Fast Solvers for Mesh-Based Computations (Hardback, 334 pages)

Book - December 2015

1 author:

Maciej Paszynski
AGH University of Science and Technology in Kraków

Some of the authors of this publication are also working on these related projects:

- Solver performance analysis and optimization [View project]
- Adaptive and isogeometric parallel strategies for efficient accurate solution of difficult non-stationary problems [View project]
Dedicated from A to Z to my wife Anna and two daughters, Joanna and Zuzanna.
Table of Contents

List of Figures ............................................................................................................................... xi
List of Tables ................................................................................................................................... xix
Preface ............................................................................................................................................... xxi
  Historical Background .................................................................................................................. xxiii
  Structure of the Book ................................................................................................................... xxvi
  Related Works ............................................................................................................................... xxxvi
  Acknowledgment .......................................................................................................................... xxxvii

1. Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems ........................................................................................................... 1
  1.1. Derivation of the Linear System for the One-Dimensional Finite Difference Method 1
  1.2. Algebraic Algorithm of the Multi-Frontal Solver ........................................................ 5
  1.3. Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm ............................................................. 11
  1.4. One-Dimensional Finite Element Method with Linear Basis Functions .............. 26
  1.5. One-Dimensional Isogeometric Collocation Method with Quadratic B-Splines ....... 32
  1.6. One-Dimensional Finite Element Method with Bubble Basis Functions ............. 37

2. One-Dimensional Non-Stationary Problems ......................................................................... 49
  2.1. Euler Scheme with Respect to Time Mixed with Finite Element Method with Linear Basis Functions with Respect to Space ................................................................. 49
  2.2. \( \alpha \)-Scheme with Respect to Time Mixed with Method with Linear Basis Functions for Space ................................................................. 60

3. Multi-Frontal Direct Solver Algorithm for Multi-Diagonal One-Dimensional Problems 67
  3.1. One-Dimensional Collocation Method with Higher-Order B-Splines .................... 67
  3.2. One-Dimensional Isogeometric Finite Element Method ............................................ 69

4. Multi-Frontal Direct Solver Algorithm for Two-Dimensional Grids with Block Diagonal Structure of the Matrix ......................................................................................... 81
TABLE OF CONTENTS

4.1. Two-Dimensional Projection Problem with Linear Basis Functions .................................. 81
4.2. Two-Dimensional Mesh with Anisotropic Edge Singularity ............................................. 85
4.3. Two-Dimensional Mesh with Point Singularity ............................................................... 94

5. Multi-Frontal Direct Solver Algorithm for Three-Dimensional Grids with Block Diagonal Structure of the Matrix........................................................................................................ 101
5.1. Three-Dimensional Projection Problem with Linear Basis Functions ............................ 101
5.2. Three-Dimensional Mesh with Anisotropic Face Singularity ......................................... 105
5.3. Three-Dimensional Mesh with Anisotropic Edge Singularity ...................................... 115
5.4. Three-Dimensional Mesh with Point Singularity ......................................................... 122

6. Multi-Frontal Direct Solver Algorithm for Two-Dimensional Isogeometric Finite Element Method ...................................................................................................................... 125
6.1. Isogeometric Finite Element Method for Two-Dimensional Problems ........................... 125
6.2. Graph-Grammar for Generation of the Elimination Tree .............................................. 127
6.3. Graph-Grammar Productions for the Solver Algorithm .............................................. 137

7. Expressing Partial LU Factorization by BLAS Calls .............................................................. 151
7.1. LU Factorization of A(1,1) ............................................................................................. 152
7.2. Multiplication of A(1,2) by the Inverse of A(1,1) .......................................................... 152
7.3. Multiplication of b(1) by the Inverse of A(1,1) ............................................................. 153
7.4. Matrix Multiplication and Subtraction A(2,2)=A(2,2)–A(2,1)A(1,2) .......................... 155
7.5. Matrix Vector Multiplication and Subtraction b(2)=b(2)–A(2,1)b(1) ......................... 155
7.6. Example ........................................................................................................................ 156

8. Multi-Frontal Solver Algorithm for Arbitrary Mesh-Based Computations ........................ 159
8.1. Multi-Frontal Solver Algorithm for Arbitrary Grids ...................................................... 159
8.2. Hypermatrix Module ..................................................................................................... 166
8.3. Elimination Tree Module .............................................................................................. 168
8.4. Supernodes System Module ....................................................................................... 171
8.5. Interface ........................................................................................................................ 176
8.6. Structure of Matrices for Different Two-Dimensional Methods .................................. 195
8.6.1. Two-Dimensional Finite Difference and Finite Element Method with Linear Basis Functions .................................................................................................................. 195
8.6.2. Two-Dimensional Finite Element Method with Bubble Basis Functions .......... 195
8.6.3. Two-Dimensional Isogeometric Collocation and Finite Element Method .... 197

9. Elimination Trees .................................................................................................................. 199
9.1. Elimination Trees and Multi-Frontal Solvers ................................................................. 199
TABLE OF CONTENTS

9.2. Quasi-Optimal Elimination Tree for Two-Dimensional Mesh with Point Singularity. 205
9.3. Quasi-Optimal Elimination Tree for Two-Dimensional Mesh with Edge Singularity. 210
9.4. Nested-Dissection Elimination Tree for Two-Dimensional Mesh with Edge Singularity. 217
9.5. Minimum Degree Tree for Two-Dimensional Mesh with Edge Singularity. 222
9.6. Estimation of the Number of Floating Point Operations and Memory Usage. 229
9.7. Elimination Trees for Three-Dimensional Grids. 231
9.7.1. Elimination Tree for Three-Dimensional Mesh with Point Singularity. 231
9.7.2. Elimination Tree for Three-Dimensional Mesh with Anisotropic Edge Singularity. 233
9.7.3. Elimination Tree for Three-Dimensional Mesh with Anisotropic Face Singularity. 233
9.7.4. Elimination Tree for Three-Dimensional Mesh with Edge Singularity. 233
9.7.5. Elimination Tree for Three-Dimensional Mesh with Face Singularity. 234
9.7.6. Elimination Tree for Three-Dimensional Uniform Mesh. 236

10. Reutilization and Reuse of Partial LU Factorizations. 239
10.1. Idea of the Reutilization Algorithm. 239
10.2. Example Implementation of the Reutilization Algorithm. 243
10.3. Idea of the Reuse Algorithm. 252
10.4. Example Implementation of the Reuse Algorithm. 258

11. Numerical Experiments. 269
11.1. Measuring the Solver Performance by Means of Execution Time. 269
11.2. Measuring the Solver Performance by Means of the Number of Floating Point Operations (FLOPs). 271
11.3. Measuring the Solver Performance by Means of Efficiency and Speedup. 272
11.4. Graph-Grammar-Based Multi-Thread GALOIS Solver for Two-Dimensional Grids with Singularities. 273
11.4.1. Comparison of Execution Times. 274
11.4.2. Comparison of FLOPs for Different Elimination Trees. 275
11.4.3. Comparison of Efficiency and Speedup. 277
11.5. Graph-Grammar-Based Multi-Thread GALOIS Solver for Three-Dimensional Grids with Singularities. 280
11.6. Graph-Grammar-Based One-Dimensional Isogeometric Finite Element Method GPU Solver. 283
11.6.1. Comparison of Execution Time. 283
### TABLE OF CONTENTS

11.6.2. Comparison of Speedup of Parallel Solver ....................................................... 286

11.7. Graph-Grammar-Based Two-Dimensional Isogeometric Finite Element Method
    GPU Solver .............................................................................................................. 286
    11.7.1. Comparison of Execution Time ....................................................................... 287
    11.7.2. Comparison on the Number of FLOPs ............................................................. 289

11.8. Graph-Grammar-Based Solver for Two-Dimensional Adaptive Finite Element Method
    ........................................................................................................................................ 289
    11.8.1. The Radical Mesh with Two Finite Elements .................................................. 290
    11.8.2. L-shape Domain Problem .............................................................................. 290
    11.8.3. The Radical Mesh with Two Point Singularities ............................................ 294

11.9. Graph-Grammar-Based Solver for Three-Dimensional Adaptive Finite Element Method
    ........................................................................................................................................ 299
    11.9.1. Fichera Model Problem ................................................................................. 299

Bibliography .................................................................................................................. 303
Index ............................................................................................................................... 311
List of Figures

P.1  Dependency plan for the parts of the book describing graph-grammar-based solvers for one-dimensional mesh-based computations. ........................................ xxviii

P.2  Dependency plan for the parts of the book describing graph-grammar-based solvers for two- and three-dimensional mesh-based computations resulting in a similar structure of the solver such as one-dimensional methods. ........ xxix

P.3  Dependency plan for the parts of the book defining the solver for arbitrary mesh-based computations. ................................................................. xxx

P.4  Dependency plan for the parts of the book describing elimination trees for different grids. ................................................................. xxxii

P.5  Dependency plan for the parts of the book describing numerical results. ........ xxxiii

P.6  Dependency plan for the parts of the book describing numerical results. ........ xxxiv

1.1  Structure of the matrix for one-dimensional finite difference method. ............ 5

1.2  Construction of the exemplary elimination tree by execution of productions \((P_1) - (P_2)_1 - (P_2)_2 - (P_2)_3 - (P_2)_4 - (P_3)_1 - (P_3)_2 - (P_3)_3 - (P_3)_4 - (P_3)_5 - (P_3)_6\). ................................................................. 14

1.3  **Left panel:** Dependency graph between tasks. **Right panel:** Shading of the dependency graph. ................................................................. 14

1.4  Execution of the graph-grammar productions \((A_1) - (A)_1 - (A)_2 - (A)_3 - (A)_4 - (A)_5 - (A)_6 - (A)_7 - (A)_8 - (E_2)_1 - (E_2)_2 - (E_2)_3 - (A_{root}) - (E_{root})\) representing the multi-frontal solver algorithm running over the exemplary elimination tree. ................................................................. 22

1.5  **Top panel:** Dependency graph between tasks. **Bottom panel:** Shading of the dependency graph. ................................................................. 23

1.6  Structure of the graph-grammar productions for one-dimensional finite difference and finite element method with linear basis functions and the collocation method with quadratic basis functions. ................................................................. 31

1.7  Structure of the matrix for one-dimensional finite element method with bubble functions. ................................................................. 40
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Exemplary plot with the solutions from particular time steps.</td>
<td>60</td>
</tr>
<tr>
<td>3.1</td>
<td>Structure of the matrix for the one-dimensional isogeometric finite element method with quadratic basis functions.</td>
<td>72</td>
</tr>
<tr>
<td>3.2</td>
<td>Structure of the graph-grammar solver for the one-dimensional isogeometric finite element method with quadratic basis functions.</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Binary elimination tree for 2D mesh with edge singularity. Elimination of interiors of layers.</td>
<td>86</td>
</tr>
<tr>
<td>4.2</td>
<td>Binary elimination tree for 2D mesh with edge singularity. Merge and elimination of interfaces of layers.</td>
<td>87</td>
</tr>
<tr>
<td>4.3</td>
<td>Binary elimination tree for 2D mesh with edge singularity. Merging and elimination of top problem.</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>Binary elimination tree for 2D mesh with point singularity. Elimination of interiors of layers.</td>
<td>95</td>
</tr>
<tr>
<td>4.5</td>
<td>Binary elimination tree for 2D mesh with point singularity. Merge and elimination of interfaces of layers.</td>
<td>96</td>
</tr>
<tr>
<td>4.6</td>
<td>Binary elimination tree for 2D mesh with point singularity. Merge and elimination of the top problem.</td>
<td>97</td>
</tr>
<tr>
<td>4.7</td>
<td>Constrained approximation with linear basis functions.</td>
<td>98</td>
</tr>
<tr>
<td>5.1</td>
<td>Binary elimination tree for 3D mesh with anisotropic face singularity. Elimination of interiors of layers.</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>Binary elimination tree for 3D mesh with anisotropic face singularity. Merge and elimination of interfaces of layers.</td>
<td>109</td>
</tr>
<tr>
<td>5.3</td>
<td>Binary elimination tree for 3D mesh with anisotropic face singularity. Merge and elimination of top problem.</td>
<td>110</td>
</tr>
<tr>
<td>5.4</td>
<td>Binary elimination tree for 3D mesh with anisotropic edge singularity. Elimination of interiors of layers.</td>
<td>119</td>
</tr>
<tr>
<td>5.5</td>
<td>Binary elimination tree for 3D mesh with anisotropic edge singularity. Merge and elimination of interfaces of layers.</td>
<td>120</td>
</tr>
<tr>
<td>5.6</td>
<td>Binary elimination tree for 3D mesh with anisotropic edge singularity. Merge and elimination of top problem.</td>
<td>121</td>
</tr>
<tr>
<td>5.7</td>
<td>Binary elimination tree for 3D mesh with point singularity. Elimination of interiors of layers.</td>
<td>122</td>
</tr>
<tr>
<td>5.8</td>
<td>Binary elimination tree for 3D mesh with point singularity. Merge and elimination of interfaces of layers.</td>
<td>123</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

5.9 Binary elimination tree for 3D mesh with point singularity. Merge and elimination of top problem. .......................................................... 124

6.1 2D mesh for the B-spline-based finite element method. ................................. 126
6.2 Graph-grammar productions for generation of the elimination tree. .................. 127
6.3 Graph-grammar productions for generation of the elimination tree. .................. 128
6.4 Graph-grammar productions for generation of the elimination tree. .................. 129
6.5 Graph-grammar productions for generation of the elimination tree. .................. 130
6.6 Graph-grammar productions for generation of the elimination tree. .................. 131
6.7 Graph-grammar productions for generation of the elimination tree. .................. 132
6.8 Merging of four-element matrices into a parent level system. ......................... 139
6.9 Graph-grammar production for merging of four matrices. ............................. 140
6.10 Graph-grammar productions for generation of the element frontal matrices. ....... 149
6.11 Frontal matrices generated for all the leaves of the tree. .............................. 150

8.1 Left panel: A single two-dimensional finite element with polynomial orders of approximation over edges equal to \( p_1, p_2, p_3, p_4 \) and polynomial order of approximation over the interior equal to \( p_h \) in the horizontal direction and \( p_v \) in the vertical direction. Right panel: Sizes of blocks in element local matrix: 1 unknown per element vertex, \( p_i-1 \) unknowns per edge, and \( (p_h-1)(p_v-1) \) unknowns for the element interior. .......................................................... 161
8.2 Management of nodes associated with leaves of the elimination tree. ............... 161
8.3 Sending of the Schur complements on the level of leaves. ............................ 162
8.4 Management of nodes from the second level of the elimination tree. ............... 163
8.5 Sending of the Schur complements on the second level of the elimination tree. ... 164
8.6 Management of nodes associated with the top of the elimination tree. ............. 165
8.7 Element frontal hypermatrices on our four finite elements case. .................... 177
8.8 Mapping of supernodes into elements on our four finite elements case. .......... 177
8.9 Elimination tree on our four finite elements case. .................................... 178
8.10 First step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements. ..................................... 181
8.11 Second step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements. ............................. 181
8.12 Third step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements. ............................. 182
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.13</td>
<td>Fourth step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements.</td>
</tr>
<tr>
<td>8.14</td>
<td>Fifth step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements.</td>
</tr>
<tr>
<td>8.15</td>
<td>Sixth step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements.</td>
</tr>
<tr>
<td>8.16</td>
<td>Seventh step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements.</td>
</tr>
<tr>
<td>8.17</td>
<td>Eighth step of the multi-frontal solver algorithm executed on a two-dimensional grid with rectangular elements.</td>
</tr>
<tr>
<td>8.18</td>
<td>Construction of the system at the root node.</td>
</tr>
<tr>
<td>8.19</td>
<td>Solution of the system at the root node.</td>
</tr>
<tr>
<td>8.20</td>
<td>Backward substitutions performed.</td>
</tr>
<tr>
<td>8.21</td>
<td>First step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.22</td>
<td>Second step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.23</td>
<td>Third step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.24</td>
<td>Fourth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.25</td>
<td>Fifth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.26</td>
<td>Sixth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.27</td>
<td>Seventh step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.28</td>
<td>Eighth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.29</td>
<td>Ninth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.30</td>
<td>Tenth step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.31</td>
<td>Last step of the parallel multi-frontal solver algorithm.</td>
</tr>
<tr>
<td>8.32</td>
<td>Example of two finite difference or finite element mesh with vertices for linear basis functions.</td>
</tr>
<tr>
<td>8.33</td>
<td>Structure of matrix for two-dimensional finite difference and finite element method with linear basis functions.</td>
</tr>
<tr>
<td>8.34</td>
<td>Example of two finite element mesh with quadratic basis functions spread over vertices, edges and interiors.</td>
</tr>
<tr>
<td>8.35</td>
<td>Structure of matrix for two-dimensional finite element method with bubble basis functions.</td>
</tr>
<tr>
<td>8.36</td>
<td>Structure of matrix for two-dimensional isogeometric finite element method with quadratic B-spline basis functions.</td>
</tr>
<tr>
<td>9.1</td>
<td>Elimination tree for two finite element mesh.</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

9.2 The DAG for the example of the elimination tree for the two finite element mesh. 200
9.3 Hypergraph representation of the mesh with two elements. ....................... 201
9.4 Execution of graph-grammar productions \((P_{genM1\_1}), (P_{elimM1\_1}), (P_{genM1\_2}), (P_{elimM1\_2}), (P_{mergeM1\_1,2}), (P_{solveM1\_1,2})\) representing the execution of the multi-frontal solver algorithm. .................... 201
9.5 Shading of the dependency graph. ................................................. 202
9.6 Visual explanation of \(a\) and \(b\). .................................................... 203
9.7 Point singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part I ................................................................. 206
9.8 Point singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part II ................................................................. 207
9.9 Dependency graph for graph-grammar productions expressing the multi-frontal solver algorithm over two-dimensional mesh with point singularity. ........ 208
9.10 Optimal elimination tree for point singularity for sequential solver. ............ 209
9.11 Optimal elimination tree for edge singularity ....................................... 210
9.12 Edge singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part I ................................................................. 211
9.13 Edge singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part II ................................................................. 212
9.15 Edge singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part IV ............................................................... 214
9.16 Edge singularity, volume and neighbors. Execution of graph-grammar produc- tions. Part V ............................................................... 215
9.17 Shaded dependency graph for an edge singularity for left-hand side of the mesh. 216
9.18 Edge singularity, nested-dissections. Execution of graph-grammar productions. Part I ................................................................. 217
9.19 Edge singularity, nested-dissections. Execution of graph-grammar productions. Part II ................................................................. 218
9.20 Edge singularity, nested-dissections. Execution of graph-grammar productions. Part III ............................................................... 219
9.21 Edge singularity, nested-dissections. Execution of graph-grammar productions. Part IV ............................................................... 220
LIST OF FIGURES

9.23  Edge singularity, nested-dissections. Execution of graph-grammar productions.
      Part V ................................................................. 221
9.24  Edge singularity, AMD. Execution of graph-grammar productions. Part I. .... 222
9.25  Edge singularity, AMD. Execution of graph-grammar productions. Part II. ... 223
9.26  Edge singularity, AMD. Execution of graph-grammar productions. Part III. . 224
9.27  Edge singularity, AMD. Execution of graph-grammar productions. Part IV. ... 225
9.28  Edge singularity, AMD. Execution of graph-grammar productions. Part V. ... 226
9.29  Edge singularity, AMD. Execution of graph-grammar productions. Part VI. ... 227
9.30  Edge singularity, AMD. Execution of graph-grammar productions. Part VII. .. 228
9.31  Estimations of the computational complexity for 3D grid with point singularity. 232
9.32  Estimations of the computational cost for 3D grid with anisotropic edge
      singularity .............................................................. 233
9.33  Estimations of the computational cost for 3D grid with face singularity. ....... 234
9.34  Estimations of the computational cost for a 3D grid with edge singularity. .... 235
9.35  Estimations of the computational complexity for a 3D grid with face singularity. 235
9.36  Estimations of the computational cost for a 3D uniform grid. ..................... 236

10.1  2D mesh with anisotropic edge singularity with elimination tree layered toward
      the singularity ........................................................ 240
10.2  Standard multi-frontal solver computations for the case of the three elements. . 241
10.3  Removing part of the mesh and part of the tree before reutilization. ............ 242
10.4  Reutilization of partial LU factorizations over unrefined parts of the mesh. .. 243
10.5  2D uniform mesh with regular elimination tree. ............................... 254
10.6  Standard multi-frontal solver computations. ..................................... 255
10.7  Identification of identical sub-systems. ......................................... 256
10.8  Reuse of partial LU factorizations over identical parts of the mesh. ............ 257

11.1  Log-log scale comparison of the execution times of the MUMPS and GALOIS
      solvers for different numbers of threads, for different numbers of refinement
      levels, and for the mesh with edge singularity. .......................... 274
11.2  Log-log scale comparison of the execution times of the MUMPS and GALOIS
      solvers for different numbers of threads, for different numbers of refinement
      levels, and for the mesh with point singularity. ........................ 275
11.3  Log-log scale comparison of the efficiency of the MUMPS and GALOIS solvers
      for different numbers of threads, for different numbers of refinement levels, for
      the mesh with edge singularity. ................................. 277
LIST OF FIGURES

11.4 Log-log scale comparison of the speedup of the MUMPS and GALOIS solvers for different numbers of threads, for different numbers of refinement levels, for the mesh with edge singularity. ......................... 278
11.5 Balancing of our optimal elimination tree. ............................ 278
11.6 Balancing of nested-dissections elimination tree. ....................... 279
11.7 3D mesh with point singularities. .................................... 281
11.8 Execution time of GALOIS solver for 3D mesh with point singularity with uniform $p = 2$ for different number of threads. Comparison with MUMPS solver. . 282
11.9 Execution time of GALOIS solver for 3D mesh with point singularity with uniform $p = 3$ for different number of threads. Comparison with MUMPS solver. . 282
11.10 1D solver using linear B-splines. Comparison of the GPU cost vs. the CPU one. . 284
11.11 1D solver using quadratic B-splines. Comparison of the GPU cost vs. the CPU one. 284
11.12 1D solver using cubic B-splines. Comparison of the GPU cost vs. the CPU one. . 285
11.13 1D solver using quartic B-splines. Comparison of the GPU cost vs. the CPU one. 285
11.14 Speedup of the 1D GPU solver for 2048 elements, for linear, quadratic, cubic, quintic and quartic B-splines. ................. 286
11.15 Comparison of execution time of GPU solver versus MUMPS sequential CPU solver for linear B-splines. ......................... 287
11.16 Comparison of execution time of GPU solver versus MUMPS sequential CPU solver for quadratic B-splines. ...................... 288
11.17 Comparison of execution time of GPU solver versus MUMPS sequential CPU solver for cubic B-splines. .......................... 288
11.18 Measurements of GFLOPS on NVIDIA GTX 780 for linear, quadratic and cubic B-splines. ......................................... 289
11.19 The two-dimensional radical mesh. ................................. 290
11.20 Comparison of the number of non-zero entries for the radical mesh for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation $p = 5$. .......................... 291
11.21 Comparison of the execution time the radical mesh for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation $p = 5$. . 292
11.22 L-shape domain. ...................................................... 292
11.23 Euclidean norm of the gradient of the solution to the L-shape domain problem goes to infinity. ......................... 293
xviii LIST OF FIGURES

11.24 Comparison of the number of non-zero entries for the L-shape domain problem for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation \( p = 5 \). ......................................................... 294

11.25 Comparison of the execution time the L-shape problem for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation \( p = 5 \). 295

11.26 Radical mesh with two singularities. ................................................................. 296

11.27 Comparison of the number of non-zero entries for the radical mesh with two singularities for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation \( p = 5 \). ......................................................... 297

11.28 Comparison of the execution time for the radical mesh with two singularities for the solver without reutilization (Hypersolver), the solver with reutilization (Reutilization), and the state-of-the-art MUMPS solver for the polynomial order of approximation \( p = 5 \). ......................................................... 298

11.29 The Fichera model problem. ............................................................................. 300

11.30 Comparison of the number of non-zero entries for the Fichera problem for the solver without reutilization (Hypersolver), and the state-of-the-art MUMPS solver with METIS, and PORD orderings, for the polynomial order of approximation \( p = 5 \). ......................................................... 300

11.31 Comparison of the execution time for the Fichera problem for the solver with reutilization (Reutilization=FE+B.S.), and the state-of-the-art MUMPS solver with METIS and PORD orderings, for the polynomial of approximation \( p = 5 \). 301
List of Tables

9.1 Rough estimation of the number of memory transfers for different operations. 204
9.2 Estimation of computational cost of a sample two-element domain for graph-grammar productions expressing the multi-frontal solver algorithm for $p = 2$. 204
9.3 Estimation of the number of operations of LU factorization for nested-dissections and area and neighbors (optimal) ordering algorithms for edge singularity. 229
9.4 Estimation of the number of memory transfers for nested-dissections and area and neighbors (optimal) ordering algorithms for edge singularity. 229
9.5 Estimation of the execution time defined as the number of floating-point operations plus number of memory transfers*100, for nested-dissections and area and neighbors (optimal) ordering algorithms for edge singularity. 230
11.1 Comparison of FLOPs for our trees vs. MUMPS with nested-dissections (METIS), approximate minimum fill (AMF), approximate minimum degree (AMD), quasi-approximate minimum degree (QAMD) and PORD, and SCOTCH, executed over the mesh with point singularity. 276
11.2 Comparison of FLOPs for area and neighbors algorithm vs. MUMPS with nested-dissections (METIS), approximate minimum fill (AMF), approximate minimum degree (AMD), quasi-approximate minimum degree (QAMD), PORD, and SCOTCH, executed over the mesh with edge singularity. 276
11.3 Comparison of the number of iterations of ILUPCG solver for three-dimensional mesh with quadratic polynomials and $4 \times 4 \times 4$ singularities, for hybrid and standard algorithm. 283
Preface

This book is based on more than 10 years of my personal experience with – first designing and implementing parallel direct solvers for different mesh-based computations, and – later, teaching the subject to computer science students.

Direct solvers are the core part of many engineering analyses performed using different mesh-based methods, such as the finite difference method, the collocation method, the finite element method, and the isogeometric finite element or collocation methods. Let us focus on the representative Finite Element Method (FEM) [52]. The finite element solution process starts with the generation of a mesh describing the geometry of the computational problem. Next, the physical phenomena governing the problem are described using partial differential equations (PDEs). In addition to this differential system, boundary and initial conditions may need to be specified to guarantee the uniqueness of the solution. Then, the PDE system is discretized into a system of linear equations by using the finite element method. The resulting algebraic system is inverted using a solver algorithm.

Existing direct solvers of linear equations (for example, MUMPS [63], SuperLU [96], PARDISO [73], and HSL [50]) are based on solving a linear system given by a global matrix and one or several right-hand sides. The global matrix is provided either as a list of non-zero entries, or it is obtained from merging a sequence of element frontal matrices. In both cases, the additional available knowledge about the structure of the computational mesh is lost. This book presents an alternative way of constructing multi-frontal direct solver algorithms for mesh-based computations. The construction of the solver algorithm is based on the additional available knowledge concerning the structure of the computational mesh. The alternative method presented in this book allows us to outperform traditional direct solver algorithms. The mesh-based solvers have been applied for efficient simulations of several engineering problems. Relevant application examples are the simulations of the propagation of electromagnetic or acoustic waves over a human head model [30, 40, 80], earth monitoring simulations such as resistivity logging measurements [74, 75, 79, 81], or isogeometric finite element method simulations [57, 58, 94].
The construction of the direct solver algorithm based on the structure of computational mesh allows for better decomposition of the computational problem into sets of independent tasks. This in turn allows us to obtain a solver algorithm that delivers more efficient parallel implementation [78, 79, 80, 81]. Additionally it allows us to implement some special tricks such as the reuse of computations for identical sub-parts of the mesh [85], and the reutilization of LU factorizations over unrefined parts of the mesh [77]. These techniques are not easily available for classic direct solvers. This book describes how to design and implement such mesh-based direct solver algorithms.

The following key features summarize the contents of the book:

- The book targets graduate and PhD students, as well as researchers who would like to learn how to design and implement parallel direct solvers for mesh-based computations.

- The structure of the book follows the structures of the matrices, starting from tri-diagonal matrices resulting from one-dimensional mesh-based methods, through multi-diagonal or block-diagonal matrices and ending with general sparse matrices.

- Each chapter of the book discusses how to design and implement a parallel sparse direct solver specific for a particular structure of the matrix. All the solvers presented in the book are either designed from scratch or based on the solvers already designed and implemented in the previous chapters.

- Each chapter derives the complete Java or Fortran code of the parallel sparse direct solver that can be used during labs with students, or it presents modifications of the source code developed in the previous chapters.

- The selection of Java language is motivated by its simplicity. We are aware that Java may not provide the optimal performance of matrix computations, and alternative C or Fortran implementations may outperform the Java solver. However, the exemplary Java codes can be used as a reference for designing and implementing parallel direct solvers in more efficient languages. In particular, in the Numerical Experiments chapter we show three efficient implementations, one implemented in C with NVIDIA CUDA, targeting GPUs, the second one implemented in C with GALOIS, targeting a multi-core Linux node, and the third one implemented in Fortran, targeting Linux cluster nodes.

- We also derive exemplary element frontal matrices for two- or three-dimensional mesh-based computations. These matrices can be used as references for testing the developed parallel direct solvers.

I am first going to describe the history of my research regarding direct solvers, then I will relate my research to other works on direct solvers, and finally I will explain the structure of the book.
Historical Background

I started my work on the direct solver and related topics in 2005, when I joined Leszek Demkowicz as a postdoc at the Institute for Computational and Engineering Science (ICES), The University of Texas at Austin. My first job was to parallelize hp adaptive finite element method code. Part of the job was to incorporate the parallel version of the frontal solver with the adaptive code, originally developed in 2000 by Timothy Walsh [93]. The parallel frontal solver was obtained from parallelization of the sequential frontal solver, originally developed in 1970 by Bruce Irons [53]. I learned that the sequential version of the frontal solver worked over the computational mesh partitioned into many finite elements. The frontal solver browsed finite elements, one by one, generated element frontal matrices in the prescribed order, and performed some partial eliminations of unknowns related to element nodes. I also learned that the input for these solvers was an order of element, and that each finite element submits its frontal matrix to these solvers. Different orders of elements resulted in different computational costs of the frontal solver algorithm. Thus the performance of frontal solvers depends on the structure of the computational mesh and on the order of the elements submitted to these solvers. Indeed in 2010 I realized that two- or three-dimensional grids h-refined toward point singularities result in the linear computational cost of the frontal solver algorithm. This was the topic of the doctoral dissertation of my first PhD student, Piotr Gurgul [44], defended in 2014.

The parallel version of the frontal solver, developed by Timothy Walsh [93], was used on the computational mesh decomposed into many sub-domains. The parallel frontal solver used multiple frontal solvers working over the particular sub-domains in parallel. The resulting interface sub-matrices, called the Schur complements, were collected and merged together and solved on a single processor. Parallelization of the frontal solver proposed by Timothy Walsh is based on the domain decomposition paradigm. Distribution of data concerns the computational mesh. It is a different approach than in the classical parallel direct solver, where the partition of data concerns the global system of linear equations, not necessarily the computational mesh.

My personal practical experience with direct solvers reached another level of understanding when I started using sequential and parallel versions of the MUMPS solver [5, 6, 7, 63], with other postdocs of Prof. Leszek Demkowicz, including Jason Kurtz and David Pardo. We interfaced sequential and parallel MUMPS solvers to sequential and parallel hp adaptive finite element codes. I learned the idea of multiple frontal matrices processed at the same time, and realized that the MUMPS solver is not really interested in the topological structure of the computational mesh, but rather in the non-zero pattern of the global matrix. In 2006, working as a postdoc for Carlos Torres-Verdin, from the Department of Petroleum and Geosystems Engineering, University of Texas at Austin, I designed and implemented a parallel solver, generalizing the idea of Timothy Walsh in a multi-level recursive way [79]. The frontal solvers executed over sub-domains result in the Schur complement matrices associated with the interfaces between sub-domains. These sub-domains are
joined into pairs, the Schur complements are also merged together, and the fully assembled unknowns related to the common interfaces between sub-domains are eliminated. This process is recursively repeated until all sub-domains are joined into one big sub-domain, and all the Schur complement matrices are joined into one matrix associated with a single cross-section of the domain. I implemented such a solver algorithm for two-dimensional grids with the help of David Pardo and Victor Calo, postdocs of Carlos Torres-Verdin, and Tom Hughes, also from ICES. Our solver, using the additional knowledge of the topological structure of the mesh, outperformed the state-of-the-art MUMPS solver working on the level of the global matrix. The MUMPS solver didn’t have such additional knowledge. Later, in 2010, I repeated this experiment for three-dimensional problems, outperforming the MUMPS solver again [80].

Since that time, I have been trying to answer questions such as

- What are the benefits that a solver can gain from additional knowledge of the topological structure of the computational mesh from where the global system of linear equations originated?
- Is it possible to reconstruct the topological structure of the mesh from the non-zero pattern of the global matrix?
- Are the classical multi-frontal solvers missing something when they work on the global matrix ignoring the structure of the mesh?

In the following years, I tried to answer these questions by designing and implementing multi-frontal solvers based on the topological structure of the computational mesh and comparing them to classical multi-frontal solvers, working with the global matrix without the additional knowledge of the structure of the mesh.

Classical sequential multi-frontal solvers use the ordering of unknowns in the global matrix to guide the elimination process. They introduce elimination trees to guide the elimination process. First, I realized that the ordering of unknowns in sequential multi-frontal solvers is obtained from browsing the elimination tree in a breadth-first search (BFS) way. In this sense, from the point of view of sequential multi-frontal solvers, the elimination trees are equivalent to the so-called ordering algorithms. Second, I have found out that the ordering of unknowns is basically prescribing the order of elimination of rows in the global matrix. But the ordering does not provide the information about what rows we should subtract from. Classical solvers obtain this information by looking at the global structure of the matrix. However, when we work with the topological structure of the mesh, with elements, element frontal matrices, and basis functions related to the nodes of the mesh, spread over one or several finite elements, this information can be easily found by looking at the topology of the mesh and the adjacency information between the elements, supports of the basis functions, and distribution of the nodes within elements.

This led me to the idea of a hypermatrix, originally proposed by Demmel [31], where the hypermatrix entries are connected with sub-matrices related to particular supernodes of the mesh. I
Preface

designed and implemented the multi-frontal solver with the hypermatirx idea in years 2008–2010, working again with Leszek Demkowicz on the project aimed at multi-physical simulations of the acoustics of the human head [30], over very complicated non-uniform mesh with non-uniform distribution of high polynomial orders of approximation. This project also involved collaboration with David Pardo, who moved to Spain and became a research professor at the Basque Center for Applied Mathematics at the University of the Basque Country in Bilbao [80].

At the same time, I started teaching subjects such as Theory of Concurrency, Finite Difference and Finite Element Method and Adaptive Algorithms at AGH University, where inspired by Prof. Robert Schaefer and my wife, Dr. Anna Paszyńska from Jagiellonian University, I came up with the idea of using graph-grammar systems and trace-theory-based schedulers for expressing multi-frontal solver algorithms as sets of basic undividable independent tasks that can be executed in parallel, set by set. I first proposed such a graph-grammar-based parallel solver for the finite difference method in 1D and 2D [66]. I developed these ideas further with my students when I was teaching courses Simulations of Continuous Processes and Adaptive Algorithms at AGH University and later Graph-Grammars and Algorithmic Transformations at King Abdullah University of Science and Technology, where I was invited by Victor Calo, who moved there from ICES.

After a successful presentation at the International Conference on Computational Science (ICCS) in 2010 concerning the trace-theory-based graph-grammar parallel solver and publishing an invited paper about the solver in the first issue of the Journal of Computational Science [80], Victor Calo from King Abdullah University of Science and Technology invited me to work with him in developing fast solvers for the isogeometric finite element method and related applications. Victor Calo introduced me to the isogeometric finite element method and opened my eyes to the fact that the performance of multi-frontal direct solvers is related to extended supports of the B-spline basis functions [23, 95] used over computational meshes in the isogeometric finite element method.

In the meantime, working again with Victor Calo and David Pardo, I came up with the idea of reutilization of the LU factorization over unrefined parts of the mesh. Reutilization allows reducing by one order of magnitude the computational cost of the multi-frontal solver algorithm when working with computational grids $h$-refined toward singularities. This idea can be implemented in a solver provided that it has additional information about the topological structure of the mesh. At the same time, after a visit of Luis Garcia-Castillo from University of Madrid, with another PhD student, Marcin Sieniek, we came up with the idea to reuse branches of the elimination tree over identical parts of the mesh. This concept of reuse also cannot be implemented without the knowledge of the solver about the structure of the computational mesh. Working with grids with singularities also inspired the dissertation of my second PhD student, Arkadiusz Szymczak, who has removed the deadlock problem occurring during anisotropic mesh adaptations [89, 90].

With the help of two PhD students, Krzysztof Kuźnik (who later left to work in industry) and Maciej Woźniak, in 2012–2014, we came up with a graph-grammar-based implementation of the multi-frontal solver for the isogeometric finite element method in 1D and 2D, for GPU [58, 94].
Later, with a new PhD student, Paweł Lipski, we realized that very similar implementation of the graph-grammar-based multi-frontal direct solvers can be used for the isogeometric collocation methods.

A graph-grammar solver has also been implemented in the GALOIS system [39, 78] by three of my PhD students, Konrad Jopek, Maciej Woźniak, and Damian Goik (who later left my team). This happened after listening to my presentation at ICES concerning the graph-grammar-based solvers for the isogeometric finite element method by Keshav Pingali from ICES. His GALOIS system for concurrent execution of graph-grammar productions [83] seemed to be an ideal platform for implementing fast parallel graph-grammar-based solvers. Both sequential and parallel multi-thread graph-grammar-based solvers implemented in GALOIS outperformed the alternative sequential and parallel versions of the MUMPS solver [78].

At the same time, after inviting Robert van de Geijn to visit the Department of Computer Science at AGH University where I work, I came up with the idea of the automatic search for optimal elimination trees for a given class of grids [2]. This idea was inspired by van de Geijn’s work on the automatic generation of optimal dense algebra solvers, considering all possible implementations expressed by grammar productions for a given architecture of processor. We continue this research line with Mikhail Moshkov from KAUST and his PhD student Hassan AbouEisha. Motivated by the results of the automatic search, Anna Paszyńska has proposed several heuristic algorithms generating optimal elimination trees for two- and three-dimensional adaptive grids [67].

Recently I have realized that it is possible to design and implement solvers with linear computational cost for grids with singularities, namely for two-dimensional grids refined toward point or edge singularity [39, 44], as well as for three-dimensional grids anisotropically \( h \)-refined toward point, edge or face singularities, and for three-dimensional grids isotropically refined toward point or edge singularity. The three-dimensional grid isotropically refined toward face singularity can be solved with computational complexity \( O(N^{1.5}) \), identical to the complexity of the uniform two-dimensional grids. The other grids, refined in a mixed way toward point and edge, point and face, or point and edge and face singularities, can be solved with computational complexity between \( O(N) \) to \( O(N^{1.5}) \).

**Structure of the Book**

The main goal of this book is to teach graduate and PhD students, as well as researchers interested in the topic, how to design and implement efficient parallel multi-frontal solvers for mesh-based computations. The only prerequisite for reading the book is to understand some basic algebra transformation and know the basics of Java and Fortran programming languages. After reading the book, a graduate level student will be able to design and implement several parallel multi-frontal direct solvers that may be able to compete with other general purpose solvers when applied to linear systems of equations resulting from mesh-based computations.
The book is accompanied by exemplary JAVA and Fortran codes. The codes are available for download at http://www.ki.agh.edu.pl/FastSolvers.

In general, the structure of the book follows the structures of the global matrices resulting from the application of different one- two- and three-dimensional mesh-based methods. This is because different structures of the matrices imply different design patterns for the parallel multi-frontal solvers used for efficient LU factorizations.

In the second chapter, entitled Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems, I derive the parallel multi-frontal solver for one-dimensional mesh-based methods, including the finite difference method, the finite element method with linear and higher-order basis functions, the isogeometric collocation method, and the isogeometric finite element method, which generate tri-diagonal matrices. The dependency of the sub-chapters is presented in Figure P.1. Basically, I start from the simplest example, the one-dimensional finite difference method, and the sequential frontal solver algorithm for the tri-diagonal matrix. Later, I derive the algebraic structure of the multi-frontal direct solver algorithm, and present the decomposition of this algorithm into basic undividable tasks that I call graph-grammar productions. I construct the dependency relation between the tasks and derive the sets of tasks that can be executed concurrently on shared-memory parallel machines. I provide model Java code implementing the derived parallel algorithm. I must emphasize that the Java examples in this book are not optimized. They simply introduce the basic software engineering concepts behind the graph-grammar-based parallel multi-frontal solvers, and once learned they can be simply extended to more advanced tools and environments such as the GALOIS system for concurrent execution of graph-grammar productions over multi-core nodes, as well as NVIDIA CUDA for GPU graphic processing units.

In the following parts of the chapter I show how to extend the ideas and the source code for the one-dimensional finite difference method into the one-dimensional finite element method with linear basis functions, as well as into the isogeometric collocation method. In these cases, the only updates concern the element frontal matrices, and the structure of the solver and the rest of its code remain unchanged.

Next, I present some additional changes that must be done when we switch to a one-dimensional finite element method with bubble basis functions. I introduce some new graph-grammar productions dealing with interiors of elements and show how to update the structure of the parallel solver. The matrix generated from the one-dimensional finite element method with bubble basis functions can be reduced to the tri-diagonal system, by performing a static condensation. This is why this method is a part of the second chapter as well.

In Chapter 2, I discuss necessary updates to the structure of the solver and the source code when dealing with time dependent problems, including the simple Euler scheme and the general alpha-scheme for which \( \alpha = \frac{1}{2} \) generalizes to the Cranck–Nicolson scheme. This is illustrated in Figure P.1. The matrices for the non-stationary problems also have tri-diagonal structures, so the updates to the Java code are straightforward.
Figure P.1: Dependency plan for the parts of the book describing graph-grammar-based solvers for one-dimensional mesh-based computations.

In the following chapter, *Multi-Frontal Direct Solver Algorithm for Multi-Diagonal One-Dimensional Problems*, I discuss the two one-dimensional methods that lead to the multi-diagonal structure of the matrix, namely, the one-dimensional collocation method and the one-dimensional isogeometric finite element method with higher-order B-spline basis functions. I discuss the derivation of matrices and updates to the model Java code.

In the following part of the book, the ideas presented for one-dimensional mesh-based computations are generalized to the case of two-dimensional problems. The dependency structure of the next chapters is illustrated in Figure P.2. I show how to extend the Java code and graph-grammar produc-
Figure P.2: Dependency plan for the parts of the book describing graph-grammar-based solvers for two- and three-dimensional mesh-based computations resulting in a similar structure of the solver such as one-dimensional methods.

...
three-dimensional projection problems. I would like to emphasize that the structure of the solver for any single-equation elliptic problem will be identical, and we only need to replace the input element frontal matrices generated in some graph-grammar productions. Multi-equation elliptic problems such as e.g., linear elasticity problems or other problems such as the Stokes problem or the Maxwell problem will also require us to replace the single matrix entries by block entries related to multi-equations formulations.

In Chapter 4, I present how to update the structure of the solver and the model Java code to deal with the two-dimensional isogeometric finite element method over a regular patch of elements.
In the following part of the book, I introduce the multi-frontal solver algorithm for arbitrary two- or three-dimensional mesh-based computations that do not fall into any of the categories of the matrices and any of the categories of the meshes already considered in the previous chapters. The plan for this part of the book is presented in Figure P.3.

The general solver algorithm can deal with the two-dimensional finite difference method, two-dimensional finite element method, with either linear or higher-order basis functions as well as with isogeometric collocation and finite element methods, in particular working over regular grids. The construction of the multi-frontal solver algorithm for arbitrary mesh-based computations is based on the idea of the hypermatrix, with sub-matrices related to mesh supernodes with possibly higher-order polynomial basis functions. The order of the processing of the element frontal matrices is prescribed by the elimination tree. The general purpose solver takes the elimination tree as part of its input.

Thus, in the next chapter, Elimination Trees, I present different elimination trees carefully constructed for several two-dimensional and three-dimensional grids with singularities. I present the elimination trees for two-dimensional grids with point and edge singularities, as well as for three-dimensional grids with point, edge and face singularities. All the elimination trees deliver linear computational complexity of the sequential solver, except for the case of the tree for three-dimensional grids with face singularity, which results in $O(N^{1.5})$ computational complexity. The structure of this chapter is presented in Figure P.4. I also discuss in Chapter 10 the benefits resulting from the knowledge of the structure of the computational mesh, namely the techniques of reutilization and reuse.

The next chapter, Numerical Experiments, presents several numerical examples, as illustrated in Figures P.5 and P.6. The examples concern the GALOIS implementation of the graph-grammar-based solver for the two- and three-dimensional grids with singularities, the GPU implementation of the graph-grammar-based solver for the one- and two-dimensional isogeometric finite element method, as well as examples concerning the Fortran implementation for Linux cluster nodes. I also discuss the issue of fair comparison of two multi-frontal direct solvers.
Figure P.4: Dependency plan for the parts of the book describing elimination trees for different grids.
12.4 Graph-grammar-based multi-thread GALOIS solver for two-dimensional grids with singularities

12.1 Measuring the solver performance by means of execution time

12.2 Measuring the solver performance by means of the number of floating point operations (FLOPs)

12.3 Measuring the solver performance by means of efficiency and speedup

12.5 Graph-grammar-based multi-thread GALOIS solver for the three-dimensional grids with singularities

12.4 Graph-grammar GPU solver for the one-dimensional isogeometric finite element method

12.6 Measuring the solver performance by means of execution time

12.2 Measuring the solver performance by means of the number of floating point operations (FLOPs)

12.3 Measuring the solver performance by means of efficiency and speedup

12.7 Graph-grammar-based GPU solver for the two-dimensional isogeometric finite element method

Figure P.5: Dependency plan for the parts of the book describing numerical results.
Figure P.6: Dependency plan for the parts of the book describing numerical results.
Related Works

The solver algorithms presented in this book can be used to compute solutions of systems of linear equations obtained from several mesh-based methods. This includes the finite difference method, the finite element method with linear and hierarchical basis functions, and the isogeometric finite element and collocation methods.

The first mesh-based method discussed is the finite difference method. The method itself as well as its accuracy can be derived from the Taylor expansion [84]. The non-zero pattern of the systems of linear equations for the finite difference method is, in essence, similar to the structure of the systems resulting from the finite element method with linear basis functions. The finite element method allows one to approximate the solutions of the weak form of PDEs, whereas the finite difference method works with an approximation of the PDE itself.

The principles of the finite element method for different engineering problems are described in the books of Zienkiewicz [98] and Hughes [52]. Many engineering problems require the usage of computational meshes with local singularities. The computational meshes are refined in the areas where the singularities are identified by adaptive algorithms [28, 29]. Local $h$-refinements are essential to solve a variety of engineering problems [9, 54, 64, 82], and different versions of the $h$-adaptive algorithm have been designed for that purpose [10, 16, 17, 36, 65].

The $p$-adaptivity in turn requires addition of the hierarchical bubble basis functions [11], and the $hp$-adaptivity is the mixture of both the $h$ and $p$ methods [28, 29], delivering exponential convergence of the numerical error with respect to the number of degrees of freedom (number of basis functions, or mesh size).

The adaptive finite element method with hierarchical bubble basis functions provides $C^0$ global continuity of the solution, while $C^p$ continuity is restricted only to the interiors of finite elements. In some cases where the solution of the PDE is of the higher $C^p$ global continuity, the isogeometric finite element method can be used instead [24]. The isogeometric methods utilize B-splines as basis functions, and thus, they deliver $C^k$ global continuity [24]. The higher continuity obtained across elements allows IGA to attain optimal convergence rates for any polynomial order, while using fewer degrees of freedom [3, 13]. This reduced number of degrees of freedom may not immediately correlate with a reduction of the computational cost, since the density of the global system of linear equations and the solution time per degree of freedom increases with the global continuity [21, 23]. Despite of the increased cost of higher global continuity spaces, they have proven very popular and useful. For example, higher continuous spaces have allowed the solution of higher-order partial differential equations with elegance [19, 26, 27, 41, 42, 92]. They also allowed for the solution of several non-linear problems of engineering interest [14, 15, 18, 20, 22, 33, 49, 51].

The isogeometric collocation method [8] is a modern computational technique that is supposed to overcome some computational cost problems related to the isogeometric finite element method. Careful selection of the collocation points by, e.g., the Demko algorithm [25] guarantees the conver-
gence of the method. The structure of the global matrices generated by the isogeometric collocation method is similar to the one generated by the isogeometric finite element method, but with lower B-spline basis functions. This, and the fact that there is no need for numerical integration, makes this method an attractive alternative.

The multi-frontal solver algorithm [34, 35] is the state-of-the-art method for solving sparse linear systems of equations. It is a generalization of the frontal method originally proposed by Bruce Irons [53]. The general multi-frontal solvers are designed to deal with sparse systems of linear equations resulting from any application, not necessarily by the mesh-based methods. In this book I focus on the development of specific multi-frontal solvers for sparse systems of linear equations obtained from mesh-based computations only. I must emphasize that these solvers such as MUMPS [5, 6, 7, 63], PARDISO [60], SuperLU [96] or PaSTiX [48] are very good general purpose solvers, and they can be efficiently used for solving mesh-based problems as well as for solving linear systems resulting from other applications. However, knowledge of the topological structure of the computational mesh allows us to speed up mesh-based solvers in ways not easily available for classical general solvers. One of the ways of speeding up the solver is the reutilization of partial LU factorizations over unrefined parts of the mesh [77]. Another way to speed up mesh-based solvers is to reuse partial LU factorizations over identical parts of the mesh. We summarize these ideas in the chapter titled Reutilization and Reuse of Partial LU Factorizations.

The computational cost of the multi-frontal solver algorithm depends on the structure of the elimination tree it processes. The construction of the optimal elimination tree that results in the minimal number of operations performed by the solver for an arbitrary mesh is equivalent to the problem of finding an ordering of subtraction of rows in the Gaussian elimination procedure that also results in the minimal number of operations. This problem is NP-complete [97].

There are several heuristic algorithms for constructing efficient elimination trees. The most popular one is the nested-dissection method [55] available through the METIS library [56]. The nested-dissection algorithm is proven to generate optimal elimination trees over regular grids [59]. However, for non-regular grids this is no longer the case. Other popular algorithms include the minimum degree algorithm [47], the PORD algorithm [88], and their variations such as the approximate minimum degree algorithm [4], and the minimum cut algorithm [37]. Recently Paszyńska [67] proposed the so-called volume and neighbors algorithm constructing an elimination tree that uses additional knowledge on the structure of the mesh, not only the sparsity pattern of the global matrix. The volume and neighbors algorithm outperforms alternative ordering algorithms over grids that are h-refined toward singularities, under some restrictions [78]. An elimination tree that works well for sequential processing may not necessarily be optimal for parallel processing. Thus the tree rotation algorithm is used, transforming the not well-balanced sequential trees into well-balanced trees more suitable for parallel processing [38].

In the chapter, Elimination Trees, I define several elimination trees, starting with regular binary trees for one-dimensional problems, regular binary trees for two-dimensional grids with point and
Preface

anisotropic edge singularities, regular binary trees for three-dimensional grids with point singularities, and anisotropic edge and face singularities. These trees are optimal for parallel processing and they result in logarithmic computational complexity of the multi-frontal solver algorithm, provided the number of available cores is large enough. I also propose elimination trees designed for two-dimensional grids with edge singularities, and three-dimensional grids with edge and face singularities. The elimination trees utilized are optimal, in the sense that they have been obtained from a dynamic programming search [2] and constructed by a heuristic algorithm for either two-dimensional adaptive grids [78] or three-dimensional adaptive grids [67].

In this book I present how to design and implement graph-grammar-based solvers. I start with simple one-dimensional problems, and provide a full Java graph-grammar-based code for this model example. Next, I present how to extend this Java code to the more complicated cases, such as non-stationary problems, two-dimensional isogeometric problems, and two- or three-dimensional grids with singularities. The graph-grammars have already been successfully used for modeling generation and adaptation of two-dimensional grids with triangular, rectangular and mixed elements [68], as well as three-dimensional grids with hexahedral, tetrahedral or mixed elements [69, 71, 72]. The multi-frontal solver algorithm has been already expressed through graph-grammar-productions attributing finite elements with frontal matrices, to both two-dimensional adaptive grids [81] and three-dimensional adaptive grids [80]. In these papers we do not explicitly construct the elimination tree, but it is encoded in the order of execution of graph-grammar productions. However, we realized recently that graph-grammar productions can be obtained directly from the elimination tree and we implemented such a graph-grammar solver in the GALOIS system [78]. I present both approaches. For the first part of the book, where the elimination trees are binary well structured trees, the definition of graph-grammar productions for the solver algorithm is straightforward. For the case of two-dimensional grids with edge singularities, and three-dimensional grids with edge and face singularity, I present the graph-grammar expressing the solver algorithm over the hypergraph representation of the mesh [45, 46, 87], as well as the equivalent dependency graph presenting the relations between the graph-grammar productions, understood as basic undividable tasks that can be executed concurrently, set by set. For the construction of the dependency graph I refer the reader to the trace theory [32]. For scheduling of graph-grammar productions, I refer the reader to the simple schedulers resulting from the shading of the dependency graph, to the automatic schedulers implemented within GPU [94], as well as to the advanced scheduler from the GALOIS system [83].

Acknowledgment

I would like to acknowledge all the great scientists who helped me on my scientific way. Prof. Robert Schaefer from AGH University; Prof. Leszek Demkowicz from ICES, UT, Austin; Prof. Carlos Torres-Verdin who leads the Joint Research Consortium on Formation Evaluation at UT, Austin; Prof. David Pardo from the University of the Basque Country, Bilbao, Spain; Prof. Victor
Preface

Calo from KAUST; Prof. Mikhail Moshkov from KAUST; Prof. Keshav Pingali from ICES, UT, Austin; Prof. Robert van de Geijn from the Department of Computer Science, UT, Austin; Prof. Luis Garcia-Castillo, from University of Carlos III, Madrid, Spain; and Prof. Ignacio Muga, from the Catholic University of Valparaiso, Chile.

I would also acknowledge work with my former and current PhD students, including Piotr Gurgul, Arkadiusz Szymczak, Marcin Sieniek, Maciej Woźniak, Konrad Jopec, Marcin Łoś, Marcin Skotnickzy, Paweł Lipski, and also Damian Goik and Hassan Abou-Eisha. I also acknowledge the many students from the Department of Computer Science, AGH University, who tested the Java codes developed in this book.

My work on direct solvers in Poland in years 2005–2015 has been supported by the Foundation for Polish Science, the Polish Ministry of Scientific Research and Highest Education, and the Polish National Science Center.
Chapter 1

Multi-Frontal Direct Solver Algorithm
for Tri-Diagonal and Block-Diagonal
One-Dimensional Problems

In this chapter, we introduce the multi-frontal solver algorithm for a class of one-dimensional problems resulting in tri-diagonal and block-diagonal matrices. In particular, this class includes the one-dimensional finite difference method (tri-diagonal matrix), the one-dimensional finite element method with linear basis functions (tri-diagonal matrix), the one-dimensional isogeometric collocation method with quadratic B-spline basis functions (tri-diagonal matrix), and the one-dimensional finite element method with bubble basis functions (block-diagonal matrix).

1.1 Derivation of the Linear System for the One-Dimensional
Finite Difference Method

We focus on an exemplary simple one-dimensional elliptic problem with mixed Dirichlet and Neumann indexboundary conditions:

\[-\frac{d}{dx} \left( \frac{du(x)}{dx} \right) = 0 \quad (1.1)\]

\[u(0) = 0 \quad (1.2)\]

\[\frac{du(1)}{dx} = 1 \quad (1.3)\]
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal
One-Dimensional Problems

In the finite difference method, we select \( N + 1 \) points \( \{ x_i \}_{i=0,...,N} = \{ \frac{i}{N} \}_{i=0,...,N} \), distributed over the domain \([0, 1]\). First, we take values of problem (1.3) at these points:

\[
\begin{align*}
\frac{d}{dx} \left( \frac{du(x)}{dx} \right) &= 0 & (1.4) \\
\frac{du(x_{N})}{dx} &= 1 & (1.5)
\end{align*}
\]

Second, we replace the first and second derivatives at (1.4-1.6) with their finite difference discretizations:

\[
\begin{align*}
\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} &= 0 & \quad i = 1, ..., N - 1 & (1.7) \\
\frac{u_N - u_{N-1}}{h} &= 1 & (1.8)
\end{align*}
\]

Here \( u_i = u(x_i) = u \left( \frac{i}{N} \right) \), \( h = \frac{1}{N} \). We cancel out the \( h^2 \) factors and organize the terms in rows to obtain

\[
\begin{align*}
u_0 &= 0 & (1.10) \\
u_{i-1} - 2u_i + u_{i+1} &= 0 & \quad i = 1, ..., N - 1 & (1.11) \\
-u_{N-1} + u_N &= h & (1.12)
\end{align*}
\]

Third, we construct a global system of linear equations

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -2 & 1 & 0 & 0 & 0 & 0 \\
0 & \ldots & \ldots & \ldots & 0 & 0 & 0 \\
0 & 0 & 1 & -2 & 1 & 0 & 0 \\
0 & 0 & 0 & \ldots & \ldots & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -2 & 1 \\
0 & 0 & 0 & 0 & 0 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
\vdots \\
u_i \\
\vdots \\
u_{N-1} \\
u_N
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
\vdots \\
0 \\
\vdots \\
0 \\
h
\end{bmatrix}
\]

(1.13)

Let us focus on the exemplary problem with 4 points, \( \frac{0}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{3} \), and follow the steps of the Gaussian elimination procedure. We start with the forward elimination:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & -2 & 1 & 0 \\
0 & 1 & -2 & 1 \\
0 & 0 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0 \\
h
\end{bmatrix}
\]

(1.14)
1.1 Derivation of the Linear System for the One-Dimensional Finite Difference Method

We divide the first row by the diagonal entry:

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ h \end{bmatrix} \]  \hfill (1.15)

We subtract the first row from the second row, multiplied by the entry from the first column, to get zero below the diagonal:

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 - 1 \times 1 = 0 & -2 - 0 \times 1 = -2 & 1 - 0 \times 1 = 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ h \end{bmatrix} \]  \hfill (1.16)

We divide the second row by the diagonal entry:

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1/2 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ h \end{bmatrix} \]  \hfill (1.17)

We subtract the second row from the third row, multiplied by the entry from the second column, to get zero below the diagonal:

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1/2 & 0 \\ 0 & 1 & -1 \times 1 = 0 & -2 \times (-1/2) + 1 = -3/2 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ h \end{bmatrix} \]  \hfill (1.18)

We divide the third row by the diagonal entry:

\[ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1/2 & 0 \\ 0 & 0 & 1 & -2/3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ h \end{bmatrix} \]  \hfill (1.19)
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

We subtract the third row from the fourth row, multiplied by the entry from the third column, to get zero below the diagonal: \(4^{rd} = 4^{rd} - 3^{rd} \cdot A(4, 3) = 4^{rd} - 3^{rd} \cdot (-1)\)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & -1/2 & 0 \\
0 & 0 & 1 & -2/3 \\
0 & 0 & -1 & -1/2 \\
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0 \\
-1 - 1 \cdot (-1) = 0 \\
1 - (-2/3) \cdot (-1) = 1/3 \\
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3 \\
\end{bmatrix}
\begin{bmatrix}
h - 0 \cdot (-1) = h \\
\end{bmatrix}
\]

Finally, we divide the fourth row by the diagonal entry: \(4^{rd} = 4^{rd}/A(4, 4) = 4^{rd}/(1/3)\)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & -1/2 & 0 \\
0 & 0 & 1 & -2/3 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0 \\
3h \\
\end{bmatrix}
\]

We continue with backward substitution:

\(u_3 = 3h/1 = 3h\)
\(u_2 - 2/3u_3 = 0 \rightarrow u_2 = 2/3 \cdot 3h = 2h\)
\(u_1 - 1/2u_2 = 0 \rightarrow u_1 = 1/2 \cdot 2h = h\)
\(u_0 = 0\)

We implement the Gaussian elimination for this tri-diagonal matrix by using only 3 columns and \(N + 1\) rows, so the resulting Gaussian elimination procedure has \(O(N)\) computational complexity.

\[
\begin{bmatrix}
\mathbf{X} & 1 & 0 \\
1 & -2 & 1 \\
\ldots & \ldots & \ldots \\
1 & -2 & 1 \\
\ldots & \ldots & \ldots \\
-1 & 1 & X \\
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_i \\
u_i \\
u_i \\
u_{N-1} \\
u_N \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
\ldots \\
0 \\
\ldots \\
0 \\
h \\
\end{bmatrix}
\]

However, having multiple cores available, we can solve the above system in parallel. The resulting computational complexity is \(O(logN)\). This can be done by partitioning the matrix into multiple frontal matrices, as presented in Figure 2.1.
1.2 Algebraic Algorithm of the Multi-Frontal Solver

The multi-frontal solver algorithm starts by decomposing the system of linear equations (1.22). In order to decompose the global matrix into a set of local linear systems that sum up to the original matrix, we perform the following partition. We consider an interval \((x_i, x_{i+1})\) and write down all the discrete equations that include unknowns \(u_i\) and \(u_{i+1}\):

\[
\begin{align*}
\ldots & \\
\begin{align*}
u_{i-2} - 2u_{i-1} + u_i &= 0 \\
u_{i-1} - 2u_i + u_{i+1} &= 0 \\
u_i - 2u_{i+1} + u_{i+2} &= 0
\end{align*} & (1.24) \\
\ldots & (1.27)
\end{align*}
\]

We partition each equation into two parts, associated with two intervals:

\[
\begin{align*}
\ldots & \\
\begin{align*}
u_{i-2} - u_{i-1} &= 0 & x \in [x_{i-2}, x_{i-1}], & u_i - u_{i-1} = 0 & x \in [x_{i-1}, x_i] \\
u_{i-1} - u_i &= 0 & x \in [x_{i-1}, x_i], & u_{i+1} - u_i = 0 & x \in [x_i, x_{i+1}] \\
u_i - u_{i+1} &= 0 & x \in [x_i, x_{i+1}], & u_{i+2} - u_{i+1} = 0 & x \in [x_{i+1}, x_{i+2}]
\end{align*} & (1.28)
\end{align*}
\]

We now pick up sub-equations that are related to interval \([x_{i-1}, x_i]\) only:

\[
\begin{align*}
u_i - u_{i-1} &= 0 & (1.29) \\
u_{i-1} - u_i &= 0 & (1.30)
\end{align*}
\]
They form the following system of linear equations:

\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_{i-1} \\
u_i
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\] (1.31)

Similarly, we pick up the sub-equations that are related to interval \([x_i, x_{i+1}]\) only:

\[
u_i - u_{i-1} = 0
\] (1.32)
\[
u_{i-1} - u_i = 0
\] (1.33)

They form the following system of linear equations:

\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_i \\
u_{i+1}
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\] (1.34)

We perform such decompositions for all the intervals, and, finally, the original system of linear equations for the one-dimensional finite difference method is decomposed into the following set of systems:

\[
\begin{bmatrix}
1 & 0 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_0 \\
u_1
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\] ...
\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_{i-1} \\
u_i
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_{N-2} \\
u_{N-1}
\end{bmatrix} =
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\]
\[
\begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
u_{N-1} \\
u_N
\end{bmatrix} =
\begin{bmatrix}
0 \\
h
\end{bmatrix}
\] (1.35)

Note that the set of systems of linear equations (1.35) is not equivalent to the original system (1.22) unless we sum the systems of linear equations (1.35) up! Indeed, each individual system (1.35) is contradictory. This decomposition of the system of linear equations into several local frontal matrices – which are not fully assembled yet – is the input for the multi-frontal solver algorithm.
1.2 Algebraic Algorithm of the Multi-Frontal Solver

The multi-frontal solver algorithm merges the first and the second, the third and the fourth, and the fifth and the sixth matrix to obtain

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & -2 & 1 \\
0 & 1 & -1 \\
-1 & 1 & 0 \\
1 & -2 & 1 \\
0 & 1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
u_3 \\
u_3 \\
u_4 \\
u_5 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\] (1.36)

\[
\begin{bmatrix}
-1 & 1 & 0 \\
1 & -2 & 1 \\
0 & 1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
u_5 \\
u_6 \\
u_7 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
-h \\
\end{bmatrix}
\] (1.38)

Note that only the central row is fully summed up for all the matrices. We call such a row the fully assembled row. In general, the first and the third rows are not fully assembled yet. The only exceptions are the first system, which has the first row fully assembled, and the last system, which has the last row fully assembled. The multi-frontal solver reorders the equations (unknowns) in the system to place the fully-assembled central row at the beginning of the system. We treat all the systems in the same way, making no distinction for the first and the last ones. In the next section, when we will assign the computational tasks to process the frontal matrices concurrently, this will allow us to design the same computational tasks for all the frontal matrices. We do not need to distinguish the first and the last frontal matrix. The resulting reordered systems follow:

\[
\begin{bmatrix}
-2 & 1 & 1 \\
0 & 1 & 0 \\
1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
u_2 \\
u_1 \\
u_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\] (1.39)

\[
\begin{bmatrix}
-2 & 1 & 1 \\
1 & -1 & 0 \\
1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
u_4 \\
u_3 \\
u_5 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\] (1.40)

\[
\begin{bmatrix}
-2 & 1 & 1 \\
1 & -1 & 0 \\
1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
u_6 \\
u_5 \\
u_7 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
-h \\
\end{bmatrix}
\] (1.41)
At this point, after merging and rearranging the input frontal matrices, the multi-frontal solver algorithm performs the elimination of the first fully-assembled row. The first row is subtracted from the second and the third rows. This is done by the following algorithm:

```plaintext
1   divider = A(0,0)
2   for icol=0,2
3       A(0,icol)=A(0,icol)/divider
4   end loop
5   b(0)=b(0)/divider
6   for irow=1,2
7       multiplier = A(irow,0)
8       A(irow,0)=0
9       for icol=1,2
10          A(irow,icol) = A(irow,icol) - A(irow,0) * A(0,icol)
11      end loop
12     b(irow)=b(irow)-A(irow,0)*b(0)
13   end loop
```

We subtract the fully-assembled row from the rows that are not fully assembled, because the subtractions and additions are commutative. That is, we subtract the fully assembled row at this point, and in the following step, we add the remaining part of the non-fully assembled row. The systems, resulting from partial forward eliminations, are:

\[
\begin{bmatrix}
1 & -\frac{1}{2} & -\frac{1}{2} \\
0 & 1 & 0 \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_2 \\
u_1 \\
u_3
\end{bmatrix}
= \begin{bmatrix}0 \\
0 \\
0
\end{bmatrix} \quad (1.42)
\]

\[
\begin{bmatrix}
1 & -\frac{1}{2} & -\frac{1}{2} \\
0 & -\frac{1}{2} & \frac{1}{2} \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_4 \\
u_3 \\
u_5
\end{bmatrix}
= \begin{bmatrix}0 \\
0 \\
0
\end{bmatrix} \quad (1.43)
\]

\[
\begin{bmatrix}
1 & -\frac{1}{2} & -\frac{1}{2} \\
0 & -\frac{1}{2} & \frac{1}{2} \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_6 \\
u_5 \\
u_7
\end{bmatrix}
= \begin{bmatrix}0 \\
0 \\
h
\end{bmatrix} \quad (1.44)
\]

Now, we focus on the right bottom \(2 \times 2\) sub-matrices, the part that is still missing contributions from the neighboring intervals. These sub-matrices are called the Schur complements. The multi-frontal solver algorithm merges the first and the second Schur complement matrices to obtain a new
1.2 Algebraic Algorithm of the Multi-Frontal Solver

3 × 3 system:
\[
\begin{bmatrix}
1 & 0 & 0 \\
\frac{1}{2} & -1 & \frac{1}{2} \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_3 \\
u_5
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]
(1.45)

Only the central row is fully assembled at this level. We reorder the rows in order to have the fully-assembled row at the beginning:
\[
\begin{bmatrix}
-1 & \frac{1}{2} & \frac{1}{2} \\
0 & 1 & 0 \\
\frac{1}{2} & 0 & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_3 \\
u_1 \\
u_5
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]
(1.46)

We eliminate the fully-assembled row by subtracting it from the second and the third rows (which are not fully assembled yet). This results in:
\[
\begin{bmatrix}
1 & -\frac{1}{2} & -\frac{1}{2} \\
0 & 1 & 0 \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_3 \\
u_1 \\
u_5
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]
(1.47)

The right bottom 2 × 2 sub-matrix is our new Schur complement. Finally, the multi-frontal solver algorithm merges the last Schur complement with the third Schur complement to get the root system:
\[
\begin{bmatrix}
1 & 0 & 0 \\
\frac{1}{2} & -\frac{3}{4} & \frac{1}{2} \\
0 & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_5 \\
u_7
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
-h
\end{bmatrix}
\]
(1.48)

This final problem, called the root problem, is fully assembled, and we perform the full forward elimination followed by the backward substitution. This is done by standard Gaussian elimination, implemented without the loops, to speed up the execution:

```plaintext
1  divider = A(0,0)
2  for icol=0,2
3      A(0,icol)=A(0,icol)/divider
4  end loop
5  b(0)=b(0)/divider
6  for irow=1,2
7      multiplier = A(irow,0)
8      A(irow,0)=0
9      for icol=1,2
10         A(irow,icol) = A(irow,icol) - A(irow,0) * A(0,icol)
```
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

11 end loop
12 b(irow)=b(irow)-A(irow,0)*b(0)
13 end loop
14 divider = A(1,1)
15 for icol=1,2
16 A(1,icol)=A(1,icol)/divider
17 end loop
18 b(1)=b(1)/divider
19 for irow=2,2
20 multiplier = A(irow,1)
21 A(irow,1)=1
22 for icol=1,2
23 A(irow,icol) = A(irow,icol) - A(irow,1) * A(1,icol)
24 end loop
25 b(irow)=b(irow)-A(irow,1)*b(1)
26 end loop
27 divider = A(2,2)
28 A(2,2)=1
29 b(2) = b(2)/divider
30 //backward substitutions
31 x(2)=b(2)/A(2,2)
32 x(1)=(b(1)-A(1,2)*x(2))/a(1,1)
33 x(0)=(b(0)-A(0,1)*x(1)-A(0,2)*x(2))/a(0,0)

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & -\frac{2}{3} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_5 \\
u_7
\end{bmatrix}
=
\begin{bmatrix}
0 \\
0 \\
6h
\end{bmatrix}
\tag{1.49}
\]

We get the solution with \( h = \frac{1}{6} \).

\[
\begin{align*}
u_7 &= 1 & (1.50) \\
u_5 &= \frac{2}{3}u_7 = \frac{2}{3} & (1.51) \\
u_1 &= 0 & (1.52)
\end{align*}
\]

Finally we proceed with recursive backward substitutions:

\[
\begin{align*}
u_3 &= \frac{1}{2}u_1 + \frac{1}{2}u_5 = \frac{1}{3} & (1.53) \\
u_2 &= \frac{1}{2}u_1 + \frac{1}{2}u_3 = \frac{1}{6} & (1.54)
\end{align*}
\]
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

\[
\begin{align*}
  u_4 &= \frac{1}{2} u_3 + \frac{1}{2} u_5 = \frac{1}{2} \\
  u_6 &= \frac{1}{2} u_5 + \frac{1}{2} u_7 = \frac{5}{6}
\end{align*}
\]

(1.55) (1.56)

Exercises

1. Please check that the solution obtained from the multi-frontal algorithm is identical to the one obtained from the Gaussian elimination.

2. Please compute the round-off error as a relative error the numerical and exact solution \( u = 1 \), and plot the round-off error for growing the number of intervals.

1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

In this sub-chapter we decompose the multi-frontal solver algorithm into basic indivisible tasks. We call these tasks graph-grammar productions. We analyze the partial relation between these tasks. We also analyse how the solver can be run concurrently. We present an object-oriented Java implementation of a graph-grammar-based multi-frontal solver.

We start with the data structure suitable for storing systems of linear equations decomposed into sub-systems (1.23–1.34). Note that the multi-frontal solver algorithm works over a binary elimination tree, and the frontal matrices are stored at the nodes of the tree.

We create a Vertex class, representing an elimination tree node.

```java
1 class Vertex {
2     Vertex (Vertex Left, Vertex Right, Vertex Parent, String Lab){
3         this.m_left=Left;
4         this.m_right=Right;
5         this.m_parent=Parent;
6         this.m_label=Lab;
7         m_a = new double[3][3];
8         m_b = new double[3];
9         m_x = new double[3];
10     }
11     String m_label;
12     Vertex m_left;
13     Vertex m_right;
14     Vertex m_parent;
15     double[][] m_a;
```
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

The construction of the elimination tree is a multi-stage process. We start from construction of the root of the tree, followed by construction of the two son nodes, up to the leaves of the tree. We create several tasks, responsible for creation of the root, the internal nodes, and the leaf nodes. Following [66], we call these tasks graph-grammar productions, or shortly productions. We propose the following three productions, called $P_1$, $P_2$ and $P_3$.

```java
class P1 extends Production {
    P1(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex S) {
        System.out.println("p1");
        Vertex T1 = new Vertex(null, null, S, "T");
        Vertex T2 = new Vertex(null, null, S, "T");
        S.set_left(T1);
        S.set_right(T2);
        S.set_label("root");
        return S;
    }
}
class P2 extends Production {
    P2(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("p2");
        Vertex T1 = new Vertex(null, null, T, "T");
        Vertex T2 = new Vertex(null, null, T, "T");
        T.set_left(T1);
        T.set_right(T2);
    }
}
class P3 extends Production {
    P3(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("p3");
        Vertex T1 = new Vertex(null, null, T, "T");
        Vertex T2 = new Vertex(null, null, T, "T");
        T.set_left(T1);
        T.set_right(T2);
    }
}
```
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

```java
10 T.set_right(T2);
11 T.set_label("int");
12 return T;
13 }
14 }
```

class P3 extends Production {
    P3(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("p3");
        Vertex T1 = new Vertex(null, null, T, "node");
        Vertex T2 = new Vertex(null, null, T, "node");
        T.set_left(T1);
        T.set_right(T2);
        T.set_label("leaf");
        return T;
    }
}
```

The construction of the elimination tree in our example can be expressed by the following execution of these productions, illustrated in Figure 1.2.

The execution of the productions form a sequence of computational tasks. We name our tasks following the names of the productions. We add suitable indices. We group all the names of the tasks and construct the alphabet $A$ of tasks:

$$A = \{(P_1), (P_2)_1, (P_2)_2, (P_2)_3, (P_2)_4, (P_3)_1, (P_3)_2, (P_3)_3, (P_3)_4, (P_3)_5, (P_3)_6\} \quad (1.57)$$

We plot the dependency relation between tasks, as it is presented on the left panel in Figure 1.3. Additionally, we shade the dependency graph, in such a way that different shades of gray represent sets of tasks that can be executed concurrently.

We can also get an analogous result by employing the trace theory [32, 66]. Namely, we first introduce the sequence of tasks representing the sequential execution of the solver algorithm

$$\{(P_1) - (P_2)_1 - (P_2)_2 - (P_2)_3 - (P_2)_4 - (P_3)_1 - (P_3)_2 - (P_3)_3 - (P_3)_4 - (P_3)_5 - (P_3)_6 \} \quad (1.58)$$

Next, we define the dependency relation between the tasks

$$(P_1)D\{(P_2)_1, (P_2)_2\}$$
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

Figure 1.2: Construction of the exemplary elimination tree by execution of productions $(P1) - (P2)_{1} - (P2)_{2} - (P2)_{3} - (P2)_{4} - (P3)_{1} - (P3)_{2} - (P3)_{3} - (P3)_{4} - (P3)_{5} - (P3)_{6}$.

Figure 1.3: **Left panel:** Dependency graph between tasks. **Right panel:** Shading of the dependency graph.

\[
\begin{align*}
(P2)_{1} D & \{(P2)_{3}, (P2)_{4}\} \\
(P2)_{3} D & \{(P3)_{1}, (P3)_{2}\} \\
(P2)_{4} D & \{(P3)_{3}, (P3)_{4}\} \\
(P2)_{2} D & \{(P3)_{5}, (P3)_{6}\}
\end{align*}
\]
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

The group of tasks that can be executed in parallel, set by set, can be obtained by deriving the Foata Normal Form defined in the following way

\[
\begin{align*}
[a_1^1 a_2^1 ... a_l^1] [a_1^2 a_2^2 ... a_l^2] ... [a_1^n a_2^n ... a_l^n] \\
\forall k \forall i, j \in \{1, ..., l_k\} a_k^i I a_k^j, i <> j, I = A \times A - D \\
\forall k \forall i \in \{1, ..., l_k\} \exists j \in \{1, ..., l_{k-1}\} a_k^{i-1} I a_k^j
\end{align*}
\] (1.60) (1.61) (1.62) (1.63)

In other words, in the Foata Normal Form the tasks are sorted into sets such that all tasks from a set are independent, and for each task from the next set there exist at least one dependent task from the previous set. Note that each task from the next set can depend on only one task from the previous set, not from all of them. The Foata Normal Form that can be generated from the sequence of tasks (1.59) is the following

\[
\begin{align*}
((P1)) \ [(P2)_1(P2)_2] \ [(P2)_3(P2)_4(P3)_5(P3)_6] \ [(P3)_1(P3)_2(P3)_3(P3)_4]
\end{align*}
\] (1.64)

This corresponds to the four shades of gray used for the graph shading on the right panel in Figure 1.3.

The concurrent execution of the tasks must be controlled by some scheduler. Let us focus on our simple exemplary Java code. We utilize a `CountDownLatch` class from the `java.util.concurrent` library for synchronization of the tasks.

Additionally, we need a base `Production` class to represent our graph-grammar productions:

```java
import java.util.concurrent.CountDownLatch;
abstract class Production extends Thread {
    Production(Vertex Vert, CountDownLatch Barrier) {
        m_vertex = Vert;
        m_barrier = Barrier;
    }
    //returns first vertex from the left
    abstract Vertex apply(Vertex v);
    //run the thread
    public void run() {
        m_barrier.inc();
        //apply the production
        m_vertex = apply(m_vertex);
        m_barrier.countDown();
    }
    //vertex where the production will be applied
    Vertex m_vertex;
```
Having the productions for construction of the tree nodes, the barrier class, and the order of execution of the graph-grammar productions, we implement the Executor class that acts as the scheduler:

```java
import java.util.concurrent.CountDownLatch;

class Executor extends Thread {
    public synchronized void run() {
        Vertex S = new Vertex(null,null,null,"S");
        //schedule tasks in sets
        try {
            //[(P1)]
            CountDownLatch barrier = new CountDownLatch(1);
            P1 p1 = new P1(S,barrier);
            p1.start();
            barrier.await();
            //[(P2)(P2)2]
            barrier = new CountDownLatch(2);
            P2 p2a = new P2(p1.m_vertex.m_left,barrier);
            P2 p2b = new P2(p1.m_vertex.m_right,barrier);
            p2a.start();
            p2b.start();
            barrier.await();
            //[(P2)(P2)4(P3)5(P3)6]
            barrier = new CountDownLatch(4);
            P2 p2c = new P2(p2a.m_vertex.m_left,barrier);
            P2 p2d = new P2(p2a.m_vertex.m_right,barrier);
            P3 p3a = new P3(p2b.m_vertex.m_left,barrier);
            P3 p3b = new P3(p2b.m_vertex.m_right,barrier);
            p2c.start();
            p2d.start();
            p3a.start();
            p3b.start();
            barrier.await();
            //[(P3)(P3)2(P3)3(P3)4]
        }
    }
}
```
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

```java
32    barrier = new CountDownLatch(4);
33    P3 p3c = new P3(p2c.m_vertex.m_left, barrier);
34    P3 p3d = new P3(p2c.m_vertex.m_right, barrier);
35    P3 p3e = new P3(p2d.m_vertex.m_left, barrier);
36    P3 p3f = new P3(p2d.m_vertex.m_right, barrier);
37    p3c.start();
38    p3d.start();
39    p3e.start();
40    p3f.start();
41    } catch
42         (InterruptedException e) {
43             e.printStackTrace();
44         }
45     }

Next, we create graph-grammar productions responsible for constructing the frontal matrices

```java
1    class A extends Production {
2        A(Vert, CountDownLatch Barrier) {
3            super(Vert, Barrier);
4        }
5        Vertex apply(Vertex T) {
6            System.out.println("A");
7            T.m_a[1][1]=1.0;
8            T.m_a[2][1]=-1.0;
9            T.m_a[1][2]=-1.0;
10           T.m_a[2][2]=1.0;
11           T.m_b[1]=0.0;
12           T.m_b[2]=0.0;
13           return T;
14       }
15    }

1    class A1 extends Production {
2        A1(Vert, CountDownLatch Barrier) {
3            super(Vert, Barrier);
4        }
5        Vertex apply(Vertex T) {
6```
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

6    System.out.println("A1");
7    T.m_a[1][1]=1.0;
8    T.m_a[2][1]=-1.0;
9    T.m_a[1][2]=0.0;
10   T.m_a[2][2]=1.0;
11   T.m_b[1]=0.0;
12   T.m_b[2]=0.0;
13   return T;
14 }
15 }

class AN extends Production {
    AN(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("AN");
        T.m_a[1][1]=1.0;
        T.m_a[2][1]=-1.0;
        T.m_a[1][2]=-1.0;
        T.m_a[2][2]=1.0;
        T.m_b[1]=0.0;
        T.m_b[2]=h;
        return T;
    }
}

We also need to implement productions responsible for the merging of frontal matrices and for eliminating fully assembled rows

class A2 extends Production {
    A2(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("A2");
        T.m_a[0][0] = T.m_left.m_a[2][2] + T.m_right.m_a[1][1];
        T.m_a[1][0] = T.m_left.m_a[1][2];
    }
}
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

```java
class E2 extends Production {
    E2(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }

    Vertex apply(Vertex T) {
        System.out.println("E2");
        T.m_a[2][0] = T.m_right.m_a[2][1];
        T.m_a[0][1] = T.m_left.m_a[2][1];
        T.m_a[1][1] = T.m_left.m_a[1][1];
        T.m_a[2][1] = 0.0;
        T.m_a[0][2] = T.m_right.m_a[1][2];
        T.m_a[1][2] = 0.0;
        T.m_a[2][2] = T.m_right.m_a[2][2];
        T.m_b[0] = T.m_left.m_b[2] + T.m_right.m_b[1];
        T.m_b[1] = T.m_left.m_b[1];
        T.m_b[2] = T.m_right.m_b[2];
        return T;
    }
}

class Aroot extends A2 {
    Aroot(Vertex Vert, CountDownLatch Barrier) {
```
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

```java
1  class Eroot extends Production {
2      Eroot(Vertex Vert, CountDownLatch Barrier) {
3          super(Vert, Barrier);
4      }
5   }
6
7   Vertex apply(Vertex T) {
8       System.out.println("Eroot");
9       // divide first row by diagonal
10      T.m_b[1] /= T.m_a[1][1];
11      T.m_a[1][2] /= T.m_a[1][1];
12      T.m_a[1][0] /= T.m_a[1][1];
13      T.m_a[1][1] /= T.m_a[1][1];
14      // 2nd = 2nd - 1st * diag
15      T.m_b[0] -= T.m_b[1] * T.m_a[0][1];
16      T.m_a[0][2] -= T.m_a[1][2] * T.m_a[0][1];
17      T.m_a[0][0] -= T.m_a[1][0] * T.m_a[0][1];
18      T.m_a[0][1] -= T.m_a[1][1] * T.m_a[0][1];
19      // divide second row by diagonal
20      T.m_b[0] /= T.m_a[0][0];
21      T.m_a[0][2] /= T.m_a[0][0];
22      T.m_a[0][0] /= T.m_a[0][0];
23      // 3rd = 3rd - 2nd * diag
24      T.m_b[2] -= T.m_b[0] * T.m_a[2][0];
25      T.m_a[2][2] -= T.m_a[0][2] * T.m_a[2][0];
26      T.m_a[2][0] -= T.m_a[0][0] * T.m_a[2][0];
27      // divide third row by diagonal
28      T.m_b[2] /= T.m_a[2][2];
29      T.m_a[2][2] /= T.m_a[2][2];
30      // b.s.
```
Finally, we need productions for backward substitution

```java
class BS extends Production {
    BS(Vertex Vert, CountDownLatch Barrier) {
        super(Vert, Barrier);
    }
    Vertex apply(Vertex T) {
        System.out.println("BS");
        if (T.m_label.equals("node")) return T;
        T.m_left.m_x[1] = T.m_x[1];
        T.m_left.m_x[2] = T.m_x[0];
        T.m_left.m_x[0] = (T.m_left.m_b[0] - T.m_left.m_a[0][1] * T.m_left.m_x[1] - T.m_left.m_a[0][2] * T.m_left.m_x[2]) / T.m_left.m_a[0][0];
        T.m_right.m_x[1] = T.m_x[0];
        T.m_right.m_x[2] = T.m_x[2];
        T.m_right.m_x[0] = (T.m_right.m_b[0] - T.m_right.m_a[0][1] * T.m_right.m_x[1] - T.m_right.m_a[0][2] * T.m_right.m_x[2]) / T.m_right.m_a[0][0];
        return T;
    }
}
```

Please note that the execution of the solver algorithm over the elimination tree in our example is expressed by the following sequence (executions of graph-grammar productions), illustrated in Figure 1.4. The dependency relation between tasks and the shading of the dependency graph allows us to define sets of tasks that can be executed concurrently, set by set.

We name our tasks by graph-grammar productions with suitable indices, and construct alphabet $A$ of tasks:

$$A = \{(A_1), (A_1)_1, (A_1)_2, (A_2)_1, (A_2)_2, (A_2)_3, (E_2)_1, (E_2)_2, (E_2)_3, (AN), \}$$
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

Figure 1.4: Execution of the graph-grammar productions

\[(A_1) - (A)_1 - (A)_2 - (A)_3 - (A)_4 - (AN) - (A_2)_1 - (A_2)_2 - (A_2)_3 - (E_2)_1 - (E_2)_2 - (E_2)_3 - (Aroot) - (Eroot)\]

representing the multi-frontal solver algorithm running over the exemplary elimination tree.

\[(Aroot), (Eroot), (BS)_1, (BS)_2, (BS)_3, (BS)_4\] (1.65)

We plot the dependency relation between tasks, as presented on the top panel in Figure 1.5. Additionally, we shade the dependency graph, in such a way that the shades of gray represent sets of tasks that can be executed concurrently.

We can also get an analogous result by employing the trace theory [32, 66]. Namely, we first introduce the sequence of tasks representing the sequential execution of the solver algorithm:

\[(A1) - (A)_1 - (A)_2 - (A)_3 - (A)_4 - (AN) - (A2)_1 - (A2)_2 - (A2)_3 -
(E2)_1 - (E2)_2 - (E2)_3 - (Aroot) - (Eroot) - (BS)_1 - (BS)_2\] (1.66)

Next, we define the dependency relation between the tasks

\[
\{(A1), (A)_1\}D(A2)_1
\]
\[
\{(A)_2, (A)_3\}D(A2)_2
\]
\[
\{(A)_4, (AN)\}D(A2)_3
\]
\[
(A2)_1D(E2)_1
\]
\[
(A2)_2D(E2)_2
\]
\[
(A2)_3D(E2)_3
\]
\[
\{(E2)_1, (E2)_2\}D(A2)_4
\]
\[
(A2)_4D(E2)_4
\]
\[
\{(E2)_3(E2)_4\}D(Aroot)
\]
\[
(Aroot)D(Eroot)
\]
\[
(Eroot)D((BS)_1, (BS)_2)
\]
\[
(BS)_1D((BS)_3, (BS)_4)
\]
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

Figure 1.5: **Top panel:** Dependency graph between tasks. **Bottom panel:** Shading of the dependency graph.
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

The group of tasks that can be executed in parallel, set by set, can be obtained by deriving the Foata Normal Form

\[
\begin{align*}
&([A_1](A_2)(A_3)(A_4)(AN)] \quad \begin{bmatrix}
(A_2)_1 & (A_2)_2 & (A_2)_3 & \end{bmatrix} \quad \begin{bmatrix}
(E_2)_1 & (E_2)_2 & (E_2)_3 \end{bmatrix} \quad \begin{bmatrix}
(A_2)_4 & \end{bmatrix} \quad \begin{bmatrix}
(E_2)_4 \end{bmatrix}
\end{align*}
\]

(1.69)

\[
\begin{align*}
&[(E_{root})] \quad [(A_{root})] \quad ((BS)_1(BS)_2) \quad ((BS)_3(BS)_4)
\end{align*}
\]

(1.70)

which coincides with the four shades of gray used for the graph in the bottom panel in Figure 1.5.

The concurrent execution of the tasks is controlled again by our exemplary Java code.

```java
1 class Executor extends Thread {
2   public synchronized void run() {
3     // CONSTRUCTION OF ELIMINATION TREE
4     ...
5     // MULTI-FRONTAL SOLVER ALGORITHM
6     //\[[A_1](A_2)(A_3)(A_4)(AN)]
7     barrier = new CountDownLatch(6);
8     A1 localMat1 = new A1(p3c.m_vertex, barrier);
9     A localMat2 = new A(p3d.m_vertex, barrier);
10    A localMat3 = new A(p3e.m_vertex, barrier);
11    A localMat4 = new A(p3f.m_vertex, barrier);
12    A localMat5 = new A(p3a.m_vertex, barrier);
13    AN localMat6 = new AN(p3b.m_vertex, barrier);
14    localMat1.start(); localMat2.start(); localMat3.start();
15    localMat4.start(); localMat5.start(); localMat6.start();
16    barrier.await();
17     //\[[A_2](A_2)(A_2)_3]
18     barrier = new CountDownLatch(3);
19     A2 mergedMat1 = new A2(p2c.m_vertex, barrier);
20     A2 mergedMat2 = new A2(p2d.m_vertex, barrier);
21     A2 mergedMat3 = new A2(p2b.m_vertex, barrier);
22     mergedMat1.start();
23       mergedMat2.start(); mergedMat3.start();
24     barrier.await();
25     //\[[E_2](E_2)(E_2)_3]
26     barrier = new CountDownLatch(3);
27     E2 gaussElimMat1 = new E2(p2b.m_vertex, barrier);
28     E2 gaussElimMat2 = new E2(p2c.m_vertex, barrier);
29     E2 gaussElimMat3 = new E2(p2d.m_vertex, barrier);
```
1.3 Graph-Grammar-Based Model of Concurrency of the Multi-Frontal Solver Algorithm

29     gaussElimMat1.start(); gaussElimMat2.start();
30         gaussElimMat3.start();
31     barrier.await();
32     //[][](A2)_4]
33     barrier = new CountDownLatch(1);
34     A2 mergedMat4 = new A2(p2a.m_vertex, barrier);
35     mergedMat4.start();
36     barrier.await();
37     //[][](E2)_4]
38     barrier = new CountDownLatch(1);
39     E2 gaussElimMat4 = new E2(p2a.m_vertex, barrier);
40         gaussElimMat4.start();
41     barrier.await();
42     //[][](Aroot)
43     barrier = new CountDownLatch(1);
44     Aroot mergedRootMat = new Aroot(p1.m_vertex, barrier);
45     mergedRootMat.start();
46     barrier.await();
47     //[][](Eroot)
48     barrier = new CountDownLatch(1);
49     Eroot fullElimMat = new Eroot(p1.m_vertex, barrier);
50         fullElimMat.start();
51     barrier.await();
52     //[][](BS)_3(BS)_4]
53     barrier = new CountDownLatch(2);
54     BS backSub1 = new BS(p1.m_vertex, barrier);
55     BS backSub2 = new BS(p2a.m_vertex, barrier);
56     backSub1.start();
57     backSub2.start();
58     barrier.await();
59     //[][](BS)_3(BS)_4]
60     barrier = new CountDownLatch(2);
61     BS backSub3 = new BS(p2c.m_vertex, barrier);
62     BS backSub4 = new BS(p2d.m_vertex, barrier);
63     backSub3.start();
64     backSub4.start();
65     barrier.await();
66
Multi-Frontal Direct Solver Algorithm for Tri-Diagonal and Block-Diagonal One-Dimensional Problems

Exercises

1. Please add the TreeDrawer class that plots the elimination tree.
2. Please extend the code so it prints the solution vector.
3. Please extend the Executor class so it generates an arbitrary elimination tree of size $2^k$.
4. Please extend the Executor class so it generates an arbitrary elimination tree of any size.

1.4 One-Dimensional Finite Element Method with Linear Basis Functions

In the finite element method, we transfer the strong formulation (1.1–1.3) into the weak formulation. The weak form is obtained by using $L^2$ inner products with test function $v$, integrating by parts, and incorporating boundary conditions:

$$
- \int_0^1 \frac{d}{dx} \left( \frac{du(x)}{dx} \right) v(x) \, dx = 0,
$$

(1.71)

$$
\int_0^1 \frac{du(x)}{dx} \frac{dv(x)}{dx} \, dx + \left[ \frac{du(x)}{dx} v(x) \right]_0^1 = 0,
$$

(1.72)

$$
\int_0^1 \frac{du(x)}{dx} \frac{dv(x)}{dx} \, dx + \frac{du(1)}{dx} v(1) - \frac{du(0)}{dx} v(0) = 0,
$$

(1.73)

$$
\int_0^1 \frac{du(x)}{dx} \frac{dv(x)}{dx} \, dx + v(1) - \frac{du(0)}{dx} = 0.
$$

(1.74)

We end up with the following weak form:

Find $u \in V = \{ u \in H^1(0,1) : u(0) = 0 \}$ such that

$$
b(v, u) = l(v), \forall v \in V,
$$

(1.75)

where $b(u, v) = \int_0^1 \frac{dv(x)}{dx} \frac{du(x)}{dx} \, dx$.

(1.76)

(1.77)

(1.78)

In the case of the one- finite element method, we partition the domain $[0,1]$ into a set of intervals called finite elements $E_i$:

$$
E_i = [\xi_{i-1}, \xi_i] = \left[ \frac{i-1}{N}, \frac{i}{N} \right], i = 1, \ldots, N.
$$

(1.79)

---

The labs for this chapter are accompanied with JAVA codes which can be downloaded from http://www.ki.agh.edu.pl/FastSolvers/Chapter1_3