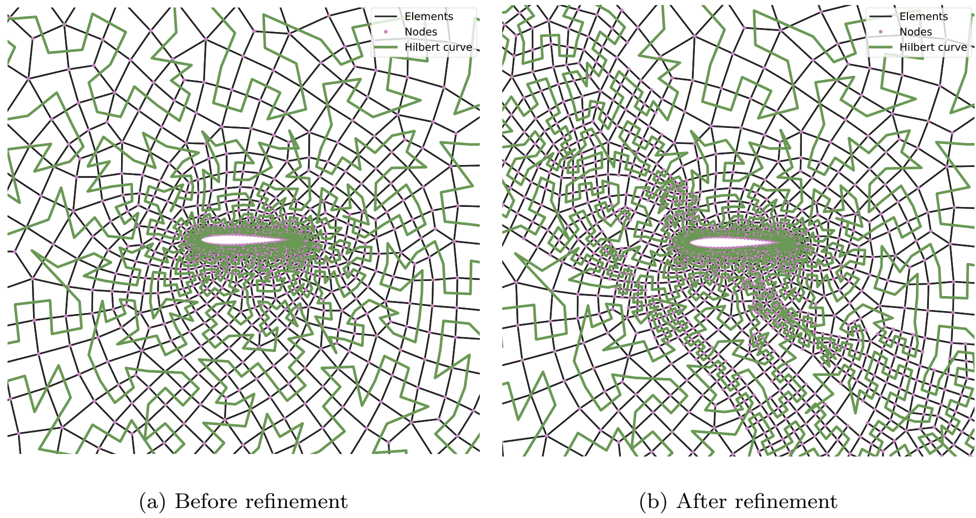
Figure A.6: Correct Hilbert curve: This is the real Hilbert curve, on a 128×128 domain where it is defined.

Next, Figures A.7 and A.8 show the pseudo-Hilbert curve generated by this ordering method in more detail. We show the mesh before and after a single refinement step to showcase how the properties of the curve are kept when refining. Figure A.7 and A.8 show the entirety of the domain and a closer view of the airfoil, respectively.



(a) Before refinement (b) After refinement

Figure A.7: Adaptive mesh refinement with an unstructured mesh: The mesh is refined, created elements follow the initial pseudo-Hilbert curve. (a) Initial mesh (b) Mesh after a single refinement step



(a) Before refinement (b) After refinement

Figure A.8: Adaptive mesh refinement with an unstructured mesh (detail): The mesh is refined, created elements follow the initial pseudo-Hilbert curve. There are some jumps along the curve around the airfoil. (a) Initial mesh (b) Mesh after a single refinement step

As the previous figures show, the refinement maintains the path of the curve very well, even if the initial mesh before refinement shows some problems. Firstly, there are some jumps along the airfoil and the edges of the domain. This is caused by the fact that the Hilbert curve is not defined for domains with holes and them. The airfoil is a hole in the domain, and the

elements at the edges of the domain are very big compared to the Hilbert curve resolution, which functionally creates a big hole on the edge of the domain where there are no elements. Luckily, the Hilbert curve naturally forms a path around the center of the domain, which is why the pseudo-Hilbert curve also goes around the airfoil instead of through it. If the airfoil was anywhere else in the domain, the result would be worse.

Secondly, the pseudo-Hilbert curve sometimes jumps over elements and sometimes takes surprising paths. This is because the underlying Hilbert curve along which the elements are sorted is very fine compared to the elements. The elements are only sorted according to where their centres fall on that curve, and sometimes the curve will miss the centre of an element and make multiple round trips before intersecting it, while encountering elements that are further along the way. This can be seen by looking closely at Figures A.4 and A.8a, where elements would be sorted very differently if their centres were moved slightly.

A potential solution to this problem would be to generate a very coarse underlying Hilbert curve, sorting the elements along that curve, and only refining the parts of the curve where multiple elements get the same index. These areas would be refined until the elements no longer share indices. This is future work, as the current incarnation works adequately to show the program functions well with complex unstructured meshes.

To conclude, this algorithm permits the use of meshes that were not numbered according to the Hilbert curve, even if the Hilbert curve is not defined for these mesh shapes. The pseudo-Hilbert curve gives good locality and short inter-block boundaries when partitioning the domain, both qualities which are crucial to obtaining good performance when using GPUs.

Appendix B

Hybrid Solver

In this work, we use GPUs to accelerate spectral methods. We use GPUs because they have a higher parallel throughput that traditional CPUs. We have shown that we can gain up to around three times the performance of CPUs on HPC systems in Section 7.3.

By that metric, by using only GPUs we leave 25% of the performance of those systems on the table. While we perform computations on GPUs, each GPU is paired with a CPU core to feed it data and instructions, while the other CPU cores are left idle. If we could design a program that uses both GPUs and CPUs for computations, we could unlock the full processing power of those HPC systems. This appendix describes a heterogeneous *hybrid solver* that uses both CPUs and GPUs to compute the solution.

In this work, we implemented a CPU version of the program to perform the CPU-GPU comparison for Section 7.3. Both versions use the same interface to communicate between processes via MPI, which means it was simple to make them work together heterogeneously. To pick the kind of worker a process is, we launch multiple processes via MPI. The first processes pick GPU devices until there are none remaining in the system and execute the GPU version of the code, and the remaining processes execute the CPU version of the code.

The work now needs to be partitioned between CPUs and GPUs. Since those two types of processors have vastly different capabilities, the partition must be uneven in order to obtain the best performance. Section 6.2 describes the algorithm to repartition a 1D workload. At the end of the section, we make the assumption that all workers have the same capacity to simplify the equation. This is no longer the case if we can have both CPU and GPU workers. We will use Equation 6.4 to separate the work between workers. It states:

$$w_{ideal,p} = W\frac{c_p}{C}, \tag{B.1}$$

where W is the total workload, C is the total capacity, p is a worker index, c_p is the capacity

of worker p, and $w_{ideal,p}$ is the ideal workload for worker p. We can easily find W by summing the number of elements in the problem, and C by summing the capacity of each worker. The capacity of each worker, representing the relative speed at which it solves problems or the number of elements it can compute in the same time as others, is the only unknown. This is the quantity that must be different for CPUs and GPUs.

For the purpose of this quick exploration of hybrid solvers, we will use a simple heuristic to determine the relative capacity of CPUs and GPUs. We run a simple case with one GPU and time its execution, then do the same with a single CPU. This gives us the performance ratio between the two types of workers. On the consumer platform described in Subsection 7.1.3, the ratio amounts to about 16. This means that a GPU solves the problem 16× faster that a single CPU core, and should solve it in the same time as 16 CPU cores. We give a capacity of 16 to GPU workers, and of one to CPU workers.

We use the same test case from Section 7.2 to examine the hybrid version of the program. We use the platform described in Subsection 7.1.3, with 32 CPU cores and one GPU. We use a mesh with 64 elements in the x and y direction, and a polynomial order N of four. Figure B.1 shows the mesh partitioning after load balancing.

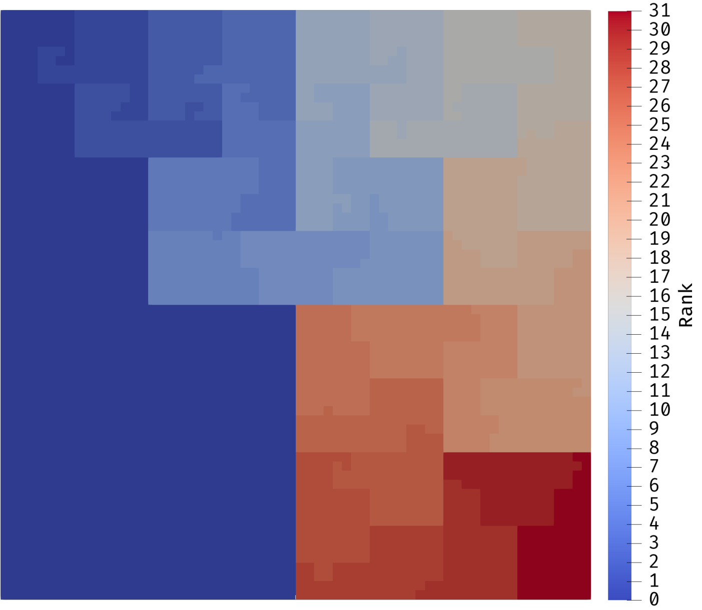


Figure B.1: Load balancing with heterogenous workers: The first worker is a GPU and has more elements in its mesh block.

Figure B.1 shows that the first worker, with a rank of zero, has much more elements in its mesh block than the other workers. The first worker is a GPU, and as such should have 16 times as many elements as the other workers. Along with 31 CPU workers with a weight of one each, the GPU worker should have about one third of the total elements of the mesh. The partitioning seen on the figure corroborates this.

In this case, the computing time for the hybrid solver was only faster than the CPU and GPU solvers when computed with a relative capacity of 4 instead of 16. This is possibly linked to the fact that the GPU is no longer optimally loaded since it only has part of the elements compared to the comparison case used to determine the capacity. More work will need to be performed in order to find the optimal capacities. The fact that the platform used for testing only has a consumer-level GPU is probably also influencing the results.

Two limitations should be addressed before this hybrid solver becomes a real solution: the meshes are initially partitioned equally, and we have to use heuristics for the capacity of different workers.

Currently, the program is made to accept as an input already-partitioned meshes when executed on multiple workers. The majority of those multi-block meshes will be partitioned such that each block has the same complexity. A mesh partitioner is provided with the program, and splits single-block meshes into multi-block meshes, with each block having the same numbers of elements. This poses no problem with a GPU-only or CPU-only solver, as all the workers will be the same. With a hybrid solver, this means that the mesh will be far from ideal until load balancing is performed, as the GPU and CPU will have the same number of elements. One solution is to perform load-balancing once before starting the problem, but this increases the computation time each time the program is launched. A second solution is to partition meshes with the worker topology in mind, and produce already load-balanced meshes. This does not work for meshes that are already in multi-block format, and makes the meshes specific to a particular topology. The meshes would have to be re-partitioned for each worker topology, or simply partitioned as part of the main program on each launch. As the overhead of load-balancing is low, the first option is probably the best.

The second issue is that we have to run cases in order to determine the relative capacities of CPUs and GPUs. The relative capacities depend on the specific GPU and CPU models, and should be computed again for each computer platform. More than that, it depends on the problem as the amount of mesh refinement and polynomial order changes the performance ratio between CPUs and GPUs. To get the best performance, we therefore would need to run each case on CPUs, then on GPUs before we can compute it on both with the computed capacities. This is tedious, and running each case three times completely negates any performance gain brought by using a hybrid solver. The solution is to either use a ballpark value for the relative capacities and have it be non-ideal in some cases, or compute it as the problem is solved. This would imply timing each worker, and using the ratio between the time it took to compute a number of iterations and the number of elements in its block as its capacity. This way, the capacity of each worker would be specific to its hardware and the current problem. It could even change as the solution progresses, and be tailored to individual CPU cores or GPUs if there is a performance from unit to unit. It would also enable the program to run on different types of computers at the same time, and make the program more robust.

All in all, a hybrid solver is a straightforward upgrade to make to increase the computational

resources available to solve problems. Some work remains to be done to make sure the work is partitioned fairly between the heterogeneous workers, but even with approximated capacities the hybrid solver shows promising results. This is definitely an interesting area of future work.