A Relaxed Optimization Theory for Structured Stencil-based Parallel Applications

Massimiliano Meneghin

Supervisor
Marco Vanneschi

10 January 2010
Abstract

IN THIS THESIS we introduce a new optimization theory for stencil based applications which is centered on a relaxed definition of computational equivalence. The proposed optimization techniques provide notable results on different computational aspects: from the reduction of communication overhead to the reduction of the computation time, passing though the minimization of memory requirement without performance lost.

All the classical optimization theory is based on defining transformations that can produce optimized programs which are computational equivalent with respect to original ones. According to Kennedy, two programs are equivalent if from the same input data they produce identical output data.

We propose a innovative approach that completely drifts away from the classical one. Indeed, we base our new optimization theory on the concept of relaxed computational equivalence. Two programs are relaxed equivalent if from the same input data they produce identical output values which however can be differently organized in their output data structure space.

Exploiting a structured approach and studying this new optimization theory on stencils featuring specific patterns of functional dependencies, we discover a set of novel transformations which result in significant optimizations.

Among all the new transformations, the most notable one, which aims the reduction of the number of communications necessary to implement a stencil based application, results to be the best optimization technique with respect to the ones cited in literature.

All the improvements provided by the transformations that are presented in this thesis have been both formally proved and experimentally tested on cluster and multi-core architectures.
The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

Notes on Structured Programming,

Edsger Dijkstra
Contents

1 Introduction
   1.1 Contributes and Motivation of the Thesis .................................. 1
   1.2 Classical Optimization Theory ............................................... 2
   1.2.1 The Concept of Computational Equivalence ............................ 5
   1.2.2 The Mechanisms of Data Dependency ................................. 5
   1.2.3 Most Important Classic Transformations Based on Data Dependency .................................................................................. 6
   1.2.4 Optimizations not Based on Data Dependency ...................... 8
   1.3 Structured Parallel Programming .......................................... 9
   1.4 Plan of the Thesis ................................................................. 12

2 The Structured Stencil Model ...................................................... 15
   2.1 A Formalization of Stencils Through a Structured Vision ............. 17
   2.1.1 An Informal Analysis of a Generic Stencil .......................... 17
   2.1.2 The Formalization of the Structured Stencil Model ............... 22
   2.1.3 Considerations on the “Structured” Feature ...................... 29
   2.2 Tutorial Examples ............................................................... 30
   2.2.1 Laplace ........................................................................... 30
   2.2.2 Red-Black ................................................................. 31
   2.2.3 Floyd-Warshall ............................................................ 39
   2.3 A Classification of Space Invariant Stencils ............................ 41
   2.3.1 The Equivalence Relation Between Stencil Components .......... 41
   2.3.2 A Classification Based on the Step Set Component .............. 42
   2.3.3 A Classification Based on the Shape Set Component ............. 44
   2.3.4 A Classification Based on Relations Between Application Point and Shape ................................................................ 45
   2.3.5 A Classification Based on the Relations Between Shapes ....... 46
   2.3.6 The Semi Uniform Stencil Class ...................................... 47
   2.3.7 The Space Invariant Stencil Family: The Big Picture .......... 53
   2.3.8 The Boundary Problems .................................................. 55
   2.4 An Extension of Space Invariant Stencils ............................... 61
   2.5 A Specification of the Structured Model for \mathcal{HUA} Stencils ....... 64
   2.5.1 The Definition of the \mathcal{HUA} Model .............................. 64
## 3 The $\mathcal{HUA}$ Complete Architecture

3.1 The Reference Architecture ............................................. 73
3.2 The World of Partitions ..................................................... 75
  3.2.1 Working Hypotheses on Partitioning Strategies .................... 75
  3.2.2 Conventions and Annotations for Partitions ....................... 76
  3.2.3 Regions: the Classification of Partition Elements ................. 78
3.3 The World of Concurrency .................................................. 83
  3.3.1 The $\mathcal{LC}$ language ................................................. 83
  3.3.2 Communication Cost Model ........................................... 84
  3.3.3 $\mathcal{LC}$ and $\mathcal{HUA}$ Stencils ................................ 84
3.4 A Nine Point Stencil at Work ............................................ 86
  3.4.1 Functional Dependency Level ....................................... 86
  3.4.2 Partition Dependency Level ........................................ 87
  3.4.3 Concurrent Level: The $\text{naive}$ Method ......................... 89
  3.4.4 Concurrent Level: The $\text{shift}$ Method ......................... 92
  3.4.5 Environments of the Experimental Tests .......................... 95
  3.4.6 Experimental Results ................................................ 96
  3.4.7 Conclusions .......................................................... 102

## 4 $\mathcal{Q}^{-}$ transformations

4.1 $\mathcal{Q}^{-}$ transformations ............................................... 105
  4.1.1 Defining $\mathcal{Q}^{-}$ transformations ............................. 105
  4.1.2 $\mathcal{Q}^{-}$ transformations are different from Skewing ........ 107
4.2 Positive $\mathcal{Q}^{-}$ transformations ................................... 110
  4.2.1 Defining $\mathcal{Q}^{+}^{-}$ transformations ......................... 110
4.3 A Nine Point Stencil at Work with $\mathcal{Q}^{+}^{-}$ transformations .... 113
  4.3.1 Functional Dependency Level ....................................... 113
  4.3.2 Partition Dependency Level ........................................ 113
  4.3.3 Concurrent Level: The $q$ Method ................................ 115
  4.3.4 Concurrent Level: The $q\_\text{shift}$ method ..................... 119
  4.3.5 Experimental Results ................................................ 121
  4.3.6 Conclusions .......................................................... 128
4.4 A Closer Analyses on $\mathcal{Q}^{+}^{-}$ transformations ................ 129
  4.4.1 Analytic Analysis of the Nine Point Stencil ..................... 129
  4.4.2 The Jacobi Case ...................................................... 129
4.5 Negative $\mathcal{Q}^{-}$ transformations ................................ 134
  4.5.1 Defining $\mathcal{Q}^{-}$ transformations ............................. 134
  4.5.2 Combining Positive and Negative $\mathcal{Q}^{-}$ transformations ... 137
4.6 Extending $\mathcal{Q}^{-}$ transformations to The Semi Uniform ........ 138
4.7 Conclusions .............................................................. 140
5 Step Fusion Transformations 141
5.1 Data Replication in HUA Stencil 143
5.1.1 The Oversending Method 143
5.1.2 Oversending Performance Model 145
5.1.3 Oversending and $Q-trans formations$ 149
5.2 Step Fusion Transformations 152
5.2.1 A Structured Interpretation of Oversending 152
5.2.2 Formal Definition of $SF$ Transformations and Their Properties 154
5.2.3 Step Fusion for Linear Step Functions 158
5.2.4 $SF$ and Oversending 158
5.3 $SF$ and Sequential Computations 161
5.3.1 Relation between Shape Cardinality and $SF$ Level 161
5.3.2 Temporal Locality Factor in $SF-trans formations$ 162
5.3.3 Asymptotic Analysis of both Computations and Communications 163
5.3.4 Taking into account Cache Memory Hierarchy 168
5.3.5 Experimental Results 169
5.3.6 Conclusions 174

6 Space Overlapping Transformations 177
6.1 The Implementation of the Working Domain 179
6.1.1 Naive Implementation 179
6.1.2 Support Buffer Implementation 180
6.1.3 Space Overlapping Implementation 182
6.2 $QM-trans formations$ 187
6.2.1 Positive $QM-trans formation$ 187
6.2.2 Negative $QM-trans formation$ 191
6.2.3 Performance Tests 193

7 Conclusions 201
A MammuT 203
A.1 Structured Parallel Programming 206
A.2 $LLC$ Language Semantics and Syntax 209
A.2.1 $LLC$ Channel API 209
A.2.2 $MPI$ and $LLC$ 210
A.3 $LLC$ Channel Abstract Protocol 212
A.4 Channel Abstract Optimization 214
A.4.1 Static Refilling: the $w_protocol$ 214
A.4.2 The $K_plus_one$ Optimization 216
A.5 Concrete Implementation on the Cell 218
A.5.1 The Cell Architecture 218
A.5.2 Channel Implementation on Cell 218
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.5.3 Signal-based Implementation</td>
<td>222</td>
</tr>
<tr>
<td>A.5.4 DMA1 Implementation</td>
<td>222</td>
</tr>
<tr>
<td>A.5.5 DMA2 Implementation</td>
<td>223</td>
</tr>
<tr>
<td>A.6 The Cost Model</td>
<td>225</td>
</tr>
<tr>
<td>A.7 Conclusion and Feature Works</td>
<td>227</td>
</tr>
<tr>
<td>Bibliography</td>
<td>229</td>
</tr>
<tr>
<td>Index</td>
<td>235</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Representations, in a one dimension space, of both Jacobi stencil (fig. 2.21(a)) and its modified version (fig. 2.21(b)), where the application point has been shifted of one position. Example of one step computation of the original (fig. 2.21(a)) and the modified Jacobi (fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (fig. 2.21(e)) and modified (fig. 2.21(f)) Jacobi to highlight the different communication patterns.

2.1 Description in pseudo code of the Jacobi stencil application on a two dimensional toroidal space. To keep a light annotation we suppose the indexes are automatically re-mapper into the toroidal space, i.e. $J_{out}[-1,-1]$ matrix access is transformed into $J_{out}[9,9]$.

2.2 Representation of an application point and the corresponding shape which are featured by the Jacobi application whose description in pseudo code is reported in Figure 2.1.

2.3 Description in pseudo code of the Gauss-Seidel stencil application on a two dimensional space.

2.4 Representation of time instants of the structured stencil model. Evaluation of an element of a spatial structure are legal only between steps, which are considered as a atomic operations.

2.5 Representation of an application point and the corresponding shape which are featured by the Laplace application whose description in pseudo code is reported in Figure 2.6.

2.6 Representation in pseudo code of a Laplace stencil application on a two dimensional toroidal space. To keep the annotation light we suppose the indexes are automatically mapped on the toroidal space, i.e. $J_{out}[-1,+1]$ is transformed into $J_{out}[+1023,+1]$.

2.7 Representation in pseudo code of a Red-Black stencil application on a two dimensional toroidal space. To keep a light annotation we suppose the indexes are automatically re-mapped into the toroidal space, i.e. for example $J_{out}[-1,-1]$ matrix access is transformed into $J_{out}[99,99]$.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8</td>
<td>Representation of two application points and the corresponding shape which are featured by Red step of the Black-Red application whose description in pseudo code is reported in Figure 2.7.</td>
</tr>
<tr>
<td>2.9</td>
<td>Representation of two application points and the corresponding shapes which are featured by Black step of the Black-Red application whose description in pseudo code is reported in Figure 2.7.</td>
</tr>
<tr>
<td>2.10</td>
<td>Description in pseudo code of a Floyd-Warshall stencil application on a two dimensional space.</td>
</tr>
<tr>
<td>2.11</td>
<td>Representation of two application points and the corresponding shape which are featured, during the iteration \textit{step} \textsubscript{1} (Figure 2.11(a)) and \textit{step} \textsubscript{4} (Figure 2.11(a)), by the Floyd-Warshall application whose description in pseudo code is reported in Figure 2.10.</td>
</tr>
<tr>
<td>2.12</td>
<td>Stencil classification in the structured model based on the relations between the elements in the step set.</td>
</tr>
<tr>
<td>2.13</td>
<td>Classification of space invariant affine stencils.</td>
</tr>
<tr>
<td>2.14</td>
<td>Representation of two application points and the corresponding shapes which are featured by Red step of extended version of the Black-Red application whose description in pseudo code is reported in Figure 2.7. The picture can be compared with the original one reported in Figure 2.8.</td>
</tr>
<tr>
<td>2.15</td>
<td>Representation of two application points and the corresponding shapes which are featured by Black step of extended version of the Black-Red application whose description in pseudo code is reported in Figure 2.7. The picture can be compared with the original one reported in Figure 2.9.</td>
</tr>
<tr>
<td>2.16</td>
<td>Representation of HUA space invariant stencil with respect to all presented stencil classification.</td>
</tr>
<tr>
<td>2.17</td>
<td>Representation in pseudo code of a Jacobi stencil application on a one dimensional non toroidal space.</td>
</tr>
<tr>
<td>2.18</td>
<td>Representation of some application points and the corresponding shapes which are featured by the one dimension non toroidal Jacobi application whose description in pseudo code is reported in Figure 2.17.</td>
</tr>
<tr>
<td>2.19</td>
<td>Representation of some application points and the corresponding shapes which are featured by the \textbf{extended} one dimension non toroidal Jacobi application whose description in pseudo code is reported in Figure 2.17.</td>
</tr>
<tr>
<td>2.20</td>
<td>Representation of some application points and the corresponding shapes which are featured, during different steps, by a reduce application.</td>
</tr>
</tbody>
</table>
2.21 Representations, in a one dimension space, of both Jacobi stencil (fig. 2.21(a)) and its modified version (fig. 2.21(b)), where the application point has been shifted of one position. Example of one step computation of the original (fig. 2.21(a)) and the modified Jacobi (fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (fig. 2.21(c)) and modified (fig. 2.21(f)) Jacobi to highlight the different communication patterns.

3.1 Representation of our reference architecture for implementing HUA stencils.

3.2 Representation of the partition reference system for a two dimensional working domain.

3.3 Graphical representation of stencil shape (3.3(a)), incoming dependent (in gray color) and independent regions (3.3(b)) and finally outgoing dependent (in gray color) and independent regions (3.3(c)).

3.4 A representation of our reference architecture for implementing HUA stencils that highlights for each level the main mechanisms.

3.5 Representation in pseudo code of a nine point stencil (Nine) based application on a two dimensional toroidal space. To keep the annotation light we suppose the indexes are automatically mapped on the toroidal space, i.e.\( J_{\text{out}}[-1,+1] \) is transformed into \( J_{\text{out}}[+1023,+1] \).

3.6 Graphical representation of stencil shape (3.6(a)), incoming dependent (in gray color) and independent regions (3.6(b)) and finally outgoing dependent (in gray color) and independent regions (3.6(c)) of a nine point stencil.

3.7 Representation in pseudo code of a nine point stencil application described at the concurrent level exploiting the naive method.

3.8 Graphical representation of the incoming (fig. 3.8(a)) and outgoing (fig. 3.8(b)) communication of a naive implementation of the nine point stencil at the concurrent level.

3.9 Representation in pseudo code a nine point stencil application described at the concurrent level exploiting the shift method.

3.10 Graphical representation of the incoming (fig. 3.10(a)) and outgoing (fig. 3.10(b)) communication of a shift implementation of the nine point stencil at the concurrent level.

3.11 Communication overheads featured by naive and shift implementations of the nine point stencil in a two dimension space performed on dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.

3.12 Communication overheads featured by naive and shift implementations of the twenty-seven point stencil in a three dimension space performed on dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.
3.13 Communication overheads featured by naive and shift implementations of the nine point stencil in a two dimension space performed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.

3.14 Communication overheads featured by naive and shift implementations of the twenty-seven point stencil in a three dimension space performed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.

3.15 Communication overheads featured by naive and shift implementations of the twenty-seven point stencil in a three dimension space performed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.

4.1 Pseudo code of one iteration of the Gauss-Seidel stencil on a two dimensional space. The stencil features \{(1,1), (0,1)\} as dependency vector and \(\mathcal{R} = \{(0,1)(0,-1)(1,0)(-1,0)\}\) as shape (consider an extension of the shape element to non \(\mathcal{H}UA\) stencil).

4.2 Pseudo code of one iteration of the skewed Gauss-Seidel stencil on a two dimensional space. The skewed stencil features \{(1,1), (0,1)\} as dependency vector and \(\mathcal{R} = \{(0,1)(0,-1)(1,0)(-1,0)\}\) as shape (consider an extension of the shape element to non \(\mathcal{H}UA\) stencil).

4.3 Graphical representations of the relative shapes of respectively Jacobi and \(Q^+[Jacobi]\) stencils.

4.4 Jacobi pseudo code.

4.5 \(Q^+[Jacobi]\) pseudo code.

4.6 Graphical representation of stencil shape (4.6(a)), incoming dependent (in gray color) and independent regions (4.6(b)) and finally outgoing dependent (in gray color) and independent regions (4.6(c)) for the \(Q^+[nine]\) stencil.

4.7 Representation in pseudo code a \(Q^[nine]\) stencil described at the concurrent level exploiting the \(q\) method.

4.8 Graphical representation of the incoming (fig. 4.8(a)) and outgoing (fig. 4.8(b)) communications of a \(q\) implementation of the nine point stencil at the concurrent level.

4.9 Representation in pseudo code a \(Q^[nine]\) stencil described at the concurrent level exploiting the \(q-shift\) method.

4.10 Graphical representation of the incoming (fig. 4.10(a)) and outgoing (fig. 4.10(b)) communications of a \(q-shift\) implementation of the nine point stencil at the concurrent level.

4.11 Number of communications per single step exploiting different methods.

4.12 Performance result of a nine point stencil in a two dimension space, executed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Performance result of twenty-seven point stencil in a three dimension space, executed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.</td>
</tr>
<tr>
<td>4.14</td>
<td>Nine point stencil in a two dimension space, performed on Cell B.E. IBM multicore exploiting the MammuT implementation of LC.</td>
</tr>
<tr>
<td>4.15</td>
<td>Performance result of a nine point stencil in a two dimension space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.</td>
</tr>
<tr>
<td>4.16</td>
<td>Performance result of a twenty-seven point stencil in a three dimension space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.</td>
</tr>
<tr>
<td>4.17</td>
<td>Shape 4.17(a) of the Jacobi stencil. Incoming (fig. 4.17(b)) and outgoing (fig. 4.17(c)) communications of a naive implementation at the concurrent level.</td>
</tr>
<tr>
<td>4.18</td>
<td>Shape 4.18(c) of the $Q^+[Jacobi]$ stencil. Incoming (fig. 4.18(d)) and outgoing (fig. 4.18(e)) communications of a $q$ implementation at the concurrent level.</td>
</tr>
<tr>
<td>4.19</td>
<td>Jacobi stencil in a two dimension space, performed on Cell B.E. IBM multi-core exploiting the MammuT implementation of LC.</td>
</tr>
<tr>
<td>4.20</td>
<td>Evolution of the element value positions in the spatial structure in the case of a Jacobi stencil defined over a non toroidal domain space: 4.24(a) with positive $Q-transformation$, 4.24(b) with the interleaving of positive and negative $Q-transformation$.</td>
</tr>
<tr>
<td>5.1</td>
<td>Representation in pseudo code of a Jacobi stencil implemented at the concurrent level, exploiting a naive method and a row partitioning strategy.</td>
</tr>
<tr>
<td>5.2</td>
<td>Representation in pseudo code of a Jacobi stencil implemented at the concurrent level, exploiting a oversending method and a row partitioning strategy.</td>
</tr>
</tbody>
</table>
5.3 Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized with the oversending method according to the pseudo code in Figure 5.2. .......................................................... 147
5.4 Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized with the q-oversending method. .......................... 148
5.5 Graphical representation of the communication patterns of naive, q, oversending and q-oversending methods exploited to implement a Jacobi stencil in a row partition configuration. ................................. 150
5.6 Graphical representation of the step fusion transformations which is applied to a Jacobi stencil. .......................................................... 153
5.7 Graphical representation of $SF^i$[Laplace] stencils for i from one to three. .......................................................... 156
5.8 Graphical representation of $SF^i$[nine] stencils for i from one to three. 156
5.9 Graphical representations of shapes and communication patterns of $Jacobi, SF^2[Jacobi], Q[Jacobi], QSF^2[Jacobi]$ stencils. .............. 159
5.10 Element Increasing Factor for Jacobi, Laplace and Nine. ............... 162
5.11 Graphical representation of time locality for Jacobi and $SF[Jacobi]$ stencils .......................................................... 163
5.12 Graphical representation of time locality for different stencils ......... 164
5.13 Performance trend for $SF-transformations$ for Jacobi 5.13(a) and Laplace 5.13(b) .......................................................... 165
5.14 Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB .......................................................... 170
5.15 Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB ............... 171
5.16 Intel(R) Xeon(R) CPU 5150 @ 2.66GHz cache size 4096 KB .......... 172
5.17 Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB ...... 173
5.18 Element Increasing Factor for Jacobi, Laplace and Nine extended in a three dimension space. .......................................................... 175
6.1 Description in pseudo code of a naive implementation of the Jacobi stencil on a toroidal space. .......................................................... 180
6.2 Description in pseudo code of a buffered implementation of the Jacobi stencil on a toroidal space. .......................................................... 181
6.3 Description in pseudo code of a implementation with space overlapping of the Jacobi stencil defined over a toroidal space. ............... 183
6.4 Mapping between virtual and real vectors for the overlapping implementation .......................................................... 184
6.5 Jacobi, $Q^+[Jacobi], Q^-[Jacobi]$ shapes. .......................................................... 185
6.6 Graphical representations of shapes and and application points of a set of stencils trasformed with $QM-transformations$. ............... 189
6.7 Visit pattern forced by the in-place computation .......................................................... 190
6.8 Visit pattern forced by the in-place computation. .......................... 192
6.9 Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB ................................................................. 195
6.10 Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB ................................................................. 196
6.11 Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB ............... 197
6.12 Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB ............... 198
6.13 Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB ......... 199
6.14 Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB ......... 200
A.1 System structure ................................................................. 207
A.2 Pseudo code and data structure used by the $LLC$ channel abstract protocol ................................................................. 212
A.3 Data structured used by the $LLC$ abstract protocol $w$ _protocol ........ 214
A.4 Pseudo code of the $LLC$ abstract protocol $w$ _protocol ............... 214
A.5 Data structured used by the $K$ _plus_one optimization ................. 216
A.6 Pseudo code of the $K$ _plus_one optimization ........................... 216
A.7 Cell $LLC$ channel data structures A.7(a) and its concrete protocol compared with the abstract one (A.8(a)) ......................... 219
A.8 Cell $LLC$ channel data structures A.7(a) and its concrete protocol compared with the abstract one (A.8(a)) ......................... 220
A.9 Comparison of latency and bandwidth performance of different $LLC$ channel implementation and of DMA transfer ................. 224
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Step component in the structured stencil model of the Jacobi stencil application that is described in pseudo code in Figure 2.1</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Structured model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6</td>
<td>32</td>
</tr>
<tr>
<td>2.3</td>
<td>Structured Step model of the Red steps of the Red-Black (RB) stencil application described Figure 2.7</td>
<td>34</td>
</tr>
<tr>
<td>2.4</td>
<td>Structured Step model of the Black steps of the Red-Black (RB) stencil application described Figure 2.7</td>
<td>35</td>
</tr>
<tr>
<td>2.5</td>
<td>Structures stencil model of the Red-Black stencil application described in pseudo code in Figure 2.7. The step model of Red and Black steps are reported respectively in Table 2.3 and Table 2.4</td>
<td>37</td>
</tr>
<tr>
<td>2.6</td>
<td>Structured model of the Floyd-Warshall (FW) stencil application described in pseudo code in Figure: 2.10</td>
<td>40</td>
</tr>
<tr>
<td>2.7</td>
<td>Structured Step model of the Red steps of the extended Red-Black (RB) stencil application described Figure 2.7</td>
<td>50</td>
</tr>
<tr>
<td>2.8</td>
<td>Structured Step model of the Black steps of the extended Red-Black (RB) stencil application described Figure 2.7</td>
<td>51</td>
</tr>
<tr>
<td>2.9</td>
<td>Structured Step model of the 1D non toroidal Jacobi stencil (JCB) described in the pseudo code of Figure 2.17</td>
<td>59</td>
</tr>
<tr>
<td>2.10</td>
<td>Structured Step model of the extended 1D non toroidal Jacobi stencil (JCB) described in the pseudo code of Figure 2.17</td>
<td>60</td>
</tr>
<tr>
<td>2.11</td>
<td>Step component of a reduce application in the structured stencil model.</td>
<td>63</td>
</tr>
<tr>
<td>2.12</td>
<td>HUA model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6</td>
<td>66</td>
</tr>
<tr>
<td>2.13</td>
<td>HUA model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6</td>
<td>69</td>
</tr>
<tr>
<td>2.14</td>
<td>HUA model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6</td>
<td>69</td>
</tr>
<tr>
<td>3.1</td>
<td>Step component in the structured stencil model of the nine point stencil application that is described in pseudo code in Figure 3.5</td>
<td>89</td>
</tr>
<tr>
<td>4.1</td>
<td>Step component in the structured stencil model of the $Q^+[nine]$ stencil application.</td>
<td>114</td>
</tr>
</tbody>
</table>
5.1 Performance model of the communications and computations of naive, q, oversending and q-oversending methods exploited to implement a Jacobi stencil in a row partition configuration. The parameter are considered as mean time per step .................................................. 150
5.2 Number of elements of the stencils resulted by the applications of different $SF$-transformations to Jacobi, Nine and Laplace ........ 161
5.3 Number of elements of the stencils resulted by the applications of different $SF$-transformations to the three dimension extension of Jacobi, Nine and Laplace ................................................. 174
Chapter 1

Introduction

Data parallelism is a well known paradigm of parallel programming, which is characterized by replication of functions and partitioning of data over a set of virtual processing nodes.

One of the most powerful data parallel paradigm is represented by the stencil one. The class of stencil-based applications arises in many scientific fields. Stencils are exploited in explicit time-integration methods for numerical solution of partial differential equations, for example in climate, weather, ocean modelling [42, 2, 25, 44] or in Finite Difference Time Domain methods for computational electromagnetic [46], and for multimedia and image processing applications [49, 20].

A stencil-based data parallel application is characterized by a certain number of iteration steps; at each step the processing nodes calculate a new value for all the elements of the partitioned domain, complying some functional dependencies over sets of elements. Two of the main issues on stencil-based applications are communications and caching. In parallel architectures, especially targeting a fine grain parallelization, in order to lower the computation time, the communications, which are required to resolve data dependencies between different domain partitions, can represent an unavoidable lower bound [41, 15, 39]. The other important issue is about extracting features from stencil computational kernel that can be exploited to take full advantages of memory hierarchies, especially those of the new multi-core architectures [43, 40, 19, 22, 13, 28].

Generally speaking, a model is a tool exploited to manage a representation of the object of study. The action of modelling is equivalent to defining a set of bonds in order to outline the peculiar characteristics of the object. The bounds are exploited to search model properties that can be useful for the study. The more the bounds are strict the more the features of the object are highlighted, but the less general the model is.

In the classic theory of program optimization, the objects of study are program transformations that can produce equivalent programs featuring different characteristics with respect to the original one; for example higher performance, or lower memory requirements, or hopefully both of them.
The bound that is mainly exploited by the classic optimization model is the computational equivalence concept. This definition is a critical aspect because it is the yardstick by which we evaluate the legality or the safeness of a program transformation. Computational equivalence given by Allen and Kennedy [23] states that two computations are equivalent if, on the same output, they produce identical values for the output variables at the time the output statements are executed, and the output statements are executed in the same order.

All the theory of dependence analysis, which is at the core of a huge and most important class of optimizations for parallelism and cache management, is based on a fundamental property of the classic optimization model. This property, which represents a sufficient condition for equivalence, asserts that a transformation is classified as legal when it changes only the execution order of the code while preserving every dependencies.

1.1 Contributes and Motivation of the Thesis

In this thesis we introduce a new optimization theory which is centered on a relaxed definition of computational equivalence. The proposed optimization techniques provide notable results on different computational aspects: from the reduction of communication overhead to the reduction of the computation time, passing though the minimization of memory requirements without performance lost.

Our study deviates from the classic approach because of a different concept of computational equivalence. We develop an optimization theory targeting exclusively stencil-based applications featuring specific patterns of functional dependencies. This further bound, which is not present in the classic approach, allows us to lower the strictness of the computational equivalence constrain. We therefore define a ‘relaxed computational equivalence’, which classifies a transformation as relaxed legal when it is legal in the classic assertion, except for a spatial rearrangement of the values in the output data structures.

To give a clue about the cited spatial rearrangement, we consider the Jacobi stencil in a one dimension space; a well known method for partial differential equations. The stencil, represented in figure 2.21(a), works updating at step $i$ the values of all the domain elements with the mean of their left and right element values at the previous step $i - 1$. We call the left and the right elements the stencil shape (colored in gray) while the central element (with a dashed fill style) the application point. In fig. 2.21(a) $J_{in}$ represents the working domain values at step $i$ while $J_{in}$ at step $i + 1$.

Along with the “original Jacobi”, we consider also a “modified Jacobi”, which is represented in fig. 2.21(b); it has the same function of the original stencil and also the same shape, except for the application point location which is shifted on the right of one position.

Let now $L_{in} = \{2, 4, 8, 16, 24, 32, 64\}$ be a vector, representing a toroidal input
1.1. CONTRIBUTES AND MOTIVATION OF THE THESIS

Figure 1.1: Representations, in a one dimension space, of both Jacobi stencil (fig. 2.21(a)) and its modified version (fig. 2.21(b)), where the application point has been shifted of one position. Example of one step computation of the original (fig. 2.21(a)) and the modified Jacobi (fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (fig. 2.21(e)) and modified (fig. 2.21(f)) Jacobi to highlight the different communication patterns.

Neglecting the different results in a first approximation, we consider a parallel implementation of both the previous stencils, where the input and output vectors are scattered over four processing nodes as presented in fig. 2.21(e) and 2.21(f).

The key point is now to analyze how the two stencil functional dependencies differently impact on communications. One single node, in the case of the original Jacobi, features incoming and outgoing communications with both left and right neighbours. In the case of the modified Jacobi, because of the different application
CHAPTER 1. INTRODUCTION

point, one resource receives data only from the right neighbour and sends data only to the left one.

In conclusion, although the flow of exchanged data is equal in both cases, i.e. the amount of elements sent or received does not change, the modified Jacobi halves the number of incoming and outgoing communications per step, which has a direct impact onto the computation time, especially for fine grain parallelizations.

Analyzing the example with the classic optimization theory, a transformation that returns the modified Jacobi from the original one is not legal; but according to our new theory we can assert it is classified as relaxed legal. In fact, as previously noted, a linear transformation is sufficient to describe how to reconstruct the output of the original Jacobi moving left or right the elements of the other one.

Exploiting a structured approach and studying this new optimization theory on stencils featuring specific patterns of functional dependencies, we discover a set of novel transformations which result in significant optimizations for a wide class of real world stencils:

- $Q -$ transformations provide optimizations on the communication overhead, exploiting the relaxed computational equivalence and breaking the limitations of the well known owner-compute rule. With the presented transformations the number of communications required to implement a generic stencil, defined over a $n$ dimension space, can be reduced to $n$, with respect to $2 \times n$ which is the best result with respect to the solutions cited in literature [41]. We also study mechanisms to reduce the overhead of taking the output working domain back to its original space ordering. In some cases we can even provide a zero reordering overhead, still preserving the reduction of communication overhead.

Experimental tests on clusters and multi-core architectures prove that $Q -$ transformations provide better performance than other implementations. In a worst case where communication with all neighbours are required, the reduction of the communication overhead can be quantified with respect to a “naive” implementation in a value of time gain that is up to 4.5; we define the time gain parameter as the ratio between the completion time of a reference implementation, i.e. the “naive”, and a studied one, i.e. the implementation optimized with $Q -$ transformations.

- $QSF -$ transformations are a formalization and a notable extension of those techniques which exploit replication of data to reduce communications [15]. One side effect of studying $QSF -$ transformations is the definition of politics for the memory hierarchy management in stencil computational kernel. The technique, which is based on a revisitation of loop fusion classic optimizations, provides a value of time gain up to 2.1, in a sequential environment.

- $QM -$ transformations are optimizations targeting the reduction of memory constraints. This reduction is obtained without performance lost; the
type of memory accesses defined by QM–transformations rather provide
notable performance benefit. By experimental results, we prove that QM–
transformations almost halve the memory constraints and in some cases pro-
vide a value of time gain up to 2.2.

1.2 Classical Optimization Theory

As claimed by Aho et al. [3], in order to target parallelism mechanisms are required,
that help to reason about the dependencies among different dynamic executions
of the same statement to determine if they can executed on different processors
simultaneously.

In the classical theory of optimisation the field of data dependence analysis pro-
vides the previous tools. Exploiting technique based on data dependency is possible
to transform an input program into a computational equivalence that feature higher
performance. The earlier studies of data dependence for vectorization porpoise were
by Lamport [34, 35] in parallel with Kuck, Muraoka and Chen [30, 38].

Other optimizations, along with the technique based on data dependence anal-
ysis, exist such as ghost expansion and shift methods. Nevertheless they all share
the same base concept: the transformation of a program results always in a
computational equivalence one.

It is the aim of this Section to present firstly the concept of the computational
equivalence and then to introduce some of the most known transformations derived
by the classical optimization theory.

1.2.1 The Concept of Computational Equivalence

A program transformation has the goal of defining a new program that, while fea-
turing the same “meaning” of the original one, provides different characteristics; we
are obviously interested especially in better performance. In other words the trans-
formed program has to do the same things of the original one but in a different
way.

This rises the question of what behaviours the transformed program has to pre-
served. If the aim is to preserve the same run time, the transformed program can
not feature different performance with respect to the original one, hence we would
no be interested in program transformation any more. What has to be preserved is
the effects of the program.

Bacon et. al. [7] well present and argument the logical path to the definition of
a criterion of equivalence between programs. They start from the following simple
definition:

A transformation is legal if the original and the transformed programs
produce exactly the same output for identical executions.
With some examples, the authors keep digging into the problem and introduce the issue regarding the relation between program transformations and correctness. The problem is to manage the cases where the original program is not semantically correct.

The requirement for the transformed program of producing the same results would be an unnecessary bound when the original program is semantically incorrect. The authors therefore introduce the semantic correctness of the original program as a unavoidable condition in the computational equivalence definition.

Finally Bacon et. al. analyze the issue of floating point operations. Because of the finite precision of the floating point representations, the request of bitwise identical results can be too strict. Therefore they consider in the equivalence definition a certain degree of tolerance with respect to floating point operations.

We reported this discussion about the concept of the computational equivalence because it represents the break point between the classical theory and the new one that we are presenting within this thesis.

The classic approach is based on a model where a transformation is safe if the output data structures of a semantic correct program and the transformed one are exactly the same a part from a tolerance for floating point values. Considering the output data structures of the original and the transformed program, we have that according to the classical approach in a given position both data structures stores the same value: the output values feature the same spatial organization in the output data structures of the two programs.

Our approach drifts away from the basis of the classic stream. Indeed we target a relaxed type of computational equivalence: we require that the data structures of the two programs stores the same values but not necessary with the same spatial organization.

1.2.2 The Mechanisms of Data Dependency

One of the most important stream in the classical optimization theory is the data dependence analysis. A data dependence is a relation between two statements that occurs if and only if

I both statement access the same memory location and at least one of them stores into it, and

II there is a feasible run-time execution path from one statement to the other.

All the data dependence analysis is based on fundamental Theorem of Dependence of which we report the terms.

Any reordering transformation that preserves every dependence in the program preserve the meaning of the program.
1.2. CLASSICAL OPTIMIZATION THEORY

A reordering transformation is defined as program transformation that merely changes the execution order of the code, without adding or deleting executions of any statements.

The data dependence analysis mainly studies how to define transformations that, while reordering the program execution and keeping the data dependency preserved, can produce programs featuring better performance with respect to the original ones. Obviously to prove the correctness of those transformations a set of mechanisms to model data dependencies are necessary.

The mechanism exploited for the representation of the dependence information of a piece of code is dependence graph, where each node correspond to a statement and arches indicates the existence of a dependence between two statements.

Usually the dependence graph is exploited to model straight code that does not feature iterations constructs. In the analysis of loops, which is a more complicated and interesting problem, other specific mechanisms are used.

When facing loop iterations, each statement inside a loop can be executed many times and therefore is necessary to describe dependencies that can rise between different loop iterations. Those kind of dependencies is called loop-carried dependencies.

In order to model this more complex situations, Kuck [31] and Wolf [51] studied two mechanisms: respectively distance and direction vectors. The first models the dependency distances for all the iteration, where the distance represent, in term of iterations index, the computations between the execution of two statements that are linked by a dependency.

In same cases it is not possible to define at compile time the exact dependency distance or it is possible that the distance is not a constant. In those situations, the direction vectors are commonly used to partially characterized the dependencies.

The presented mechanisms are exploited to define and prove most of all the known transformation of the classical optimization theory based on data dependencies. In the following we are going to review the most important ones.

Before proceeding, we want to point out an issue of the dependencies analysis: the problem of automatically detect when two arrays reference might refer to the same element in different iterations. In analyzing a loop nest, usually a compiler tries some tests, which relay in the fact that expressions are almost linear. When dependencies are found, the compiler models them with distance or direction vectors. There are a large variety of tests, all of them aim to prove the independence in some cases. Indeed the direct problem of finding dependencies is a NP-complete problem [8].
1.2.3 Most Important Classic Transformations Based on Data Dependency

Loop Skewing

Loop skewing [34, 51] is a transformation that reshapes the iteration space to highlight existing parallelisms. The transformation changes the visit pattern in such a way that more operations can be computed in parallel. Skewing was invented to handle wavefront computations, where the updates of an array propagate like a wave across the iteration space.

Loop Tiling or Blocking

Stencil computation feature global iterations through a data structure that is typically much larger than the capacity of the available data cache. Reorganization to take full advantage of the memory hierarchies has been studied. Those investigation has principally focused on tiling techniques [1, 16, 33, 50] which attempt to exploit locality by performing operations on cache-sized blocks of data before moving to the next block. Tilling optimizations are also known as blocking techniques.

Loop Fusion

A well known transformation aiming the modification of the computation grain inside a loop is the loop fusion [52] or also known as jamming. The main concept of the transformation is replacing multiple loops with a single one containing all the statement. This optimization reduce the overhead of the loop management and can increase both parallelism and data locality.

1.2.4 Optimizations not Based on Data Dependency

We report some important optimizations for stencil-based applications. Such transformations does not come from the dependency theory but are anyway based on the same computational equivalence concept.

Message shifting

Plimpton [41] first presented a technique, which we refer to as shift method, to reduce communications with diagonal neighbours in stencil computations. The shift method cuts down the total number of sent messages per step from $3^n - 1$, which is the maximum number of neighbours in a $n$ dimension space, to $2 \times n$.

The technique is based on indirect communications with the diagonal neighbours. A single step can be seen as composed of a set of micro-steps, one for each space dimension. Differently from a naive method, the shift technique requires, in order to support micro-steps, the interleaving of send and receive operations within the same
1.3. STRUCTURED PARALLEL PROGRAMMING

step. This characteristic can produce some difficulties while managing overlapping of communications and computations.

Oversending

Ding and He [15] propose what we call oversending method but is also known as ghost expansion method. They target the reduction of communications per step exploiting a kind of over sending technique. The idea is to expand the data size exchanged between processes in such a way that communications are not required at each step.

By a high level point of view the oversending method is an optimization technique which uses the data parallel paradigm, that is based on distribution of data and replication of the functions, along with mechanism of data replication.

The technique forces a processing unit to compute its own partition and in some steps to update previous received data instead of communicating. With this method, the mean amount of data, that is exchanged per step, is equal to naive and shift ones, but the mean number of send and receive operations per step is reduced.

As the shift, the oversending method too can be interpreted as a stencil transformation, where one step of the modified stencil is a kind of macro-step that corresponds to two or more steps of the original stencil.

A stencil transformed with the oversending method can also be optimized with the shift technique in order to delete diagonal communications as presented in Ding and He paper.

Moreover the authors propose an optimized ghost expansion method for PDE problems that, playing with some algorithmic aspects of PDE solver, reduces also the mean data size exchanged.

Palmer and Nieplocha [39] summarize both the previous methods and present the result of their implementations on different distributed architectures exploiting the message passing library MPI. The main result is that no single algorithm provides optimal performance on all the platforms.

1.3 Structured Parallel Programming

The structured parallel programming features a high influence on our work.

The research stream of structured parallel programming aims the definition of a high level parallel programming environment where programmers manages mechanisms to describe their applications according to a set of parallel patterns. Therefore parallel paradigm of computation are the object of study of this stream.

Essentially, two main concepts can be used to define parallel computation patterns: the functions, which model the flow of operations, and data structures, which instead model the values of the operations.
We can define two distinct classes of patterns according to their relation with the two previous components.

I A class is based on patterns that are focused only on the peculiarities of the functions of a computation. We call this one the functional pattern class.

II A class, which is the most interesting one because of its complexity, is composed by patterns which model features of a computation that take into account both functions and data structures. This class groups all the data parallel paradigm and therefore we call it data parallel pattern class.

The first class has been the object of studies of the approach based on skeletons [12], which can be considered as the forerunner of the structured one. The most studied parallel constructs in the functional pattern class are farm and divide&conquer.

I The Farm skeleton is based on the principle of functional replication. The same computation has to be independently performed over a set of tasks. Hence, the computation is replicated over a set of processes which receive, according to some distribution strategy, the tasks that have to be computed.

The function representing the task computation is exploited as a black box. Thus the farm model does not feature any knowledge about the data structures used by the functions.

II The divide&conquer skeleton is based on the divide et impera paradigm. A problem, that has to be resolved, is recursively broken down into two or more sub-problems, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

As in the farm case, the data structures, that are exploited to represent the problem, are not modelled directly by the skeleton which is still mainly based on functional replications.

Another construct that the skeleton approach takes into analysis is the map. This pattern represents the most easy case among the elements of the data parallel pattern class.

III The map skeleton belongs to the data parallel paradigm, where both functional replication and data partitioning are exploited. The map represents the easiest case of data parallelism, where no functional dependencies are defined between elements.

This is the only case of the three skeletons where the data structures of the applications become a first class object of the model.
In all the cases, the skeleton approach models the parallel patterns as second order functions.

Despite farm, divide&conquer and map structures, there are a set of real worlds applications that are not modelled by those skeletons. An example are stencil-based applications which are represented by data parallel patterns featuring functional dependencies that have to be resolved through communications. Casanova et. al. [11] give a well structured presentation and analysis of some data parallel algorithms highlighting the importance of the topology of the reference parallel architecture model.

In the case of data parallel paradigm where functional dependencies between elements are featured, the approach of defining paradigm as second order function shows strong limitations. Indeed, a specific skeleton would be required for each specific pattern of data parallelism: a skeleton for the Jacobi stencil, one for the Laplace one and so forth. Although the previous cited stencils feature a common generic pattern, the skeleton approach fails in its representation. Hence, attacking the problem of modelling data parallel application according to the skeletal approach, imply the lost of generality and in expressiveness.

The structured approach was defined with the aim of merging functional patterns, data parallel patterns and their compositions. The approach does not fail as the skeletal one, because it represents parallel patterns not as a second order functions, but in terms of basic mechanisms of computation and data management.

In the field of data parallelism, High Performance Fortran (HPF) [37], represents the most important work and most of the optimizations theory based on data dependency analysis are linked to this project; for instance the well known “owner-computes rule” has been defined for the HPF compiler developed at the Rice University [10].

HPF aimed the parallelization of sequential Fortran code, but it can not be classified in the structured research stream because it completely centered in data parallel paradigm and it does not take in consideration the functional ones.

As reference structured environment we consider Assist [47], whose language provide the definition of applications according to functional, data parallel and their compositions. Assist, in its data parallel component, has been influenced by HPF, indeed the actual Assist model is strictly based on the owner-computes rule.

All the compiler associated to the previous parallel programming environments are based on the classical theory of optimizations: in other words they refer to the classical concept of computational equivalence.

In our work, defining a relaxed theory we discover a set of optimizations that can also be exploited to extend the compilers of both Assist and HPF or other structured environments. In the studies of this new optimizations, we borrow the approach of the structured parallel programming methodology: we focus on stencil featuring specific patterns of functional dependencies.

The relaxation of the computational equivalence with the restriction to specific patterns of dependencies, are essential hypotheses of all our reasoning. Both this
bounds form the basis of a computation model where we can define and prove the set of properties that lead to new stencil transformations.

1.4 Plan of the Thesis

The thesis is organized as follows:

Chapter 2: The Structured Stencil Model. We first present a formal model to capture the important features of a significant class of stencils. The model is called “structured stencil model”, where the adjective structured highlights not only the main role that geometric structures play in the model but also represents the purpose of setting our approach in the research stream of structured parallel programming.

Secondly, we present a set of three examples, extracted from well known algorithms in literature, to show how to represent stencils exploiting the structured model. The examples have been accurately selected to provide a wide spectrum of different stencil features.

Then, the Chapter introduces an innovative in-depth classification of stencils. In order to characterize each class and to capture its important features, that are going to be exploited in the rest of the thesis, a set of properties are presented and formally proved.

Finally, a specification of the structured mode is presented for the HUA stencil class, which is relevant entity for our studies.

Chapter 3: The HUA Complete Architecture. We analyze the complete architecture of a framework for programming HUA stencil-based applications. The software architecture that we refer to is composed by four levels called functional dependency, partition dependency, concurrent, and firmware level. In the functional dependency level, a stencil-based application is represented in terms of the structured model, where the functional dependency are highlighted. In the partition dependency level, the HUA stencil description is translated in terms of space partitions. In the concurrent level a representation of the stencil in terms of communicating processes is extracted. Finally in firmware level the program is compiled for the specific architecture.

We analyzes a particular stencil application through each of the architecture levels. During the analysis we provide a description of the best solutions in literature to optimize the number of communications in the implementation of a HUA stencil.

Chapter 4: Q−transformations One of the most used techniques exploited to describe stencil computations is the owner-computes rule. Although the rule makes the definition of a stencil parallelization an easier task, we discover that
alternative solutions can lead to important new optimizations in the relaxed theory that reduce the number of dependencies between domain partitions.

The new transformations, which are classified as relaxed safe, reduce to $n$ the maximum number of communications required to implement a \textit{HUA} stencil, where $n$ is the number of dimensions of the targeted space. The optimizations achieved by the new techniques represent the best result with respect to the solutions that are cited in literature.

\textbf{Chapter 4: Step Fusion Transformations} The scope of this Chapter is the introduction of a new and powerful class of stencil transformations called $Q$ – $\text{Step} – F_{usion}$ ($QSF$).

Data parallelisms is based on data distribution and function replication. A well known technique, called \textit{ghost cell expansion}, aims the reduction of the communication overheads though the exploitation of data replication. The transformation, which in literature is expressed as a transformation at the concurrent level, introduces a interesting thread-off between a reduction of communication overheads and the increase of a computational load.

The $QSF – transformations$ expand and restructure the main concept of the \textit{ghost cell expansion} up to the definition of a new class of stencil transformations which are defined at the functional dependency level instead of at the concurrent one. The new in-depth point of view of the \textit{ghost cell expansion} technique, makes possible the exploitation of both of the results of $Q$–$transformations$, for communication overhead reduction, and of new optimizations which are focused on lowering the computation load.

In the Chapter we analytically demonstrate the benefit of $QSF$–$transformations$ and we validate the results with a complete set of experiments on different kind of computational architectures.

\textbf{Chapter 6: Space Overlapping Transformations}. We focus on the memory requirements for the implementation at the concurrent level of a \textit{HUA} stencil.

Implementations of \textit{HUA} stencil which approximately halve the memory requirement to represent the working domain are well known, but they suffer of the drawback which is given by an increase of the computational load. Indeed those technique require a copy operation for each element of the working domain.

In this Chapter, we present and formally prove the existence of a specific $Q$–$transformations$ called $QM$–$transformations$. The stencil resulting from this new transformations can be associated to an in-place implementation that halves the memory requirements without introducing other computational overheads.
Chapter 7: Conclusions. We present the conclusions of this thesis and describe the future work.

Appendix 4: \textit{MammuT}. This Appendix deeply introduce \textit{MammuT}, the communication library that we used for the test of all our transformations on the Cell architecture.

Structured parallel programming is a parallel software development methodology that aims at delivering programmability, portability and interoperability, along with scalability and performance. To achieve those goals it is important to define both a suitable set of high level parallel constructs and a communication language. The mechanisms of this last language have to provide both very high performance and low overhead, for efficient implementation of parallel construct run time, and a clear cost model that allows for parallel construct composition optimization.

We describe our experience in defining abstract and concrete communication protocols optimized for structured parallel programming on single chip multi-core architectures. We implement and test our mechanisms on the IBM Cell BE chip multi-core. We detail a comprehensive cost model of the communications, that is a requirement for supporting automatic optimization in a structured parallel framework, and we report the achieved performance. Our implementation reaches best possible bandwidth and latency on this architecture: measured performance numbers are extremely close to the actual hardware limits.
Abstract

To start our study of stencil based applications, we need some well defined formalism to describe them and to demonstrate their properties.

In this Chapter, starting from some informal descriptions, we first present a formal model to capture the important features of a significant class of stencils. The model is called “structured stencil model”, where the adjective structured highlights not only the main role that geometric structures play in the model but also represents the porpoise of setting our approach in the research stream of structured parallel programming.

Secondly, we present a set of three examples, extracted from well known algorithms in literature, to show how to represent stencils exploiting the structured model. The examples have been accurately selected to provide a wide spectrum of different stencil features.

Then, the Chapter introduces an innovative in-depth classification of stencils. In order to characterize each class and to capture its important features, that are going to be exploited in the rest of the thesis, a set of properties are presented and formally proved.

Finally, a specification of the structured mode is presented for the *HilA* stencil class, which is relevant entity for our studies.
CHAPTER 2. THE STRUCTURED STENCIL MODEL

Contents

2.1 A Formalization of Stencils Through a Structured Vision 17
   2.1.1 An Informal Analysis of a Generic Stencil 17
   2.1.2 The Formalization of the Structured Stencil Model 22
   2.1.3 Considerations on the “Structured” Feature 29

2.2 Tutorial Examples 30
   2.2.1 Laplace 30
   2.2.2 Red-Black 31
   2.2.3 Floyd-Warshall 39

2.3 A Classification of Space Invariant Stencils 41
   2.3.1 The Equivalence Relation Between Stencil Components 41
   2.3.2 A Classification Based on the Step Set Component 42
   2.3.3 A Classification Based on the Shape Set Component 44
   2.3.4 A Classification Based on Relations Between Application
       Point and Shape 45
   2.3.5 A Classification Based on the Relations Between Shapes 46
   2.3.6 The Semi Uniform Stencil Class 47
   2.3.7 The Space Invariant Stencil Family: The Big Picture 53
   2.3.8 The Boundary Problems 55

2.4 An Extension of Space Invariant Stencils 61

2.5 A Specification of the Structured Model for \( \mathcal{HUA} \) Stencils 64
   2.5.1 The Definition of the \( \mathcal{HUA} \) Model 64
   2.5.2 The Relaxed Computational Equivalence in the \( \mathcal{HUA} \) Model 65
2.1 A Formalization of Stencils Through a Structured Vision

We begin this Section with an informal description of each of the aspects of a stencil computation in order to prepare readers to a following formal definition.

2.1.1 An Informal Analysis of a Generic Stencil

Informally, a stencil based application, or at least one of those that we are targeting in our studies, can be ideally represented as a working domain whose values are changed step by step according to some computations.

An example in a two dimension toroidal space, is the well known Jacobi iterative algorithm, that is based on finite difference approximations for solving partial differential equations. A description in pseudo code of an application based on Jacobi method is reported in Figure 2.1.

```
double J_in[10][10]; J_out[10][10];
load_working_domain_values(J_in);
for(i_step = 0; i_step < 4; i_step ++){
    forall((x,y) ∈ J_in){
        J_out[x][y] = (J_in[x][y + 1] + J_in[x][y - 1] + J_in[x + 1][y] + J_in[x - 1][y])/4;
    }
    swap(J_in, J_out);
}
return_working_domain(J_out);
```

The example features four iterations, which we call steps (in literature they are sometimes also referred as sweeps), over two matrices, which represent what we call the working domain. During a generic iteration $i$, the matrix $J_{out}$ stores the new computed domain element values, while $J_{in}$ stores the older ones, i.e. those computed in the previous iteration.

In the Jacobi algorithm, we use the `forall` construct (line 5 of the pseudo code in Figure 2.1) to highlight that no particular visit strategy is required in order to
perform the update of the $J_{out}$ matrix elements. In one single iteration, $J_{in}$ and $J_{out}$ can be accessed in whatever manner: row major, column major, or blocking. The result of the computation is correct in any case.

More formally, for a single iteration of the outermost cycle, i.e. the one controlled by the $i_{step}$ variable, the statement inside the `forall` loop does not feature any data dependency with itself. Indeed by definition [23], a necessary condition for a data dependency between two statements is that both of them access the same memory location and at least one stores into it. Focusing on the Jacobi pseudo code, the only write operations are computed on $J_{out}$ memory locations where no read operation is performed. Therefore we can conclude that no data dependency is present and consequently no particular pattern of access is required.

Resuming the characteristics of the example of the Jacobi stencil based application, we retrieve that it perfectly matches the informal description of a stencil computation that we gave at the beginning of this Section. Indeed, the Jacobi stencil consists in a working domain, which in turn is modelled by two matrices of real elements, whose values are changed according to the computation of the mean values over a set of real numbers. Two main entities thus have to be characterized in order to formally define a stencil: step and working domain.

The working domain hides a complex nature; we can highlight three important and distinct components: a spatial structure a computational domain and evaluation map.

I The spatial structure models the geometric features of the space where the elements of the stencil application are defined. In the Jacobi stencil the spatial structure represents the space of valid indexes that can be used to access the matrices $J_{in}$ and $J_{out}$.

II The computational domain defines the possible values that can be associated to a generic element of the spatial structure; in the Jacobi example the com-
2.1. A FORMALIZATION OF STENCILS THROUGH A STRUCTURED VISION

Computational domain is \( \mathbb{R} \), i.e. the set of real numbers.

Before proceeding with the introduction of the last component of the working domain, we consider mandatory a comment about the generality of the formalism we are going to present. In our study, we confine ourselves to stencils featuring a particular structure, which is characterized by two main aspects.

1. We target those stencils that do not feature loop-carried dependencies inside a single step computation. Loop dependencies in literature are classified in loop-carried, which arises because of the iterations of the loops, and in loop-independent, which arises when two statements reference the same memory location within a single iteration of all their common loops.

When no loop-carried dependency is detected, it means that there is no restriction on the possible visiting strategies that can be exploited to update all the elements of the working domain.

Significant examples of the two different configurations are Jacobi and Gauss-Seidel algorithms. Both apparently have the same stencil shape, i.e. the two stencils functional dependencies are characterized by the same geometric representation, and also both of them feature a loop-carried dependency associated to the outermost loop, i.e. the one that is controlled by the \( i_{\text{step}} \) variable and models the iteration of the steps. Nevertheless, the Gauss-Seidel algorithm presents also a loop-carried dependency inside one single step. In order to clearly highlight the two behaviours, we report in Figure 2.3 a description in pseudo code of an application based on the Gauss-Seidel method.

```c
double J[101][101];
load_working_domain_values(J);
for(i_{\text{step}} = 0; i_{\text{step}} < 4; i_{\text{step}} + +){
    for(x = 1; x < 100; x + +){
        for(y = 1; y < 100; y + +){
            J[x][y] = (J[x][y + 1] + J[x][y - 1] + J[x + 1][y] + J[x - 1][y])/4;
        }
    }
    return_working_domain(J);
}
```

**Figure 2.3:** Description in pseudo code of the Gauss-Seidel stencil application on a two dimensional space.

The differences are evident when comparing the pseudo code of the two stencil applications (see Figure 2.1 and Figure 2.3).
(a) The Jacobi example computes exploiting two distinct matrices: one for the input values and one for the output one. Differently, Gauss-Seidel exploits one single matrix as data structure; the input matrix is modified in place and then directly returned as result.

(b) Another difference, which in some manner is obviously linked to the previous point, consists in the fact that Jacobi can be expressed as a \textit{forall} cycle over all the elements of the working domain. Instead in the case of Gauss-Seidel, we are forced to exploit \textit{for} constructs to specify a particular visit of the matrix, i.e. in this case a row visit, in order to respect the loop-carried dependency associated to the single step.

This kind of behaviour is a characteristic of algorithms based on dynamic programming techniques, which feature also another important aspect. Usually these algorithms are provided with two main versions, one that features loop-carried dependency in the single step computation and the other that does not. This is the case of iterative methods for partial differential equations, where we face from one side Jacobi and Red-Black methods, i.e. implementations that do not feature loop-carried dependencies, and Gauss-Seidel on the other side.

2. Another characteristic of the stencils which we are going to target concerns the spatial structure of the working domain; we require this entity to be an invariant with respect to the step computation. In other words, the stencil application is described by exactly one input and one output equivalent spatial structures, i.e. both the data structures feature the same index space. We call this class of stencil \textit{space invariant}.

The Jacobi example falls into this category, but other stencil computations do not. For instance, a parallel \textit{reduce} operation is characterized by steps featuring an input index space that is wider than the one that is associated to the output. Another example can be the matrix multiplication algorithm, where we can define two, possibly different, input index spaces, i.e. those associated to the input matrices, and an output one, i.e. the one associated to the resulting matrix.

All the definitions of working domain presented and used in our study refer to the space invariant class. For completeness of exposition, in Section 2.4, we introduce an informal discussion about an extension of the model in order to express also those stencils that do not belong to the \textit{space invariant} class.

We now return back to the presentation of the last component of a working domain.

III The third component of a working domain is a map, in its mathematical definition, which defines a correspondence between the other two working domain components; we call it the \textit{evaluation map}. The map associates to each element of the spatial structure, i.e. to each couple of indexes of the Jacobi
2.1. A FORMALIZATION OF STENCILS THROUGH A STRUCTURED VISION

matrix, one single value of the computational domain, i.e. an element of the set \( \mathbb{R} \) of real numbers.

It could seem that the evaluation map is more or less equivalent to the mechanisms that in sequential programming languages are used to retrieve the value of an element in a multidimensional array. Those mechanisms are usually represented by the square bracket operator. For example, if we consider the element \( e = (x, y) \) in a two dimension array \( J \), the operation \( J[e] \) returns at run time a value of the computational domain.

The evaluation map has a more sophisticated structure and plays a fundamental rule in our model; it is going to be a key component in most of the presented proofs of stencil properties. Three characteristic aspects distinguish the evaluation map from the square bracket operator semantics.

I In a request of evaluation though the map of our model, an instant time is a mandatory parameter. The selected temporal instant specifies, during all the computation time, when the evaluation has to be computed. In other words, the temporal instant establishes the context for the evaluation.

II Even when the time instant is fixed, the evaluation map still features a different nature with respect to the square bracket operator. The map does not return directly a value of the computational domain as the square bracket operator do. As we will see, the result of an evaluation is a formula that is going to be expressed by a couple of entities: a function and a set of spatial element, whose evaluation results are going to be used as input parameters of the function.

III Finally, the computational map, as can be evinced by the previous points, is a static information; it is invariant with respect to both the values of the input working domain and the application run time.

An informal description of the evaluation map is not easy to briefly present because of its complexity. With the previous description we just wanted to give a clue about both the structure and the expressive power of the such mechanism. For a complete and formal definition of the map we refer to Definition 2.1.6, which describe the working domain component, and the Definition 2.1.7, which describe the step component. Indeed, the two previous definitions result in a complete and formal description of the evaluation map.

The time dimension of the evaluation map forces the introduction of a time model for a stencil based application. We consider the steps as those entities that beat the time of a stencil based computation. A interval time between two consecutive steps establishes the context for the working domain evaluation.

By convention, we consider the instant \( i \) the time before the beginning of the computation of the step \( i \) and after the end of the computations of the step \( i - 1 \). In this scenario, steps are modeled as atomic operations. Indeed, the evaluation of an
element of a spatial structure is defined only before or after a step and not during its computation.

Because in our study we focus only on space invariant stencil, we can assert the following property: for a generic step, the only possible difference between the input and the output working domains is represented by the evaluation map. Indeed from the feature of space invariant stencil class, we have that the spatial structure does not change; it is an invariant component in our model. With this prospective, a stencil step can be described as the procedure that specifies how the evaluation map of the input working domain is changed into the one of the output working domain.

2.1.2 The Formalization of the Structured Stencil Model

Based on the previous informal description of stencil components we proceed, following again the same logical path, to their formalization.

Definition 2.1.1 (Space Invariant Stencil). A space invariant stencil \( \psi \) is defined by the following components:

\[
\psi = (W_\psi, T_\psi)
\]  

(2.1)

\( W_\psi \) is called the working domain and \( T_\psi \) the step set.

A stencil therefore is simply modeled by two components, one representing the data of the computation and one concerning the specification of the computation. For instance, the Jacobi application (we associate the stencil to the abbreviation \( JCB \)) can be modeled as follows:

\[
JCB = (W_{JCB}, T_{JCB})
\]

In the stencil structured model, the formal definition of the space invariant stencil component does not give useful information; it is just a container that identifies only other components of a stencil.

To proceed with the formalization we have to analyze both the two stencil components. We focus first on the step set.

Definition 2.1.2 (Step Set). Given a space invariant stencil \( \psi = (W_\psi, T_\psi) \), its step set \( T_\psi \) is defined as an ordered set of steps:

\[
T_\psi = \{ step_1^\psi, \ldots, step_g^\psi \}
\]  

(2.2)

Moreover, the set \( \mathcal{C} = \{1, 2, \ldots, |T_\psi|, |T_\psi| + 1\} \), which is built from the indexes of the ordered set \( T_\psi \), is called evaluation time set and represents the temporal instants when an element of the working domain can be evaluated. Evaluating a working domain at time \( \tau \in \mathcal{C} \) means evaluating it between the \( step_{\tau-1}^\psi \) and \( step_\tau^\psi \) (see Figure 2.4). By definition the evaluation at time \( \tau = |T_\psi| + 1 \) is the evaluation at the end of the last step and it represents the result of the stencil computation.
2.1. A FORMALIZATION OF STENCILS THROUGH A STRUCTURED VISION

The definition of the step set is almost trivial and leaves all the formalization problems to the definition of the step component. By the way, an important aspect derives from the previous definition: the evaluating time set. It establishes the time instants of the computation, during which we can proceed to evaluate a working domain element. According to the definition, an evaluation can be performed only between two contiguous steps. The evaluation map, which is formally presented later on within the step definition, is therefore going to be parametric with respect to a time component, modeled by an element of the evolution time set.

The application based on Jacobi stencil, presented in Figure 2.1, is represented by a step set of four elements:

$$T_{JCB} = \{ \text{step}^{JCB}_1, \text{step}^{JCB}_2, \text{step}^{JCB}_3, \text{step}^{JCB}_4 \}$$

Differently from the stencil formal definition, the step set is not just a container of components. From its structure some important information can be derived to classify stencils. In the Jacobi case, $T_{JCB}$ features the characteristic of being composed by a sequence of the identical steps. Indeed from the application pseudo code, we claim that the computations associated to each step are equal:

$$\forall i, j \; \text{step}^{JCB}_i = \text{step}^{JCB}_j.$$  

In other words, if the order of the elements inside the step set is changed the result of the computation remains the same. We deeply analyze and formalize the semantic of the equality relation between steps in Section 2.3.1.

Concerning the evaluation time set, in our example we have $C_{JCB} = [1, 5]$. Therefore, according to the rules given by the model, it is possible to retrieve a parametric evaluation of an element of the working domain only before beginning of the computation ($\tau = 1$) or before the beginning of a step ($\tau = 1, 3, 4$) or at the end of the computation ($\tau = 5$).

Before metaphorically digging into the definition of the step components, we need to formalize the working domain which was used in the step set definition. As it is going to be clear at the end of the whole model formalization, the different stencil components are strictly linked one to the other. Therefore it is not possible to give independent definitions for each of them: it happens that the definition of one component uses another component and vice versa. This was the case of the step set and the working domain, where we had to give the definition of the first one exploiting the second, without that this last was previously defined in a formal
way. Introducing the spatial domain before would have lead to the same situation.

To introduce the working domain, we start by the formalization of one of its three components, the spatial structure.

**Definition 2.1.3 (General Spatial Structure).** Let $\psi$ a space invariant stencil over a $n$ dimension space. We define the spatial structure $M_\psi$ associated to $\psi$ as a subset of $\mathbb{N}^n$.

The definition collects a wide range of possible spatial structures; it includes both sparse and dynamic ones. While most of the results of our studies can be extended to general spatial structures, in the rest of the document we confine our work only to regular ones.

**Definition 2.1.4 (Regular Spatial Structure).** Given a space invariant stencil $\psi$ over a $n$ dimension space, we define a regular spatial structure $M_\psi$ a particular compact and convex subset of $\mathbb{N}^n$ containing the origin. $M_\psi$ is the result of $n$ cartesian products of natural intervals:

$$M_\psi = [0, m_1 - 1] \times [0, m_2 - 1] \times \ldots \times [0, m_n - 1]$$

The vector $m = (m_1, \ldots, m_n)$ is named length vector. A generic element $e$ of $M_\psi$ is a vector of components $(e_1, \ldots, e_n)$ defined with respect to $(\epsilon_1, \ldots, \epsilon_n)$, which is the standard base of $\mathbb{N}^n$.

Although the formal definition of the regular spatial structure can appear complicated, its meaning is straightforward. The definition selects from all the general structures those that are represented by a index space of a multidimensional matrix. The space of the indexes is represented by the Cartesian products of the index intervals along each single dimension. Each component of the length vector defines the number of items that the index space has along the associated space dimension.

Exploiting again Jacobi based application (see Figure 2.1) as tutorial example, the working domain is given by two $10 \times 10$ matrices, therefore its spatial structure is represented as $M_{JCB} = [0, 9] \times [0, 9]$. For each element $e = (x, y) \in M_{JCB}$ we have $0 \leq x \leq 9$ and $0 \leq y \leq 9$. Finally the $JCB$ length vector is $m_{JCB} = (m_x, m_y) = (10, 10)$.

Actually, the spatial model that we have just given does not exactly match the spatial structure of the Jacobi example, indeed it does not capture the toroidal feature of the space. Because we are going to extract some important properties from toroidal spaces, we present also the definition of a regular spatial structure mapped onto these particular spaces.

**Definition 2.1.5 (Toroidal and Regular Spatial Structure).** Given a space invariant stencil $\psi$ over a $n$ dimension toroidal space, we define a toroidal regular spatial structure $M_\psi$ the result of $n$ cartesian products as it follows:
2.1. A FORMALIZATION OF STENCILS THROUGH A STRUCTURED VISION

\[ M_\psi = \underbrace{\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_n}}_{n \text{ factors}} \]

The symbol \( \mathbb{Z}_{m_i} \) represents a one dimension toroidal space based on a \( m_i \) module. The vector \( m_\psi = (m_1, \ldots, m_n) \) is named module vector.

The representation of the toroidal regular spatial structures is equivalent to the regular one; the characteristics of the index space are the same except for the toroidal property. Indeed, the space identified by the toroidal segment \( \mathbb{Z}_{m_i} \) is equal to the one represented by \([0, m - 1]\), plus the additional toroidal feature.

The number of spatial points associated to the two spatial segment are equal, nevertheless the two associated index spaces are different. In the toroidal case an infinite number of indexes is associated to each spatial element, while in the other case there is a one to one correspondence between elements and indexes. This new defined spatial structure is the right model for the Jacobi working domain.

In our study, we keep in extreme consideration the features of toroidal spaces and consequent possible optimizations that can be defined for two main reasons.

I Obviously, the mechanisms to manage toroidal space can be exploited to represent all those applications that are naively defined on such distinguishing spaces. This aspect features a limited impact because it is hardly difficult to face this kind of applications in real world cases.

II Foremost, a wide range of applications can be remapped on toroidal spaces, as we are going to show later one. Thanks to the remapping mechanisms, certain non toroidal applications can exploit, additionally to their standard features, new properties that come from the new kind of target space.

We can now finally give a complete and formal definition of one of the two components defined by a stencil based application:

**Definition 2.1.6 (Working Domain).** Consider a space invariant stencil \( \psi \) with its a regular spatial structure \( M_\psi \) and its evaluation time set \( \mathcal{C} = \{1, \ldots, g\} \). Let then \( \mathcal{D} \) be the domain of the values associated to each element of the spatial structure; we name it computational domain. We define the evaluation map component as follows:

\[ \text{eval}: M_\psi \times \mathcal{C} \mapsto \mathcal{D}_\psi \]

\[ \text{eval}(e, i) = d \]  \hspace{1cm} (2.3)

Fixed a stencil computation time instant \( i \in \mathcal{C} \), the map \( \text{eval}(e, i) \) associates to each element \( e \) of the spatial structure a value of the computational domain \( \mathcal{D} \). Graphically we define \( \text{eval}(e, i) = \mathcal{M}_{\psi}^{\psi}[e] \).
Finally having defined the previous elements, a working domain $W_\psi$ of a space invariant stencil $\psi$ is given by the following components:

$$W_\psi = (M_\psi, D_\psi, M_i^t[])$$ (2.4)

Concerning the tutorial Jacobi application, we can therefore model the working domain component, according to the previous definition, as:

$$W_{JCB} = (M_{JCB} = \frac{Z}{10} \times \frac{Z}{10}, R, M_i^{JCB}[])$$

The spatial structure is toroidal, and the computational domain is the set of real numbers. About the evaluation map, from the definition of the working domain, we can just describe its type, which if of the form:

$$eval^{JCB} : \left( \frac{Z}{10} \times \frac{Z}{10} \times [1, 5] \right) \rightarrow R$$

In order to complete the whole formalization, the step component remains to be presented. Recalling that for space invariant stencil based applications the only change that a step component can produce on the input working domain is the redefinition of a new evaluation map, we can introduce the following:

**Definition 2.1.7 (Space Invariant Step).** Let $step_\psi^i$ be the i-th step of a space invariant stencil $\psi$, which is associated to a working domain $W_\psi = (M_\psi, D_\psi, M_i^t[])$. $step_\psi^i$ defines parametrically the evaluation map for the spatial structure at application time instant $i + 1$ as follows:

$$\forall e \in M_\psi \xrightarrow{step^i} (F_{(i,e)}, S_{(i,e)}^\psi)$$

$$S_{(i,e)}^\psi = \{g_1, g_2, \ldots, g_k | \forall \alpha g_\alpha \in M_\psi\}$$

$$F_{(i,e)} : D | S_{(i,e)}^\psi | \rightarrow D$$

$$M_{i+1}^t[e] = F_{(i,e)}(M_i^t[g_1], \ldots, M_i^t[g_k])$$

For the generic element $e$ we call $S_{(i,e)}$ the **stencil shape** and $e$ itself the **application point (AP)**. The set $S_{M_i^t}^\psi = \{\forall S_{(i,e)}^\psi | e \in M_\psi\}$ is named the **shape set** of $step_i$.

In a stencil representation through a sequential language, the application point coincides with left hand side element of the inner loop statement, where the computation to update the domain element is expressed. Indeed, the application point specifies the position where the result of the computation is going to be stored.

There is therefore a strong correspondence between the application point and the well known “owner-computes” rule which can be resumed as it follows:
2.1. A FORMALIZATION OF STENCILS THROUGH A STRUCTURED VISION

Definition 2.1.8 (Owner-Computes Rule). In a parallelization of a computation the processing element that owns the left-hand side element of the statement will perform the calculation.

Therefore in our model the previous definition can be revisited and rewritten as:

Definition 2.1.9 (Owner-Computes Rule in the Structured Model). In the structured model for stencil based applications the owner-computes rule establishes that during a parallel computation the processing node holding an application point is in charge of computing the new value.

The owner-computes rule is one of the most used techniques to implement data parallel applications, but, as we are going to show in the rest of the thesis, sometimes it can be a limiting strategy that precludes some interesting optimizations for the reduction of both communication and computation overheads.

Coming back once again to the Jacobi tutorial case, we have that all the steps of the step set are equal, therefore it is sufficient to define one generic step of the set as presented in Table 2.1.

\[
\begin{align*}
\forall e = (x, y) & \in M_{JCB}^{JCB} \xrightarrow{\text{step}^{JCB}} (F_{ie}, S_{ie}^{JCB}) \\
S_{(x,y)}^{JCB} & = \{ (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y) \} \\
F_{(i,e)} & : \mathbb{R}^4 \mapsto \mathbb{R} \\
F_{(i,e)} & : (r_1, r_2, r_3, r_4) \mapsto \frac{1}{4}(r_1 + r_2 + r_3 + r_4) \forall r_i \in \mathbb{R} \\
M_{JCB}^{i+1}[e] & = \frac{1}{4}(M_{JCB}^i[(x, y + 1)] + M_{JCB}^i[(x, y - 1)]) \\
& \quad + M_{JCB}^i[(x + 1, y)] + M_{JCB}^i[(x - 1, y)])
\end{align*}
\]

Table 2.1: Step component in the structured stencil model of the Jacobi stencil application that is described in pseudo code in Figure 2.1
In the Jacobi step model, each element of the spatial structure is associated to a parametric description of a shape, which in turn defines, for the generic application point, the well known geometric cross form. It is worth mentioning that, although the Jacobi step model has been provided for the generic step \( J^{CB}_i \) in a parametric way with respect to the index step \( i \), all the shapes components, i.e. \( S^{CB}_{(x,y)} \), are independent from the parameter \( i \). Therefore all the steps of the application feature the same shape set, which is, as we recall, the stencil component that groups all the shapes defined in the same step set.

Let us now turn our attention to the Jacobi step function. The component is defined from a four dimension input domain to a one dimension output one, where four is the cardinality of all the shape of one step. The step function returns the arithmetic mean of the four components of an element belonging to the \( \mathbb{R}^4 \) domain.

Finally we consider the definition of the Jacobi evaluation map for the time instant \( i + 1 \). It is declared exploiting both the information of the shape and the stencil function. From the example of Jacobi step model, it comes clear what means for us the parametric definition of the evaluation map. The evaluation of an element \( e = (x, y) \in \mathcal{M} \) at time instant \( i + 1 \) depends on the function \( F(i,e) \) and on the evaluation at the previous instant time of those elements that are indicated by the shape \( S^{CB}_{(x,y)} \). Although the exact value of an evaluation can be known only at run time, the evaluation map gives us the tools to prove system invariant properties.

The Jacobi structured model features a simple structure, but on the other hand it shows important characteristics of some main classes of stencils that we are going to present in Section 2.3. We briefly summarize the feature as follows.

I The step set is composed by all equal elements, as demonstrated by the reported single definition of a step model and also by the fact that if we change the sequence of steps, we get the same result.

II The elements of all the shapes can be identified though a linear translation (or equivalently affine transformation) of the corresponding application point. For instance if we consider the element \( \overline{e} = (x, y + 1) \), we have that it belongs to the shape \( S^{CB}_{(x,y)} \) and it can be written as:

\[
\overline{e} = (x, y + 1) = \begin{bmatrix} x \\ y + 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]

where \( (x, y) \) is obviously the element application point.

III The previous cited linear transformation are not completely general. The elements of a shape are described by a linear transformation where the matrix, usually called rotation matrix, is the identity. The characteristic that each shape is equivalent to another one except for a rigid translation means that each element in the working space geometrically features the some functional dependencies.
In the model formalization there is no restriction on the form of the $S^{\psi}_{(i,e)}$ elements. The set can collect, for example, all the elements of the spatial structure or it can be the empty set. The $S^{\psi}_{(i,e)}$ elements can depend on the specific application point or not; this means that in the same step, we can have different shapes with respect to the different application points. From the previous consideration, we can claim that the formalization, for the class of space invariant stencils, is completely generic and does not present any limitation of expressiveness.

2.1.3 Considerations on the “Structured” Feature

Before proceeding with some examples which present how to exploit the previous formalism, we would like to focus a little bit the discussion on the “structured” adjective that we have associated to our model. There are many different reasons to annotate the model with such adjective.

I The “structured” adjective captures the structural nature of the logical path that we followed to define, component after component, the whole model form. Indeed, we started defining a stencil as a couple of simple containers, i.e. the working domain and the step set, and than we recursively give the definition of all the components that characterize a stencil.

II One of the component of the model is the spatial structure, which is exploited in the definition and formalization of most of the stencil components. This fact leads us to the conclusion that the spatial structure is undoubtedly at the core of our model. We consider the reference to the spatial structure role a right justification for the use of the “structured” adjective.

III Foremost, the adjective hides our attempt to highlight that our studies are collocated in the structured parallel programming research stream. The model at the core of structured parallel programming is based on high-level mechanism to express specific parallel patterns of a computation. The strength of this approach is the complete hiding of communication and synchronizations low level aspects from programmers. This comes from the knowledge of the parallel pattern that can be exploited in the production of the parallel implementation.

In our study we concentrate on well specified data parallel patterns. The restrict focus, allow us the complete management of both of the function that are replicated and the data that are distributed. Especially the data management is at the core of the definition of a new optimization theory based on a relaxed vision of computational equivalence: the fundamental concept for studying program transformations.
2.2 Tutorial Examples

In this section we provide the modelling of a set of three stencil based applications: Laplace (LPC), which is an algorithm edge detection in image processing, Red-Black (RB), which is a method for solving partial differential equations, and Floyd-Warshall (FW), which is a well known graph analysis algorithm for finding shortest paths in a weighted, directed graph. The examples has been accurately selected to provide a wide spectrum of different stencil features.

2.2.1 Laplace

Edge detection is a problem of fundamental importance in image analysis. In typical images, edges characterize object boundaries and are therefore useful for segmentation, registration, and identification of objects in a scene. A solution for the problem is the Laplacian operator: a high-pass filter which is mathematically represented by the result of the two-dimensional sum of the second derivatives of an image convolved with a Gaussian curve. The second derivatives detect rapid intensity changes and the Gaussian smooths out the effects of noise.

Consider now an example of application that filters four times a 1024x1024 black and white image, exploiting the Laplace operator. For simplicity we suppose that the image is defined over a toroidal space. A representation of the application in pseudo code is reported in Figure 2.6.

The pseudo code structure is similar to the Jacobi example, the main difference is the shape; Laplace, as it is graphically represented in Figure 2.5, features one element more than the Jacobi shape.

Once again, we consider the hypothesis of the toroidal feature for the space where the application is defined. This hypothesis does not have any correspondence with real world cases of image processing. Indeed, input images hardly ever represents scenes that are toroidal. Nevertheless we model Laplace exploiting this hypothesis.

\textbf{Figure 2.5:} Representation of an application point and the corresponding shape which are featured by the Laplace application whose description in pseudo code is reported in Figure 2.6
double J_in[1024][1024], J_out[1024][1024];
load_working_domain_values(J_in);
for (i_step = 0; i_step < 4; i_step + +){
  forall((x, y) ∈ J_in){
    J_out[x, y] = (J_in[x, y] + J_in[x, y+1] + J_in[x, y-1] \\
                  + J_in[x+1, y] + J_in[x-1, y])/4;
  }
  swap(J_in, J_out);
}
return_working_domain(J_out);

Figure 2.6: Representation in pseudo code of a Laplace stencil application on a two dimensional toroidal space. To keep the annotation light we suppose the indexes are automatically mapped on the toroidal space, i.e. $J_{out}[-1, +1]$ is transformed into $J_{out}[+1023, +1]$ for the sake of exposition simplicity. We present and deeply discuss the problem of modelling application based on non toroidal spaces in Section 2.3.8, providing simple one dimension examples.

The structured model of the Laplace application is reported in Table 2.2. The observation that we can make are the same for the Jacobi stencil example.

I The step set contains steps that are all equivalent. If their order changes no differences can be noted.

II Each element of a shape is described by a linear transformation of the corresponding application point.

III All the shapes, defined in a step, feature the same geometric structure when analyzed exploiting a coordinate system centered into the application point. This is a consequence of the fact that the rotation matrices of the linear transformations are all equal to the identity matrix.

### 2.2.2 Red-Black

We present an example which is based on a variant of the well known Gauss-Seidel relaxation method for equation solver.

The Gauss-Seidel method is similar to Jacobi, except for a different relation with the working domain: Gauss-Seidel features data dependencies inside the same step.
### 2D Laplace (LPC) Model

\[ L_{PC} = (W_{LPC}, T_{LPC}) \]

\[ W_{LPC} = \left( \frac{\mathbb{Z}}{1024} \times \frac{\mathbb{Z}}{1024}, \mathbb{R}, M_{LPC}^i \right) \]

\[ m_{LPC} = (1024, 1024) \]

\[ T_{LPC} = \{ \text{step}_1^{LPC}, \text{step}_2^{LPC}, \text{step}_3^{LPC}, \text{step}_4^{LPC} \} \]

\[ \mathcal{C}_{LPC} = [1, 5] \]

\[ \forall e = (x, y) \in M_{LPC} \xrightarrow{\text{step}^{LPC}} \left( F_{ie}, S_{ie}^{LPC} \right) \]

\[ S_{(x,y)}^{LPC} = \{ (x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y) \} \]

\[ F_{(i,e)} : \mathbb{R}^5 \mapsto \mathbb{R} \]

\[ \mathcal{M}_{LPC}^{i+1}[e] = \frac{1}{4} (\mathcal{M}_{LPC}^i[(x, y)] + \mathcal{M}_{LPC}^i[(x, y + 1)] + \mathcal{M}_{LPC}^i[(x, y - 1)] + \mathcal{M}_{LPC}^i[(x + 1, y)] + \mathcal{M}_{LPC}^i[(x - 1, y)]) \]

---

**Table 2.2:** Structured model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6
double \( J_{\text{in}}[100][100], J_{\text{out}}[100][100] \);  
load\_working\_domain\_values(\( J_{\text{in}} \));  
for(\( i_{\text{step}} = 0; i_{\text{step}} < 4; i_{\text{step}} += 1 \)){  
    forall((x, y) \in J_{\text{in}}){  
        if((x, y) \in \text{black})  
            J_{\text{out}}[x, y] = (J_{\text{in}}[x, y + 1] + J_{\text{in}}[x, y - 1]  
                                + J_{\text{in}}[x + 1, y] + J_{\text{in}}[x - 1, y]) / 4;  
        else  
            J_{\text{out}}[x, y] = J_{\text{in}}[x, y]  
    }  
    swap(\( J_{\text{in}}, J_{\text{out}} \));  
    \( i_{\text{step}} += 1 \);  
}  
forall((x, y) \in J_{\text{in}}){  
    if((x, y) \in \text{red})  
        J_{\text{out}}[x, y] = (J_{\text{in}}[x, y + 1] + J_{\text{in}}[x, y - 1]  
                                + J_{\text{in}}[x + 1, y] + J_{\text{in}}[x - 1, y]) / 4;  
    else  
        J_{\text{out}}[x, y] = J_{\text{in}}[x, y]  
    }  
    swap(\( J_{\text{in}}, J_{\text{out}} \));  
}  
return\_working\_domain(\( J_{\text{out}} \));

**Figure 2.7:** Representation in pseudo code of a Red-Black stencil application on a two dimensional toroidal space. To keep a light annotation we suppose the indexes are automatically re-mapped into the toroidal space, i.e. for example \( J_{\text{out}}[-1, -1] \) matrix access is transformed into \( J_{\text{out}}[99, 99] \).
CHAPTER 2. THE STRUCTURED STENCIL MODEL

Figure 2.8: Representation of two application points and the corresponding shape which are featured by Red step of the Black-Red application whose description in pseudo code is reported in Figure 2.7.

\[ \forall e = (x, y) \in \mathcal{M}_{RB} \xrightarrow{\text{black}} (\mathcal{F}_{ie}, \mathcal{S}_{ie}^{RB}) \]

\[ \mathcal{S}_{(x,y)}^{RB} = \begin{cases} \{(x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\} & \text{if } e \text{ is black} \\ \{(x, y)\} & \text{if } e \text{ is red} \end{cases} \]

\[ \mathcal{F}_{(i,e)} : \begin{cases} \mathbb{R}^4 \mapsto \mathbb{R} & \text{if } e \text{ is black} \\ \mathbb{R}^1 \mapsto \mathbb{R} & \text{if } e \text{ is red} \end{cases} \]

\[ \mathcal{M}_{RB}^{i+1}[e] = \begin{cases} \frac{1}{4}(\mathcal{M}_{RB}^i[(x, y + 1)] + \mathcal{M}_{RB}^i[(x, y - 1)]) \\ + \mathcal{M}_{RB}^i[(x + 1, y)] + \mathcal{M}_{RB}^i[(x - 1, y)]) & \text{if } e \text{ is red} \\ \mathcal{M}_{RB}^i[e] \end{cases} \]

Table 2.3: Structured Step model of the Red steps of the Red-Black (RB) stencil application described Figure 2.7
2.2. TUTORIAL EXAMPLES

Figure 2.9: Representation of two application points and the corresponding shapes which are featured by Black step of the Black-Red application whose description in pseudo code is reported in Figure 2.7

\[ \forall e = (x, y) \in M_{RB} \xrightarrow{red} (F_{ie}, S_{ie}^{RB}) \]

\[ S_{(x,y)}^{RB} = \begin{cases} 
\{(x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\} & \text{if } e \text{ is red} \\
\{(x, y)\} & \text{if } e \text{ is black} 
\end{cases} \]

\[ F_{(i,e)} : \begin{cases} 
\mathbb{R}^4 \mapsto \mathbb{R} & \text{if } e \text{ is red} \\
\mathbb{R}^1 \mapsto \mathbb{R} & \text{if } e \text{ is black} 
\end{cases} \]

\[ M_{i+1}^{i+1}[e] = \begin{cases} 
\frac{1}{4}(M_{RB}^i[(x, y + 1)] + M_{RB}^i[(x, y - 1)]) + M_{RB}^i[(x + 1, y)] + M_{RB}^i[(x - 1, y)] & \text{if } e \text{ is black} \\
M_{RB}^i[e] & \text{if } e \text{ is red} 
\end{cases} \]

Table 2.4: Structured Step model of the Black steps of the Red-Black (RB) stencil application described Figure 2.7
The computations for updating element values during one step, requires values updated in the same step. In a single step, the computations are not independent, therefore it is mandatory to guarantee a specific matrix visit in such a way to respects data dependencies. This kind of situation are usual in dynamic programming techniques to which Gauss-Seidel is strongly related.

An alternative for the Gauss-Seidel method is called Five-point Red-Black Gauss-Seidel relaxation and is based on the exploitation of a Red-Black ordering. The Red-Black is a particular ordering for a matrix data structures, where elements are divided into red and black sets like in a chessboard, as represented in Figure 2.8 and Figure 2.9.

The Five-point Red-Black Gauss-Seidel relaxation method divides one step matrix update in two steps. In the first one, called Black step, the method computes a Jacobi stencil only on black points, leaving unchanged the values of red points. In the second step, called Red step, the algorithm computes Jacobi as well, but only on red points, this time leaving unchanged the black ones.

During a Black step, only red point values are read to update black elements (see Figure 2.8). Symmetrically, during the Red step, only black point values are exploited for the computations of red ones (see Figure 2.9). Reassuming the structure of a generic phase, read operations are performed only on matrix elements of one color and write operation are performed on element of the other color. Because no matrix element is the subject of both a read and a write operations in the same phase, we can claim that there is no data dependencies.

As example, Figure 2.7 reports a description in pseudo code of an application that computes two times the Five-point Red-Black Gauss-Seidel relaxation method on a one hundred by one hundred matrix.

Before analyzing the pseudo code and define a structured model for Red-Black application, we would like to focus the discussion on a well known characteristic of the algorithm.

Because of the Red-black ordering and of its relation with of the Jacobi-like geometry of the stencil shapes, the five-point Red-Black Gauss-Seidel relaxation method features the characteristic that the computations of a generic step can be computed in place, i.e there is no need of two matrices, the input one is used also to store partial and finale results.

In our analysis we discard the possibility of capturing the computation in-place feature for the two reasons that follow.

I We are interested in highlighting the formalization of a stencil based application thought the structured model. For the sake of simplicity in modelling phase, we purposely present a pseudo code that, without considering the in-place computation characteristic, explicitly exploit two matrices. Indeed, Using this description it is easier to observe the data relations in order to define the evaluation map for the step model.
2.2. TUTORIAL EXAMPLES

2D Red-Black (RB) Model

\[ RB = (W_{RB}, T_{RB}) \]

\[ W_{RB} = \left( \frac{Z}{99} \times \frac{Z}{99}, \mathbb{R}, \mathcal{M}_{RB}[.] \right) \]

\[ m_{RB} = (100, 100) \]

\[ T_{RB} = \{ black_1, red_2, black_3, red_4 \} \]

\[ C_{RB} = [1, 5] \]

| Table 2.5: Structures stencil model of the Red-Black stencil application described in pseudo code in Figure 2.7. The step model of Red and Black steps are reported respectively in Table 2.3 and Table 2.4 |

In addition, because the in-place computation property can be statically detected, we claim that it is due to a some compilation strategy to possibly to exploit for optimization purposes.

In the pseudo code Black and Red phases are completely distinct, each one represented by a \texttt{forall} construct to underline the absence of data dependencies inside them. Inside the \texttt{forall}, an \texttt{if} construct is exploited to distinguish the computation that have to be associated to back or red elements. Because we do not exploit in-place computation feature, what in the algorithm exploiting the in-place feature is the operation of leaving some elements unchanged, i.e. black point elements during red steps and red point elements in black steps, in our pseudo code it is implemented by a copy of the value from the input matrix to the output one.

The resulting application is a space invariant stencil that we can represent exploiting the structured model as shown in Table 2.5, except for the step models that are presented in Table 2.3 for red steps and in Table 2.4 for black ones.

The structured model of the Red-Black application is more sophisticated than the Laplace and Jacobi ones. The first difference is the feature of the step set; it is compose by not all equal elements. The most evident difference is the step modelling, where the shapes and the evaluation maps are decided according to the type of color of the application point.
\begin{verbatim}
 double J_in[100][100], J_out[100][100];
 load_working_domain_values(J_in);

 for(i_step = 0; i_step < 100; i_step + +)
{
    forall((x, y) ∈ J_in)
    {
       J_out = min(J_in[x, y], J_in[x, i_step], J_in[i_step, y])
    }

    swap(J_in, J_out);
}

 return_working_domain(J_out);
\end{verbatim}

\textbf{Figure 2.10:} Description in pseudo code of a Floyd-Warshall stencil application on a two dimensional space.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{floyd-warshallStencil.png}
\caption{Representation of two application points and the corresponding shape which are featured, during the iteration \textit{step}_1 (Figure 2.11(a)) and \textit{step}_4 (Figure 2.11(a)), by the Floyd-Warshall application whose description in pseudo code is reported in Figure 2.10}
\end{figure}
2.2. TUTORIAL EXAMPLES

2.2.3 Floyd-Warshall

Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest paths between all pairs of vertexes. The graph is usually represented by a matrix storing the edge weights of a input graph. The method exploits a well known technique of dynamic programming and is based on the following recursive rule:

\[
d_{i,j}^k = \begin{cases} 
  w_{i,j} & \text{iff } k = 0 \\
  d_{i,j}^{k-1} & \text{iff } k > 0 \\
  \min\left( d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1} \right) & \text{iff } k > 0 
\end{cases}
\]

where \( w \) is the matrix of the graph edge weights. The result of the algorithm is stored in the matrix \( d^n \), where \( n \) is the number of nodes of the graph.

We report a description in pseudo code of the algorithm in Figure 2.10, where the Floyd-Warshall method is exploited to calculate the shortest paths of a graph featuring one hundred nodes. In the pseudo code we keep using the same annotation used for the other examples instead of the Floyd-Warshall classic one.

The application is a space invariant stencil that can be represented as reported in Table 2.6. The main peculiar aspect of the structured model of Floyd-Warshall is the dependency of the shape element to the variable \( i \), i.e. the index of the step. The variable defines the triangular access to the matrix typical of the Floyd-Warshall method.
2D Floyd-Warshall (FW) Model

\[ FW = (\mathcal{W}_{FW}, \mathcal{T}_{FW}) \]

\[ \mathcal{W}_{FW} = ([0, 99] \times [0, 99], \mathbb{N}, \mathcal{M}_{FW}^i) \]

\[ m_{FW} = (100, 100) \]

\[ \mathcal{T}_{FW} = \{ \text{step}_1, \ldots, \text{step}_{100} \} \]

\[ \mathcal{C}_{FW} = [1, 101] \]

\[ \forall e = (x, y) \in \mathcal{M}_{FW} \xrightarrow{\text{step}_i} (\mathcal{F}_{ie}, \mathcal{S}_{FW}^{ie}) \]

\[ \mathcal{S}_{FW}^{(x,y)} = \{(x, y), (x, i), (i, y)\} \]

\[ \mathcal{F}_{(i,e)} : \mathbb{N}^3 \mapsto \mathbb{N} \]

\[ \mathcal{M}_{FW}^{i+1}[e] = \min(\mathcal{M}_{FW}^i[(x, y)], \mathcal{M}_{FW}^i[(i, y)] + \mathcal{M}_{FW}^i[(x, i)]) \]

Table 2.6: Structured model of the Floyd-Warshall (FW) stencil application described in pseudo code in Figure: 2.10
2.3 A Classification of Space Invariant Stencils

In this section, we present a classification of spatial invariant stencils which results from the analysis of the possible geometric or analytic characteristics of the various components of the structured stencil model. The same classification can be easily extended to non space invariant stencil, which are briefly presented in Section 2.4.

In Section 2.3.1 we report some definitions of comparing operations for structured stencil components, which are then exploited in Section 2.3.2 and 2.3.2 to picture a stencil classification based on the regularity properties.

2.3.1 The Equivalence Relation Between Stencil Components

In order to classify stencil we need mechanisms to formally compare them. In this Section we present some definitions of equivalence between stencil components, such as step and shape.

The step is a complex stencil component: it defines the shape set, the association of a shape to each element of the spatial structure, and also the evaluation map for a certain time instant. We distinguish two different types of equivalence, the first one is the following.

Definition 2.3.1 (Structural Step Equivalence). Two steps \( \text{step}^\psi_i, \text{step}^\psi_j \) of the same space invariant stencil \( \psi \) are structurally equivalent when they feature the same set shape and both of them associate the same shape to each element of the spatial structure. We use the annotation:

\[
\text{step}^\psi_i \sqsubseteq \text{step}^\psi_j
\]

When two steps are structurally equivalent, all their elements feature the same functional dependencies between consecutive steps. Nevertheless it is possible that, exploiting two structurally equivalent stencils, the computation of an identical input working domain result in different output working domain. In fact this kind of equivalence does not introduce any bounds on the computational functions associated to evaluation map. Differently, in the Jacobi example all steps are structurally equivalent, but they also return the same results when they are applied to the same working domain. To guarantee that the two steps return the same result we have to introduce the following definition.

Definition 2.3.2 (Complete Step Equivalence). Two steps \( \text{step}^\psi_i, \text{step}^\psi_j \) of the same space invariant stencil \( \psi \) are completely equivalent when they feature the same set shape and both of them associate the same shape and the same function to each element of the spatial structure. We use the annotation:

\[
\text{step}^\psi_i \sqcirc \text{step}^\psi_j
\]
It is trivial to prove that two completely equivalent steps on the same input working domain produce the same identical results.

In the Red-Black example, all the Black steps are completely equivalent as well as the Red ones. Instead there are no structurally or completely equivalence between the steps of the Floyd-Warshall method. Although we gave one single parametric description of all the application stencil steps, we exploited the index of the step set as parameter in order to define the shapes. Indeed, analyzing a generic spatial structure element with respect to all the application steps, we find that the associated shape is always different. A graphic representation of this consideration is reported in Figure 2.11(a) and 2.11(b), where the shape of two elements is pictured for two different steps.

Another component, whose equivalence relation is going to be important for the rest of the chapter, is the shape. With this type of equivalence we are interested in highlighting some geometric features that can link the elements of two different shapes:

**Definition 2.3.3 (Shape Equivalence).** Two shapes, which belong to the same shape set of a stencil $\psi$, are equivalent if, in a reference system centered in the associated application point, all the elements are defined by the same vectors. In other words $S_{(i,e_1)}^\psi$ and $S_{(i,e_2)}^\psi$ are equivalent if

$$S_{(i,e_1)}^\psi - e_1 = S_{(i,e_2)}^\psi - e_2$$

The subtraction operation of the component $e$ to the set $S_{(i,e)}^\psi$, translates each element of the shape in a reference system centered into the application point. To highlight that two shapes are equivalent, we use the following annotation:

$$S_{(i,e_1)}^\psi \equiv S_{(i,e_2)}^\psi$$

A less formal view to see the previous definition, is that an observer, that analyzes the two shapes standing on their respective application points, does not notice any difference in the disposition of the shape elements in the space.

In Jacobi stencil application we faced all equivalent shapes as well as in the in Laplace one. In a generic Red-Black step, all the shapes associated to the same element color are equivalent, while in a single Floyd-Warshall step, there are no equivalent shapes.

### 2.3.2 A Classification Based on the Step Set Component

We move now the focus of the discussion on the study of a stencil classification based on the geometric or analytic properties of the various components of the structured model. The first one that can be easily exploited to classifying stencils is the step set. In both Jacobi and Laplace method modelling, we highlighted that the step set is defined by a sequence of the structural equivalent steps. Therefore a property to
2.3. A CLASSIFICATION OF SPACE INVARIANT STENCILS

classify stencil in different categories are the structural or complete equivalence of all the element of the step set.

**Definition 2.3.4 (Homogeneous Stencil Class).** Let \( \psi \) be a spatial invariant stencil featuring \( T_\psi = \{ \text{step}_1^\psi, \ldots, \text{step}_g^\psi \} \) as its step set. \( \psi \) is classified as an **homogeneous** stencil if all steps define one of the following properties:

I Structural homogeneity: \( \forall \text{step}_i^\psi, \text{step}_j^\psi \in T_\psi \ \text{step}_i^\psi \cong \text{step}_j^\psi \)

II Complete homogeneity: \( \forall \text{step}_i^\psi, \text{step}_j^\psi \in T_\psi \ \text{step}_i^\psi \equiv \text{step}_j^\psi \)

Stencils that do not belong in this class are called **non homogeneous**.

A graphical representation of the stencil classes defined though the characteristics of the step set is reported in Figure 2.12.

Both Jacobi and Laplace are classified as homogeneous stencils because they are composed by only structurally equivalent steps. The Red-Black stencil, instead, is non homogeneous because it features two different kind of steps; ones for computations that update black points, i.e. Black steps, and ones for computation that update red points, i.e. Red steps. Likewise Red-Black, Floyd-Warshall is classified as non homogeneous; in the stencil step set there are no steps that are equivalent.

The property of complete equivalence is stronger than the structural one, indeed it is trivial to prove that the first property implies the second one. From another prospective we can say that the class of the stencils that are structurally equivalent contains all the steps that are completely equivalent. In the rest of the thesis, we are going to discuss about some optimizations of communication that require the structural equivalence and some other optimizations of cache management that require the more strict complete equivalence. For simplicity in the rest of the thesis when we will refer to equivalence between steps, without any other specifications, we will refer to the structural equivalence. In this way we can claim that all the properties that we will prove thanks to step equivalence are effective for both structural and complete equivalence,
2.3.3 A Classification Based on the Shape Set Component

Another important structured model component, whose features can be used to catalogue stencils, is the shape set, which was previously defined for a generic stencil $\psi$ in Definition 2.1.7 as:

$$S^\psi_M = \{S^\psi_{(i,e_1)}, \ldots, S^\psi_{(i,e_k)}\}$$

The analysis of the shape set leads to different reflections, all of them are linked by the aim of finding some shared structural characteristics between the shape set elements, which are in the form:

$$S^\psi_{(i,e)} = \{g_1, g_2, \ldots, g_k \mid \forall \alpha g_\alpha \in M_\psi\}$$

We recall that by definition in the shape set there is one and only one element (i.e. a shape) that is associated to each spatial structure element (i.e. the application point). In turn, each shape set element is composed by a set of spatial structure elements.

Figure 2.13: Classification of space invariant affine stencils
In Figure 2.13 we schematize a preview of the stencil families that we are going to describe in the following Sections.

2.3.4 A Classification Based on Relations Between Application Point and Shape

We would like now to target the characterization of stencils according to some relations between each shape element with the corresponding application point. In other words, we are trying to model those stencils whose shapes can be described parametrically with respect to their application points. With this specific aim, we define the following space invariant class.

**Definition 2.3.5 (Affine Space Invariant Stencil Class).** Let \( \psi \) a space invariant stencil featuring \( S_{M_i}^\psi \) as a parametric representation of the shape set for its generic \( \text{step}_i^\psi \). We define \( \text{step}_i^\psi \) as **affine step** if each of its shapes can be represented as a linear transformation of the corresponding application point:

\[
\forall S_{(i,e)}^\psi \in S_{M_i}^\psi, \quad \forall g \in S_{(i,e)}, \quad \exists A_g, \quad g = A_g e + q_g
\]

\( A_g \) is the **rotation matrix**, which features a non null determinant, while \( q_g \) is the **translation vector**.

We classify \( \psi \) as **affine stencil** if all its steps are affine. A stencil that does not fall in this class is tagged as **non affine**.

The shape of an affine and space invariant stencil can therefore be represented as:

\[
S_{(i,e)}^\psi = \{ A_{e_1} e_1 + q_1, A_{e_2} e_2 + q_2, \ldots, A_{e_k} e_k + q_k \mid \forall \alpha (A_{e_\alpha} e_\alpha + q_\alpha) \in M_\psi \}
\]

Consider for instance the stencil described by the following statement:

\[
J_{\text{in}}(x,y) = J_{\text{out}}(x^2, y^2) + J_{\text{out}}(x, 3 \ast y) + J_{\text{out}}(x, y)
\]

No one shape of the stencil can be defined as affine because each one has an element whose parametric description that is not expressed as a linear function.

All the examples that we have presented up to now are classified as affine, and most of the real stencil applications belongs to this class.

The affine classification is completely orthogonal to the homogeneous one. It is possible to have stencils which are homogeneous but not affine or vice versa; all the possible combination are legal. This property of orthogonality comes from the fact that the two classifications focus on features which are associated to two independent stencil components: the step set for the homogeneity and the shape elements for their descriptions in terms of linear transformations of the associated application point.
2.3.5 A Classification Based on the Relations Between Shapes

Considering the affine stencil set, we can study a more fine classification analyzing not only the geometric features of a single shape but also the possible relations that can be highlighted between shapes of the same stencil step.

A significant relation is represented by a feature of the step set component: it can be composed by only equivalent shape, as it happen in both Jacobi and Laplace examples. We therefore introduce this first sub-classification of affine stencil family.

**Definition 2.3.6 (Uniform Affine Stencil Class).** Let \( \psi \) be an affine stencil and \( S^\psi_{(i,e)} \), its shape set for the step \( step_i \). The \( step_i \) is classified as uniform step if all its shapes are equivalent:

\[
\forall e_\alpha, e_\beta \in \mathcal{M}, \ S^\psi_{(i,e_\alpha)} \triangleq S^\psi_{(i,e_\beta)}
\]

We classify \( \psi \) as uniform stencil if all its steps are in turn uniform.

The definition introduces the uniform class directly as a subset of the affine one. The soundness of the definition is given by the following theorem, which asserts that it is not possible that an uniform stencil exists that does not belong to the affine class.

**Theorem 2.3.1 (Uniform and Affine classes Relation).** The class of Uniform stencil is a proper subset of Affine one:

\[
\text{Affine} \subset \text{Uniform}
\]

**Proof.** The proof is quite easy and straightforward. If two shapes are equivalent than we have:

\[
S^\psi_{(i,e_\alpha)} - e_\alpha = S^\psi_{(i,e_\beta)} - e_\beta
\]

Expanding the definition of the step set we get:

\[
\forall e_\alpha, e_\beta \in \mathcal{M}, \ \{ g_1^\alpha - e_\alpha, g_2^\alpha - e_\alpha, \ldots, g_k^\alpha - e_\alpha \} = \{ g_1^\beta - e_\beta, g_2^\beta - e_\beta, \ldots, g_k^\beta - e_\beta \}
\]

Recalling that the sets we are considering are ordered sets:

\[
\forall e_\alpha, e_\beta \in \mathcal{M}, \ g_j^\alpha - e_\alpha = g_j^\beta - e_\beta
\]

The previous equality has to be valid for all the possible elements of the spatial structure, because of the definition of uniform stencil class. The only way to satisfy the equality is therefore that each element \( g_j \) is described parametrically with respect to the application point in the following way:

\[
S^\psi_{(i,e)} = \{ Ie + q_1, Ie + q_2, \ldots, Ie + q_k \mid \forall \alpha (Ie + q_\alpha) \in \mathcal{M}_\psi \}
\]

where \( I \) is the identity matrix. The form of \( S^\psi_{(i,e)} \) implies that the a uniform stencil is an affine stencil with the identity matrix as its rotation matrix. \( \square \)
From the previous Theorem comes an important corollary on the analytic structure of shapes in semi uniform stencils.

**Corollary 2.3.1 (Uniform Shape Structure).** A necessary condition for a stencil to belong to the uniform class is that each shape is defined as follows:

$$S^\psi_{(i,e)} = \{ I e + q_1, I e + q_2, \ldots, I e + q_k \mid \forall \alpha (I e + q_\alpha) \in M^\psi \}$$

Where \( I \) is the identity matrix.

**Proof.** The proof comes directly from the demonstration of Theorem 2.3.1.

Stencils belonging to the uniform class hide an important feature that strictly links them to toroidal spaces. This characteristic is introduced and proved in the following property:

**Property 2.3.1 (Uniform Stencil Toroidal Spaces).** Uniform stencils are defined only over toroidal spaces.

**Proof.** We use a reductio ad absurdum strategy for the demonstration; we therefore assume that it exists an uniform stencil \( \psi \) that is defined over a not toroidal regular spatial structure \( M^\psi \), characterized by a length vector \( m = (m_1, \ldots, m_n) \).

By definition of uniformity and by Corollary 2.3.1, an uniform stencil features a set of only uniform steps, whose associated shape sets are in turn composed by all equivalent shapes, with the following form:

$$S^\psi_{(i,e)} = \{ I_1 e + q_1, I_2 e + q_2, \ldots, I_k e + q_k \mid \forall \alpha (I_\alpha e + q_\alpha) \in M^\psi \}$$

We suppose a vector element \( q_\gamma = (q_0, \ldots, q_n) \) exists such that at least one component \( q_i \) is positive. Consider now the element of the spatial structure associated to the vector \( e = (m_1 - 1, \ldots, m_n - 1) \) as application point. In the corresponding shape there exists the element \( e + q_\gamma = (m_0 + q_0, \ldots, m_i + q_i, \ldots, m_n + q_n) \), which does not belong to the spatial structure; indeed, its \( i \)-th component, i.e. \( m_i + q_i \), is equal or greater than \( m_i \). We can therefore claim that we reach an absurdum, in fact by definition of shape all the elements have to belong to the spatial structure of the stencil. The same reasoning can be symmetrically applied considering a negative component of \( q_\gamma \).

We can conclude that the only space where we can define an uniform stencil has to be toroidal.

2.3.6 The Semi Uniform Stencil Class

In this Section we target the definition of another subclass of the affine stencils. The subclass highlights a set of affine stencils that, with some extensions, can be easily reported to the uniform class.
Consider as example a Black step of the Red-black application, if we analyze only the shapes associated to black elements, i.e. those featuring a cross geometry, we should say that the black shapes are affine and make up a uniform subset of the black shape set. The same reasoning can be symmetrically applied to red points, i.e those featuring a shape that is composed by only one element. To summarize, all the red shapes and also the black ones can be expressed as an affine transformation of the application point where the rotation matrix is the identity. This is a necessary, but not sufficient, condition for a stencil to be classified into the uniform stencil set as Corollary 2.3.1 claims.

What a black step lacks to be classified as uniform is to feature the same elements on both black and red shapes. Nevertheless, this drawback can be avoided defining an extension of the step such that the resulting step is uniform.

The key point is to extend, in a Black step, both the black and red shapes with a set of elements, in such a way that all the new shapes are equivalent. We call the set $U$, the set of uniformity.

It is worth mentioning that the presented extension is at the level of the structure of the shapes, indeed the new added elements are not going to be considered for the definition of the evaluation map.

We show the definition of $U$ in the case of Red-Black stencil in Table 2.7 for black steps and in Table 2.8 for the red steps. Moreover, a graphical representation of the two extensions is reported respectively in Figure 2.15 and Figure 2.14.

Analyzing the set $S^{RB}_{(x,y)} \cup U$ as the new shape we find that the extended version of both back and red steps are classified as uniform, because now all the spatial structure elements feature the same shape elements.

From the two examples, it comes clear that, as previously noted, although the shapes were extended with new elements in order to reduce the stencil to an object of the uniform class, the step functions and evaluation map were not modified; they are still defined only exploiting the elements of the original shapes.

Based on the previous observations we introduce the following new sub-classification of affine stencils.

**Definition 2.3.7 (Semi Uniform Stencil Class).** Let $\psi$ be an affine stencil and $S^\psi_M$, its shape set for the step $step_i$. The $step_i$ is classified as uniform if there exists a set $U \subset \mathcal{N}^n$, called uniformity set, such that:

$$
\forall e_\alpha, e_\beta \in \mathcal{M}, \quad \left( S^\psi_{(i,e_\alpha)} \cup (U + e_\alpha) \right) \overset{\triangleq}{=} \left( S^\psi_{(i,e_\beta)} \cup (U + e_\beta) \right)
$$

$$
\forall e \in \mathcal{M}, \quad (U + e) \subset \mathcal{M}
$$

(2.5)

Stencil $\psi$ is classified as semi uniform stencil if all its steps are semi uniform; where each step can feature a different uniform set.

Informally we say that a stencil belongs to the semi uniform class if we can define a set $U$, parametric with respect to the application point, in such a way that
Figure 2.14: Representation of two application points and the corresponding shapes which are featured by Red step of extended version of the Black-Red application whose description in pseudo code is reported in Figure 2.7. The picture can be compared with the original one reported in Figure 2.8.

Figure 2.15: Representation of two application points and the corresponding shapes which are featured by Black step of extended version of the Black-Red application whose description in pseudo code is reported in Figure 2.7. The picture can be compared with the original one reported in Figure 2.9.
\( \forall e = (x, y) \in M_{RB}^{black} \rightarrow (F_{ie}, S_{ie}^{RB}) \)

\[\mathcal{U} = \{(0, 0), (0, +1), (0, -1), (+1, 0), (-1, 0)\}\]

\[S_{(x,y)}^{RB} \cup (\mathcal{U} + e) = \begin{cases} 
\{(x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\} \cup \\
\mathcal{U} \{(x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\}\end{cases}\]

if \(e\) is black

\[\mathcal{S}_{(x,y)}^{RB} \cup (\mathcal{U} + e) = \begin{cases} 
\{(x, y)\} \cup \\
\mathcal{U} \{(x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\}\end{cases}\]

if \(e\) is red

\[F_{(i,e)} : \begin{cases} 
\mathbb{R}^4 \rightarrow \mathbb{R} \text{ if } e \text{ is black} \\
\mathbb{R}^1 \rightarrow \mathbb{R} \text{ if } e \text{ is red} \end{cases}\]

\[M_{RB}^{i+1}[e] = \begin{cases} 
\frac{1}{4}(M_{RB}^i[(x, y + 1)] + M_{RB}^i[(x, y - 1)]) + M_{RB}^i[(x + 1, y)] + M_{RB}^i[(x - 1, y)] \text{ if } e \text{ is red} \\
M_{RB}^i[e] \text{ if } e \text{ is black} \end{cases}\]

Table 2.7: Structured Step model of the Red steps of the extended Red-Black (RB) stencil application described Figure 2.7
2.3. A CLASSIFICATION OF SPACE INVARIANT STENCILS

Extended 2D Black step Model

\[ \forall e = (x, y) \in M_{RB} \xrightarrow{red} (F_{ie}, S_{ie}^{RB}) \]

\[ U = \{ (0, 0), (0, +1), (0, -1), (+1, 0), (-1, 0) \} \]

\[ S_{RB}^{RB} \cup (U + e) = \begin{cases} 
(x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y) \} \cup \\
\end{cases} \]

\[ \cup \{ (x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y) \} \]

if \( e \) is black

\[ \cup \{ (x, y) \} \cup \]

\[ \cup \{ (x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y) \} \]

if \( e \) is red

\[ F_{(i,e)} : \begin{cases} 
\mathbb{R}^4 \rightarrow \mathbb{R} \text{ if } e \text{ is red} \\
\mathbb{R}^1 \rightarrow \mathbb{R} \text{ if } e \text{ is black} 
\end{cases} \]

\[ M_{RB}^{i+1}[e] = \begin{cases} 
\frac{1}{4} (M_{RB}^i[x, y + 1] + M_{RB}^i[x, y - 1]) + M_{RB}^i[(x + 1, y)] + M_{RB}^i[(x - 1, y)] \text{ if } e \text{ is black} \\
M_{RB}^i[e] \text{ if } e \text{ is red} 
\end{cases} \]

Table 2.8: Structured Step model of the Black steps of the extended Red-Black (RB) stencil application described Figure 2.7
its elements, added to each shape of the step, transform the extended stencil in a uniform one. There is a condition to respect in the definition of \( U \). The elements that are added to a shape, i.e. the set \( U + e \), have to form a strict sub set of the spatial structure \( M \): \( (U + e) \subset M \).

From the definition we can derive a necessary condition for classify a stencil as semi uniform.

**Theorem 2.3.2 (Semi Uniform Stencil Shape Structure).** A necessary condition for a stencil step to belongs to the semi uniform class is that each of its shapes \( S^\psi_{(i,e)} \) is in the form:

\[
S^\psi_{(i,e)} = \{ Ie + q_1, Ie + q_2, \ldots, I + q_k \mid \forall \alpha (Ie + q_\alpha) \in M^\psi \}
\]

Where \( I \) is the identity matrix.

**Proof.** From Corollary 2.3.1 we have that in an uniform step all the shapes can be modelled in the following form:

\[
S^\psi_{(i,e)} = \{ Ie + q_1, Ie + q_2, \ldots, I + q_k \mid \forall \alpha (Ie + q_\alpha) \in M^\psi \}
\]

Thus in the case of a semi affine stencil we have that: \( \forall e_\alpha, e_\beta \in M \),

\[
(S^\psi_{(i,e_\alpha)} \cup (U + e_\alpha)) = \{ Ie + q_1, Ie + q_2, \ldots, I + q_k \mid \forall \alpha (Ie + q_\alpha) \in M^\psi \}
\]

Each element of both components of the union operation has to be defined as \( Ie_\alpha + q_\alpha \).

About the uniformity sets we are interested in defining the smallest one, i.e. the one featuring the lowest cardinality. Indeed, if the smallest one is not a subset of \( M \), we can claim that the associated stencil can not be classified as semi uniform. Therefore, we present the following Theorem:

**Theorem 2.3.3 (The Smallest Uniformity Set).** Let \( \psi \) be a semi uniform stencil and \( S^\psi_{M_i} \) its shape set for the step \( step_i \). The smallest uniformity set \( U^{MIN}_i \) for \( step_i \) is given by the following formula:

\[
U^{MIN}_i = \bigcup_{e \in M} \left( S^\psi_{(i,e)} - e \right) \tag{2.6}
\]

**Proof.** The demonstration is quite trivial. Firstly, \( U^{MIN}_i \) is a uniformity set for \( step_i \). Indeed by Definition 2.3.3 of shape equivalence we have that the Equation 2.8 can be rewritten as:

\[
\forall e_\alpha, e_\beta \in M, \left( S^\psi_{(i,e_\alpha)} - e_\alpha \right) \cup U^{MIN}_i = \left( S^\psi_{(i,e_\beta)} - e_\beta \right) \cup U^{MIN}_i
\]

The previous equation is guaranteed by the construction of \( U^{MIN}_i \).
Secondly, by absurdum we suppose that there exists \( \mathcal{U} \), a smaller uniformity set than \( \mathcal{U}_{i}^{\text{MIN}} \). Because the new set is smaller there exists an element \( g \) such that \( g \in \mathcal{U}_{i}^{\text{MIN}} \) and \( q \notin \mathcal{U} \). By construction of \( \mathcal{U}_{i}^{\text{MIN}} \) there exists at least one spatial structure element \( e_q \) such that its shape contains \( e_q + q \):

\[
(e_q + q) \in \mathcal{S}^\psi_{(i,e_n)}
\]

Because \( e_q + q \) belongs to a step shape but not to \( \mathcal{U} \), this last set can not be an uniformity set and we can assert that we reached a contradiction. We therefore conclude the \( \mathcal{U}_{i}^{\text{MIN}} \) is the smallest uniformity set.

About the semi uniform stencil class definition it is worth mentioning that \( \mathcal{U} \) is required to be a strict sub set of \( \mathcal{M}^\psi \). Consider the case of Floyd-Warshall: each shape have different elements, but, differently from the Red-Black stencil, the shapes depends on the step index which characterizes the typical triangular access to the matrix. In such a case, the only uniform set is the entire spatial structure \( \mathcal{M} \), therefore Floyd-Warshall stencil is not classified as semi uniform.

**Theorem 2.3.4 (Semi Uniformity).** A both sufficient and necessary condition for a step \( i \) to be classified as semi uniform is the validity of the following formula:

\[
\forall e \in \mathcal{M}, \quad \mathcal{U}_{i}^{\text{MIN}} \subset \mathcal{M}
\]  

*(2.7)*

**Proof.** The proof comes directly form Definition 2.3.8 and the Theorem 2.3.3

From the previous Theorem we can conclude that the semi affinity is a characteristics that can be automatically proved knowing all the information about the steps of the stencil.

**2.3.7 The Space Invariant Stencil Family: The Big Picture**

In this Section we would like to summarise all the properties of the space invariant stencils that we presented in the previous Sections.

The relations about the different stencil classes are straightforward to be presented. First of all we have two wide families of stencils that represent Homogeneous and Affine classes. As previously noted, the two classes are orthogonal one each other, i.e. they are independent: an homogeneous stencil can be affine or not and vice versa. While homogeneous class does not have any other sub-classification, in the affine set we distinguish three sub classes: non uniform, semi Uniform and uniform. Until now, we did just demonstrate in Proposition 2.3.1 that uniform stencils are completely contained into the affine class, and we also know by definition that non uniform stencil does not contain neither uniform nor semi uniform stencils. What remain to model is the correspondence between uniform and semi uniform, we therefore introduce the following property.
Property 2.3.2 (Uniform and Semi Uniform Classes Relations). Uniform class is a proper subclass of the semi uniform one:

\[ \text{Uniform} \subset \text{Semi Uniform} \]

Proof. Exploiting Theorem 2.3.4, it is trivial to see that for uniform stencil we have:

\[
\bigcup_{\forall e \in \mathcal{M}} \left( S_{(i,e)}^\psi - e \right) = \bigcap_{\forall e \in \mathcal{M}} \left( S_{(i,e)}^\psi - e \right) \subset \mathcal{M}
\]

Now that we have complete the entire set of relations between the different classes of space invariant stencils, we can resume all the information previous acquired in the representation of Figure 2.16.

An important aspect of the graphical representation is the intersection of uniform and homogeneous stencils: we call this class \( \mathcal{HUA} \), which stands for Homogeneous,
2.3. A CLASSIFICATION OF SPACE INVARIANT STENCILS

Uniform and Affine. In the rest of the thesis this is going to be our main issue. All the transformations we are going study are firstly presented for HUA stencils and then extended to other classes. Therefore, because of the relevance of HUA stencil in our study, in Section 2.5 we are going to introduce a specification of the structured model that explicitly highlights, with proper mechanisms, the distinguishing features of this stencil class.

2.3.8 The Boundary Problems

```plaintext
double J_in[10], J_out[10];
load_working_domain_values(J_in);
for(i_step = 0; i_step < 4; i_step ++){
    forall(x ∈ J_in){
        if(x is left border){
            J_out[x] = J_in[x + 1]
        }
        if(x is not border){
            J_out[x] = \frac{1}{2}(J_in[x] + J_in[x + 1] + J_in[x - 1])
        }
        if(x is right border){
            J_out[x] = J_in[x - 1]
        }
    }
    swap(J_in, J_out);
}
return_working_domain(J_out);
```

Figure 2.17: Representation in pseudo code of a Jacobi stencil application on a one dimensional non toroidal space.

All the examples that we have presented up to now, except for Floyd-Warshall one, were defined over toroidal spaces, but usually in real world examples the application spaces are not toroidal. In this cases the problem of managing the spatial structures bounds raises.

We have seen with the proof of Theorem 2.3.1 that an uniform stencil imply a toroidal space. In stencils defined over non toroidal space the stencil shape, which are associated to elements next to the space bounds, have to be managed differently
with respect to the others, i.e. applications with bounds space do not belong to the uniform class.

We consider the case of one dimension Jacobi defined by a regular spatial structure, which by definition is not toroidal, and we analyze what kind of information we can extract by analyzing its structured model. We choose to present a one dimension space example instead of reusing the two dimension Jacobi for the sake of simplicity: in a one dimension space we face only two classes of bound elements, while in a two dimension case they are eight.

We report the pseudo code in Figure 2.17 and a graphical representation in Figure 2.19. The non toroidal algorithm features, inside the \texttt{forall} construct, three \texttt{if} statements in order to divide the spatial structure elements in the following regions: \textit{left border}, \textit{right border} and \textit{internal}. The classical one dimension Jacobi shape is associated to the elements of the \textit{internal} region, as in the same way it would be for the case of a toroidal structure. Instead, to the elements of the \textit{left border} and \textit{right border} regions, which in this particular case are composed by one element each, a reduced shape is associated to, i.e. a classical shape thinned out of those element that would not be inside the spatial structure.

We report the structured model of the one dimension Jacobi in Table 2.9. In the description it is interesting to highlight two main aspects. Firstly, the working domain is defined over a non toroidal domain. Secondly, both the shape and step function declarations are parametrically described with respect to the membership of a spatial structure element to one of the three regions.

It is now interesting to consider the classification of the one dimension Jacobi. Firstly the stencil application belongs indisputably to the intersection of both homogeneous and affine classes. Secondly the stencil is undeniably out of the uniform class, because the shape associated to the borders are each other not equivalent and in turn both are not equivalent with the shape of the internal region elements. Considering the big picture of the stencil classification (see Figure 2.16), it remains an open problem if the stencil should be classified or not as semi uniform.

Following the same path made for the Red-Black stencil, that took us to the definition of the semi uniform class, we study an extension of the Jacobi model which is reported in Table 2.10 and it is also graphically represented in Figure 2.19.

The extended model is indisputably uniform, but we were able to define the set $U$ only paying the price of remapping the extended model on a toroidal space: $\mathcal{W}_{JCB} = (\mathbb{Z}/10\mathbb{Z}, \mathbb{R}, \mathcal{M}^{JCB}_{\text{toroidal}})$). Indeed, a set of uniformity does not exist that the extended shape of the left and right border elements can be equivalent to the extended shape of the internal section elements. The previous claim is guaranteed by the necessary condition for semi uniformity claimed by Theorem 2.3.4.

To summarise, we have defined an Jacobi stencil extension that is classified as uniform. Nevertheless, differently from the case of the Red-Black application, from the existence of the uniform extended version we can not classify, according to the current model formalization, the original non toroidal Jacobi as semi uniform. Indeed, the definition of semi uniform class (see Definition 2.3.8) establishes that a
stencil can be classified as semi uniform if it exists an extension, modeled by a set named uniformity set, that is classified as uniform. It is worth mentioning that the definition does not consider that the extension, as it happened for the Jacobi case, is provided along with a remapping of the spatial structure from regular to toroidal. We therefore introduce a specification of the semi uniformity definition for regular structures as follows.

**Definition 2.3.8 (Extension of the Semi Uniform Stencil Class for Regular Structures).** Let $\psi$ be an affine stencil, that is defined over an non toroidal spatial structure, and let $S^\psi_{\mathcal{M}_i}$ its shape set for the step $step_i$. The $step_i$ is classified as semi uniform if, after a remapping of the spatial structure on a toroidal space, there exists a set $U \subseteq N^n$, called uniformity set, such that:

$$\begin{align*}
\forall e_\alpha, e_\beta \in \mathcal{M}, \quad &\left( S^\psi_{(i,e_\alpha)} \cup (U + e_\alpha) \right) \triangleq \left( S^\psi_{(i,e_\beta)} \cup (U + e_\beta) \right) \\
\forall e \in \mathcal{M}, \quad & (U + e) \subset \mathcal{M}
\end{align*}$$

(2.8)

Stencil $\psi$ is classified as *semi uniform stencil* if all its steps are semi uniform; where each step can feature a different uniform set.
CHAPTER 2. THE STRUCTURED STENCIL MODEL

Figure 2.18: Representation of some application points and the corresponding shapes which are featured by the one dimension non toroidal Jacobi application whose description in pseudo code is reported in Figure 2.17.

Figure 2.19: Representation of some application points and the corresponding shapes which are featured by the extended one dimension non toroidal Jacobi application whose description in pseudo code is reported in Figure 2.17.
1D NON toroidal Jacobi (JCB) Step Model

\[ JCB = (W_{JCB}, T_{JCB}) \]

\[ W_{JCB} = ([0, 9], \mathbb{R}, M_{JCB}^i[:]) \]

\[ m_{JCB} = (10) \]

\[ T_{JCB} = \{ step_1^{JCB}, step_2^{JCB}, step_3^{JCB}, step_4^{JCB} \} \]

\[ C_{JCB} = [1, 5] \]

\[ \forall e = x \in M_{JCB} \xrightarrow{\text{step}_i} (F_{ie}, S_{ie}^{JCB}) \]

\[ S_{(x,y)}^{JCB} = \begin{cases} 
\{(x + 1)\} & \text{if } e \text{ is left border} \\
\{(x - 1), (x + 1)\} & \text{if } e \text{ is not border} \\
\{(x - 1)\} & \text{if } e \text{ is right border}
\end{cases} \]

\[ F_{(i,e)} : \begin{cases} 
\mathbb{R} \mapsto \mathbb{R} & \text{if } e \text{ is left border} \\
\mathbb{R}^2 \mapsto \mathbb{R} & \text{if } e \text{ is not border} \\
\mathbb{R} \mapsto \mathbb{R} & \text{if } e \text{ is right border}
\end{cases} \]

\[ M_{JCB}^{i+1}[e] = \begin{cases} 
M_{JCB}^i[(x + 1)] & \text{if } e \text{ is left border} \\
\frac{1}{2}(M_{JCB}^i[(x + 1)] + M_{JCB}^i[(x - 1)]) & \text{if } e \text{ is not border} \\
M_{JCB}^i[(x - 1)] & \text{if } e \text{ is right border}
\end{cases} \]

Table 2.9: Structured Step model of the 1D non toroidal Jacobi stencil (JCB) described in the pseudo code of Figure 2.17.
CHAPTER 2. THE STRUCTURED STENCIL MODEL

1D NON toroidal Jacobi (JCB) Step Model

\[ JCB = (W_{JCB}, T_{JCB}) \]

\[ W_{JCB} = \left( \frac{\mathbb{Z}}{10}, \mathbb{R}, M^i_{JCB}[.] \right) \]

\( m_{JCB} = (10) \)

\[ T_{JCB} = \{ step^1_{JCB}, step^2_{JCB}, step^3_{JCB}, step^4_{JCB} \} \]

\[ C_{JCB} = [1, 5] \]

\[ \forall e = x \in M_{JCB} \xrightarrow{step_i} (F_{ie}, S^{{JCB}}_{ie}) \]

\[ U = \{ +1, -1 \} \]

\[ S^{{JCB}}_{(x,y)} \cup (U + e) = \begin{cases} 
(x + 1) \cup \{(x + 1), (x - 1)\} & \text{if } e \text{ is left border} \\
(x - 1), (x + 1) \cup \{(x + 1), (x - 1)\} & \text{if } e \text{ is not border} \\
(x - 1) \cup \{(x + 1), (x - 1)\} & \text{if } e \text{ is right border} 
\end{cases} \]

\[ F_{(i,e)} : \begin{cases} 
\mathbb{R} \mapsto \mathbb{R} & \text{if } e \text{ is left border} \\
\mathbb{R}^2 \mapsto \mathbb{R} & \text{if } e \text{ is not border} \\
\mathbb{R} \mapsto \mathbb{R} & \text{if } e \text{ is right border} 
\end{cases} \]

\[ M^{i+1}_{JCB}[e] = \begin{cases} 
M^i_{JCB}[(x + 1)] & \text{if } e \text{ is left border} \\
\frac{1}{2}(M^i_{JCB}[(x + 1)] + M^i_{JCB}[(x - 1)]) & \text{if } e \text{ is not border} \\
M^i_{JCB}[(x - 1)] & \text{if } e \text{ is right border} 
\end{cases} \]

Table 2.10: Structured Step model of the extended 1D non toroidal Jacobi stencil (JCB) described in the pseudo code of Figure 2.17
2.4 An Extension of Space Invariant Stencils

It is beyond the scope of this Section to discuss, in an informal way, about some extensions to model also non space invariant stencils. We recall that we defined the space invariant stencils as ones that feature one input and one output data structures that are by the same index space. We showed that some stencil applications like Jacobi, Laplace, Red-Black and Floyd-Warshall belong to the space invariant stencil class.

We focus now on another application example, which is based on a reduce stencil. We consider a simple reduce operation that sum all the elements in a vector as shown in Figure 2.20.

In a generic $step_i$ the index space of the input data structure is wider with respect to the one of the output structure. Moreover when we study the step set $\mathcal{T} = \{step_1, step_2, step_3\}$ we find that no one step feature the same input index space. In order to model this kind of stencil an extension of the model is required.

In order to manage the reduce stencil example, the model has therefore to be consider a new definition of the working domain. As it is for the step set component, the working domain have to be redefined as a container of step working domains, each one for each step.

For instance, in the case of the reduce example we would have a working domain description like the following one

$$\mathcal{W}_{reduce} = \{ \mathcal{W}_{reduce}^{1}, \mathcal{W}_{reduce}^{2}, \mathcal{W}_{reduce}^{3} \}$$

Concerning each step working domain component, it can be described with a single parametric representation with respect to associated step index $i_{step}$:

$$\mathcal{W}_{i_{step}}^{reduce} = \left( M_{i_{step}}^{IN} = \left[ 0, \left( \frac{8}{2^{i_{step}} - 1} - 1 \right) \right], M_{i_{step}}^{OUT} = \left[ 0, \left( \frac{8}{2^{i_{step}} - 1} - 1 \right) \right], \mathbb{R}, M_{i_{step}}^{reduce}[\cdot] \right)$$

The re-definition of the working domain and especially the introduction in each single step working domain of two spatial structures, one for the input and one for the output, has a strong impact on the step stencil component. The main change is reflected on the fact that the shape and the application point are now defined over two different index spaces. The generic shape groups elements of the input structure while the application points are identified as element of the output data structure. A possible step model for the reduce application is reported in Table 2.11. In the representation we exploit the following re-definition of the evaluation map:

$$\mathcal{M}_{reduce}^i[e] = \mathcal{M}_{i}^{IN}[e], \forall e \in \mathcal{M}_{i}^{IN}$$

It is worth mentioning that from the definition of the steps of the reduced application, the shape description is independent from the step it belongs to, and moreover it is expressed as a linear transformation of the application point. Therefore, the reduce application should be classified as non space invariant and affine stencil.
The reduce application is just an example of non space invariant stencils. For instance we could mention the matrix multiplication algorithm. In this case, differently from the reduce one, we have to model also more than one input structure in the working domain: one for each of the for the two input matrices.

Because in rest of the thesis we concentrate on space invariant stencil, we consider what we have presented on non space invariant stencil a sufficient overview. We keep the formal description of an extended model, that can manage non space invariant stencil, to future works.
2.4. AN EXTENSION OF SPACE INVARIANT STENCILS

Figure 2.20: Representation of some application points and the corresponding shapes which are featured, during different steps, by a reduce application.

### REDUCE STEP MODEL

\[ \forall e = x \in M_i^{\text{OUT}} \xrightarrow{\text{step}_i^{\text{reduce}}} (F_{ie}, S_{ie}^{JCB}) \]

\[ S_{x}^{JCB} = \left\{ \forall \bar{e} \in M_i^{\text{IN}} | \bar{e} \in \{x, 2x\} \right\} \]

\[ F_{(i,e)} : \mathbb{R}^2 \rightarrow \mathbb{R} \]

\[ F_{(i,e)} : (r_1, r_2) \mapsto r_1 + r_2 \forall r_i \in \mathbb{R} \]

\[ M_i^{JCB}[e] = F_{(i,e)}(M_i^{\text{reduce}}[x], M_i^{\text{reduce}}[2x]) \]

\[ = F_{(i,e)}(M_i^{\text{IN}}[x], M_i^{\text{IN}}[2x]) \]

\[ = M_i^{\text{IN}}[x] + M_i^{\text{IN}}[2x] \]

Table 2.11: Step component of a reduce application in the structured stencil model.
2.5 A Specification of the Structured Model for HUA Stencils

We have seen in the previous Sections a model formalization and a deep classification of space invariant stencils. Now we would like to focus the discussion on a particular class called HUA. This class comes from the intersection of homogeneous and uniform ones. Because in the thesis the HUA stencils are going to play a fundamental role, we present a specialization of the structured model for HUA stencils, in order to better manage their distinguishing features.

2.5.1 The Definition of the HUA Model

Before we start the formal introduction of the HUA model, it is worth mentioning which are features of the HUA stencil class that we want to be explicitly represented.

I A HUA stencil belongs to the homogeneous class, thus its step set are composed by only equivalent elements.

II A generic HUA stencil belongs also to the affine class, therefore each element of a shape can be represented as affine transformation of the associated application point.

III Because a HUA stencil is classified as both uniform and homogeneous, we can claim that the same shape is independently associated to each element of the spatial structure and moreover the association is an invariant for all the stencil steps.

IV Because a HUA stencil belongs to the uniform class, that is a subclass of the affine family, we can assert that the rotation matrix of all the affine transformations, which are exploited to define the shape elements, is the identity matrix (see Corollary 2.3.1).

V Finally from Property 2.3.1, we know that a HUA stencil can be defined only over toroidal spatial structures.

We can now give the following formal definition:

Definition 2.5.1 (Homogeneous Uniform Affine Model). Let be a space invariant stencil belonging to the intersection of homogeneous and uniform classes.
The $HUA$ model represent one of the **homogeneous** steps of $\psi$ as it follows:

$$\forall e \in M \xrightarrow{\text{step}_i} (F, S)$$

$$S = \{g_1, g_e, \ldots, g_n | \forall \alpha g_{\alpha} = (Ie + \beta_{\alpha}) \in M\}$$

$$\Downarrow e + \{\beta_1, \beta_2, \ldots, \beta_n\}$$

$$\Downarrow e + R$$

$$M^{i+1}[e] = F(M^i[g_1], M^i[g_2], \ldots, M^i[g_n])$$

$$\Downarrow F(M^i[e + \beta_1], \ldots, M^i[e + \beta_n])$$

(2.9)

The matrix $I$ is the identity matrix, while the set $R_i = \{\beta_1, \ldots, \beta_n\}$ is called the relative shape.

In $HUA$ both the same function $F$ and the same relative shape $R$ are uniformly associated to all the domain elements independently from the considered step.

Because for uniform stencils the rotation matrix is equal to the identity matrix, we can claim that $R$ represents the minimum unit of information required to completely define the affine transformations for all the shape elements. The relative shape is the distinguishing contribution that is added by the $HUA$ model with respect to the more general structured one. The relative shape is going to be a tool to facilitate the proof of the $HUA$ stencil proprieties.

Previously in the Chapter, we analyzed two applications which belong to the $HUA$ class: Jacobi, and Laplace. We now would like to consider once again the Laplace application as an example of modelling with the new $HUA$ formalism. The Laplace $HUA$ model is reported in Table 2.12 and it can be compared with the general one reported in the Table 2.2 in Section 2.2. The new representation stresses the fact that all the steps are equivalent, that the step function does not depend by the specific domain element. Finally and foremost, the model represents a new prospective of the shape component: a rigid translation of the relative shape on the application point.

### 2.5.2 The Relaxed Computational Equivalence in the $HUA$ Model

In order to give a preview of how the relative shape can be exploited to prove $HUA$ stencil properties, we go back to the Introduction Chapter to recall the comparison of a mono dimensional Jacobi and a modified version. We report in Figure 2.21 the graphical representation of shapes and the application points of both the original and modified Jacobi version. We highlight once again that the only difference of the two stencils is given by the location of the application point with respect to the shape (compare Figure 2.21(a) and Figure 2.21(b)).
CHAPTER 2. THE STRUCTURED STENCIL MODEL

\[ \text{2D Laplace (LPC) HUA Model} \]

\[ \begin{align*}
LPC &= (W_{LPC}, T_{LPC}) \\
W_{LPC} &= \left( \frac{Z}{1024} \times \frac{Z}{1024}, \mathbb{R}, M^i_{LPC}[\cdot] \right) \\
m_{LPC} &= (1024, 1024) \\
T_{LPC} &= \{ \text{step}^{LPC}, \text{step}^{LPC}, \text{step}^{LPC}, \text{step}^{LPC} \} \\
C_{LPC} &= [1, 5]
\end{align*} \]

\[ \forall e = (x, y) \in M_{LPC} \xrightarrow{\text{step}^{LPC}} (F_e, S^i_{LPC}) \]

\[ \begin{align*}
R^{LPC} &= \{ (0, 0), (0, +1), (0, -1), (+1, 0), (-1, 0) \} \\
S^i_{LPC} &= e + R^{LPC} \\
F_e &: \mathbb{R}^5 \mapsto \mathbb{R} \\
&\quad (x_1, x_2, x_3, x_4, x_5) \mapsto \frac{\sum_{u=1}^{5} x_u}{5} \\
M^i_{LPC}[e] &= \frac{1}{5} \left( M^i_{LPC}[e + (0, 0)] + M^i_{LPC}[e + (0, +1)] + M^i_{LPC}[e + (0, -1)] + M^i_{LPC}[e + (+1, 0)] + M^i_{LPC}[e + (-1, 0)] \right)
\end{align*} \]

\textbf{Table 2.12:} HUA model of the Laplace (LPC) stencil application that is described in pseudo code in Figure 2.6
2.5. A SPECIFICATION OF THE STRUCTURED MODEL FOR $\texttt{HUA}$ STENCILS

Figure 2.21: Representations, in a one dimension space, of both Jacobi stencil (fig. 2.21(a)) and its modified version (fig. 2.21(b)), where the application point has been shifted of one position. Example of one step computation of the original (fig. 2.21(a)) and the modified Jacobi (fig. 2.21(b)) over a circular vector representing a toroidal domain. Finally a parallelization of the original (fig. 2.21(e)) and modified (fig. 2.21(f)) Jacobi to highlight the different communication patterns.

We exploited this example in the Introduction Chapter to present the concept of relaxed equivalence. Indeed, from the analysis of a single step computation that is performed, on the same input, by both the two Jacobi version (compare Figure 2.21(b) and Figure 2.21(d)), it comes out that the two results are equivalent except for a redistribution of the values in the data structures. More precisely we find that if with the original stencil an output value is stored at position $i$, in the modified version the same value is stored at position $i - 1$. We now focus on the exploitation of the relative shape entity in order to prove that the stencil shapes of the two version are linked by the same relation that exist between the resulting vectors.

Table 2.13 and Table 2.14 report the $\texttt{HUA}$ model of the two Jacobi versions. The impact of the different application points is reflected in the two different relative shapes. The same effect was also represented in the general model, nevertheless the $\texttt{HUA}$ captures in a better and concise way, though the relative shape entity, the peculiar nature of the two shape differences: the shapes are equivalent except for a
rigid translation. Indeed, exploiting \( R^{Jo} \) and \( R^{Jm} \) the two relative shapes, we can demonstrate that:

\[
S_i^{Jm} - 1 = (x + \{0, +2\}) - 1 = x + \{ +1, -1\} = S_i^{Jo}
\]

\[
S_i^{Jo} + 1 = (x + \{ +1, -1\}) + 1 = x + \{0, +2\} = S_i^{Jm}
\]

The rigid translation that links the two relative shapes is the same that links the two resulting vectors of Figure 2.21(c) and Figure 2.21(d). The presented proof can be considered as a preview of what we are going to present in the next Chapter.

We conclude stressing that there are no expressiveness differences between the \( \mathcal{HUA} \) and the general model presented in the previous Chapter. What we have previously proved, it could be also done exploiting the general model, but it would require a more complex formalism.

In order to show the potentiality of the relative shape we purposely chose the original and modified Jacobi example. The aim was to focus the discussion, once again, on the concept of relative equivalence. Indeed we want to formalize the concept in the \( \mathcal{HUA} \) model as follows.

**Definition 2.5.2 (Relaxed Computational Equivalence in the \( \mathcal{HUA} \) Model).**

Let \( \psi \) and \( \chi \) be two stencil based computations which are expressed in the \( \mathcal{HUA} \) model. Let then \( M_{\psi} \), the spatial structure of \( \psi \), and \( M_{\chi} \), the spatial structure of \( \chi \), be equivalent. A step \( \text{step}_i \) of \( \psi \) and \( \text{step}_j \) of \( \chi \) are defined relaxed equivalent if a linear transformation \( \phi \), featuring the identity as rotation matrix, exists such that:

\[
\forall e \in M_{\psi} \land e \in M_{\chi}, M_{\psi}^{i}[e] = M_{\chi}^{j}[e] \Rightarrow M_{\psi}^{i+1}[e] = M_{\chi}^{j+1}[\phi(e)]
\]

The two \( \mathcal{HUA} \) stencils \( \psi \) and \( \chi \) are relaxed equivalent if all their steps are relaxed equivalent. Finally, a stencil transformation that result in a stencil that is relaxed equivalent with the original one is classified as relaxed safe.

Informally, two steps, that features the same index space, are relaxed equivalent if, from the same input working domain, they return two output working domains that feature the same values but which can possible stored in different positions.

From one of the output working domain it is possible to reconstruct the other, knowing the linear transformation that is associated to the relaxed equivalence.
2.5. A SPECIFICATION OF THE STRUCTURED MODEL FOR $\texttt{HUA}$ STENCILS

---

### 1D Original Jacobi ($Jo$) $\texttt{HUA}$ step model

**Definition:**

\forall e = (x) \in M_{Jo} \xrightarrow{\text{step}^{Jo}} (F_e, S^{Jo}_e)

- $R^{Jo} = \{ (+1), (-1) \}$
- $S^{Jo}_e = e + R^{Jo}$
- $F_e : \mathbb{R}^5 \mapsto \mathbb{R}$
  \[
  (x_1, x_2) \mapsto \frac{\sum_{u=1}^{2} x_u}{2}
  \]
- $M^{i+1}_{Jo}[e] = \frac{1}{2} \left( M^i_{Jo}[e + 1] + M^i_{Jo}[e - 1] \right)$

| Table 2.13: $\texttt{HUA}$ model of the Laplace ($LPC$) stencil application that is described in pseudo code in Figure 2.6 |

---

### 1D modified Jacobi ($Jm$) $\texttt{HUA}$ step model

**Definition:**

\forall e = (x) \in M_{Jm} \xrightarrow{\text{step}^{Jm}} (F_e, S^{Jm}_e)

- $R^{Jm} = \{ (0), (-2) \}$
- $S^{Jm}_e = e + R^{Jm}$
- $F_e : \mathbb{R}^2 \mapsto \mathbb{R}$
  \[
  (x_1, x_2) \mapsto \frac{\sum_{u=1}^{2} x_u}{2}
  \]
- $M^{i+1}_{Jm}[e] = \frac{1}{2} \left( M^i_{Jm}[e + 1] + M^i_{Jm}[e - 2] \right)$

| Table 2.14: $\texttt{HUA}$ model of the Laplace ($LPC$) stencil application that is described in pseudo code in Figure 2.6 |
Chapter 3
The $\mathcal{HUA}$ Complete Architecture

Abstract

In this Chapter we analyze the complete architecture of a framework for programming $\mathcal{HUA}$ stencil based applications. The software architecture that we refer to is composed by four levels called functional dependency, partition dependency, concurrent, and firmware level. In the functional dependency level a stencil based application is represented in terms of the structured model, where the functional dependency are highlighted. In the partition dependency level, the $\mathcal{HUA}$ stencil description is translated in terms of space partitions. In the concurrent level a representation of the stencil in terms of communicating processes is extracted. Finally in firmware level the program is compiled for the specific architecture.

The Chapter is structured as follows. Section 3.3 presents our reference architecture while Section 3.2 and Section 3.3 analyse in detail respectively the partition dependency and the concurrent levels.

Finally, Section 3.4 analyzes a particular stencil application through each of the architecture levels. The Section introduces the best methods in literature to optimize the number of communications in the implementation of a $\mathcal{HUA}$ stencil.
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Reference Architecture</td>
<td>73</td>
</tr>
<tr>
<td>3.2</td>
<td>The World of Partitions</td>
<td>75</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Working Hypotheses on Partitioning Strategies</td>
<td>75</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Conventions and Annotations for Partitions</td>
<td>76</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Regions: the Classification of Partition Elements</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>The World of Concurrency</td>
<td>83</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The $\mathcal{L}C$ language</td>
<td>83</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Communication Cost Model</td>
<td>84</td>
</tr>
<tr>
<td>3.3.3</td>
<td>$\mathcal{L}C$ and $\mathcal{HUA}$ Stencils</td>
<td>84</td>
</tr>
<tr>
<td>3.4</td>
<td>A Nine Point Stencil at Work</td>
<td>86</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Functional Dependency Level</td>
<td>86</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Partition Dependency Level</td>
<td>87</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Concurrent Level: The <em>naive</em> Method</td>
<td>89</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Concurrent Level: The <em>shift</em> Method</td>
<td>92</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Environments of the Experimental Tests</td>
<td>95</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Experimental Results</td>
<td>96</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Conclusions</td>
<td>102</td>
</tr>
</tbody>
</table>
3.1 The Reference Architecture

In the previous Chapter we presented the stencil structured model for a certain class of stencil based applications. The model provides some mechanisms to programmers in order to describe their applications. Those mechanisms alone are not sufficient to produce a program that can be run on a target parallel architecture.

A stencil representation, in terms of the structured model, features a high abstraction degree with respect to an executable program described at firmware level. The abstraction gap therefore has to be filled up with a process of concretization, where each mechanism of high degree abstraction is implemented with mechanisms featuring a lower abstraction degree.

We relay on a concretization method based on a hierarchic level approach. Indeed, we consider an architecture composed by different levels of abstractions. Each level features its own set of mechanisms and its own language to manage the mechanisms. The concretization strategy consists in the transformation of a program described at a certain level into an equivalent one described at the level immediately below.

Iterating the concretization strategy level by level, the high level description of the stencil based application, from which we can easily study useful properties for optimization porpoise, is transformed into an equivalent executable program expressed at the firmware level.

Figure 3.1 shows our four level reference architecture. The process of concretization for a generic stencil application can be schematized as follows:

![Figure 3.1: Representation of our reference architecture for implementing HUA stencils.](image)

At the top of our architecture we face the functional dependency level,
which features a language that, helpfully with some nice syntax, allows programmers to define a stencil application according to the structured stencil model.

At this level a description of a stencil program is mainly focused on the functional dependencies between spatial elements, described in terms of shapes and application points, hence the name of the level.

The mechanisms featured by functional dependency level are the ones that we described in the previous Chapter while presenting the structured stencil model: steps, working domains, shapes, application points etc.

II Once a description of the application has been established according to the structured stencil model, it is translated into the partition dependency level. At this level a partition strategy for the elements of the spatial structure is selected and all the dependencies between spatial elements are translated into dependencies between partitions.

III At the concurrent level, a process is associated to each partition. Exploiting a message passing formalism, the partition dependencies are resolved with appropriate communications between processes. Furthermore, the data structures to represent the working domain are selected.

IV Finally the program described in a message passing language can be directly compiled into the firmware level in order to obtain an executable program for a target parallel architecture.

The representation of the architecture in Figure 3.1 well highlights where the programmers of a stencil based application are involved. Once they have produces a stencil description in the structured stencil model, they do not have to manage any aspects of the concretization phase. A set of tools, that we call framework support, can be defined to automatically produce the executable program of the application.

With respect to the presented architecture, our studies focus the analysis of the impacts that properties at the functional dependency level produce at the concurrent one. In particular the scope behind the following Sections is to understand how the functional dependencies between elements influence the number of communications between processes. Moreover, the next Chapter extends the provided result in order to define some new transformations at the dependency level can reduce the communication overheads.
3.2 The World of Partitions

A delicate phase of a data parallel application development is the partitioning. Important performance aspects of a parallel program, like the degree of parallelism, the number of communications and also their sizes, depend on the specific partitioning strategy. This is true both for distributed architecture and shared memory, where a communication has to be considered as read or write operations.

For general data parallel applications, the problem of defining a partitioning that maximises the performance is a NP problem [24, 26].

In the case of HULA space invariant stencils, where all the computations are focused on one single spatial structure and functional dependencies have a well defined geometric structure, we confine our self to regular partitioning.

In Section 3.2.1 we formally define some working hypotheses on the partitioning strategies, and in Section 3.2.2 we present some conventions and annotations that we exploit to manage partitions.

Finally in Section 3.2.3 we classify the elements of a partition in regions with respect to a target stencil. Moreover we formalize some mechanisms to represents partition dependencies.

3.2.1 Working Hypotheses on Partitioning Strategies

In our study we consider two hypotheses on partitioning. The first one concerns a restriction on the types of regular partitioning that we focus on. The second one introduces some limitations on the features of the resulting partitions, with respect to a target stencil. We start considering the restrictions on partitioning strategies.

**Working Hypothesis 3.1.** In our study we confine our self only on regular and block partitioning strategies with the following bounds.

I. The partitioning concerns all the space dimensions.

II. All the resulting partitions feature the same space volume.

It is well known that sometimes it is preferable, for performance reason, not to consider all the dimensions of a working space for the partitioning [49]. Taking again the case of the two dimension Laplace stencil, the previous sentence claims that there are cases in which it can be convenient to split the matrix by rows instead of by blocks. In other words, better performance can be reached when the partitioning of the row is avoided. This kind of strategies are not taken into account by the Working Hypothesis 3.1.

It is worth mentioning that if the first approach is convenient with respect to the second one or if it is the other way round, it depends on different parameters like the mean computation latency associated to the update of a domain element value and the parallel degree we are targeting. A simple model for this problem, in the
case of a two dimension working domain, is presented by Wilkinson and Allen [49]. The result is that for high values of parallel degree it is preferable to partition over all the dimensions.

In our study we keep focusing only on this last case, because it is the most complicated for the communication pattern and consequently it is more interesting as studding case. Moreover our results can be directly reused implicitly defining the computation in a space with a lower number of dimensions. For instance, it is possible to model the two dimension Jacobi over a one dimension space where the type of an spatial element is equivalent to a row of the original problem.

Along with the previous working hypothesis about regular partitioning we take in consideration another bound about the volume of a partition.

**Working Hypothesis 3.2.** Consider an application described by a $HUA$ stencil $\psi$. In our study we consider only those partition strategies resulting in partitions whose volume is wider than the one of the relative shape $R$ associated to $\psi$.

In the example of the Laplace based application, the previous working hypothesis imposes that the generic partition provides a surface wider than the four by four one which circumscribe the relative cross shape.

The restriction is realistic, in the sense that the introduced bound is respected by the real stencil based applications for two main reasons.

I Working domains are usually extremely wider with respect to the volume of the relative shape of a stencil. Let consider a partitioning which results in partitions whose volume is equal to the one of the stencil relative shape. In a such configuration the number of resulting partition would be extremely high.

II Foremost, because the computation for a single element features in most cases a low latency, with too small partitions there would no chance of overlapping between communications and computations; this would result in a low efficiency.

### 3.2.2 Conventions and Annotations for Partitions

In the analysis of communication impacts in stencil based applications, the partitions play a fundamental rule. To make discussions on spatial properties of a partition we introduce in this Section some useful annotations and conventions.

Considering the previous Working Hypothesis 3.1 we can give a formal definition of the partition space:

**Definition 3.2.1 (Partition Space).** Let $\psi$ be a space invariant $HUA$ stencil defined over a $n$ dimension space, and let $W^\psi$ be its working domain. The partition space $P$ of $W^\psi$ is a sub set of $\mathbb{N}^n$ where each point represents one working domain partition:
3.2. THE WORLD OF PARTITIONS

The vector $p = (p_1, \ldots, p_n)$ is the partition space length vector. The $i$-th component of $p$ defines how many partitions have been considered along the $i$-th space dimension. The generic element $P_\alpha \in \mathcal{P}$ represents a sub set of the spatial structure according to its position in $\mathcal{P}$.

The partitioning and the functional dependencies between elements defined by a stencil produces what we call partition dependencies. The effects of this dependencies is a mandatory information exchange between partitions in order to update all the elements during a single step.

Fixed a partition $P_\alpha$, the set of other partitions that store necessary information in order to update all the element of a $P_\alpha$ depends both on the partition strategy and on the considered stencil shape. Considering the Working Hypothesis 3.1 and Working Hypothesis 3.2 we derive the following important property.

Property 3.2.1 (Neighbour Interactions). A generic partition $P_\alpha$ can feature partition dependencies only with its spatial neighbour partitions.

We therefore need some mechanisms to identify in the partition space the neighbours of a partition $P_\alpha$. For the sake of simplicity, we adopt a relative reference
system centered in $P_\alpha$, in this system partitions are represented by vectors centered in $P_\alpha$. Figure 3.2 shows the partition reference system for a two dimensional working domain.

We can formally define the neighbour set of a partition as follows.

**Definition 3.2.2 (The Neighbour Set).** Let $P$ be a partitions space and let $P_\alpha$ be a generic partition in the space. Let we consider now a relative reference system centered in $P_\alpha$. In the new system, we define the neighbour index set of $P_\alpha$ the following vector set:

$$\text{NI}(P_\alpha) = \{\forall (\beta_1, \ldots, \beta_n) | \forall i \beta_i \in \{-1, 0, 1\} \text{ and } \exists j | \beta_j \neq 0\}$$

We call the generic element of the neighbour index set the movement vector. Associating to each index the corresponding neighbour partition, we define the neighbour set as follows:

$$\text{NS}(P_\alpha) = \{\forall P((\beta_1, \ldots, \beta_n)) | (\beta_1, \ldots, \beta_n) \in \text{NI}(P_\alpha)\}$$

The neighbour index set groups all the space vectors that reference the neighbours of the origin of $\mathbb{Z}$, the set of integer numbers. The elements of the set are then used in the neighbour set to identify the neighbour partitions in the relative system centered in the generic partition $P_\alpha$.

For a $n$ dimensional working domain, we can claim that the cardinality of $\text{NS}(P_\alpha)$ is equal to $3^n - 1$. Indeed, the neighbour index set, which has the same cardinality of the neighbour set, is composed by all combination of strings of length $n$ that can be composed exploiting the alphabet $\{-1, 0, 1\}$, apart from the string composed by all zero. The zero string is associated to $P_\alpha$ which does not belong to $\text{NS}(P_\alpha)$.

### 3.2.3 Regions: the Classification of Partition Elements

We know that, when a partition strategy is selected, the functional dependencies between spatial elements are translated into partition dependencies. In this section we focus on the analysis of this transformation defining a classification of the partition elements in distinct regions.

A first subdivision of a partition can be computed with respect to the functional dependencies of its local elements. We therefore consider the logical splitting of partition elements into two regions called **incoming independent region** and **incoming dependent region**. The first one is define as follow.

**Definition 3.2.3 (Incoming Independent Region).** Let $P_\alpha$ be a generic partition of a working domain and let $\psi$ be a generic HUA stencil. We define the incoming independent region (IIR) of $P_\alpha$ associated to $\psi$ the following set:

$$\text{IIR}(P_\alpha) = \{\forall e_{\text{local}} \in P_\alpha | e_{\text{remote}} \in S^\psi(e_{\text{local}}) \Rightarrow e_{\text{remote}} \in P_\alpha\}$$
3.2. THE WORLD OF PARTITIONS

Less formally, the region includes all those elements whose associated shape is completely composed by elements of the partition. Therefore the elements of the incoming independent region do not require any other information from the outside world (represented as the elements of all the other partitions), hence the name we selected for the region.

Figure 3.3(a) reports the representation of a stencil that we use to graphically show which are the regions of a partition. On purpose, we selected a such uncommon stencil, whose shape does not feature any symmetry with respect to the application point, in order to avoid possible erroneous conclusions.

Figure 3.3(a) highlights in white the elements of a general partition that, according to the considered unusual stencil, belong to the incoming independent region. What we can see from the Figure is that the incoming independent region is a convex set. It easy to prove that the observation can be extended to all the HUA stencils.

The second part in which we logically divide a generic partition is characterized as follow:

**Definition 3.2.4 (Incoming Dependent Region).** Let \( P_\alpha \) be a generic partition of a working domain and let \( \psi \) be a generic HUA stencil. We define the incoming dependent region (IDR) of \( P_\alpha \) associated to \( \psi \) the following set:

\[
IDR(P_\alpha) = \{ \forall e_{local} \in P_\alpha \mid \exists e_{remote} \in S_\psi(e_{local}) \text{ and } e_{remote} \notin P_\alpha \}
\]

The region, which is the complementary to the incoming dependent one with respect to the partition space, collects elements that, in order to be updated according to the stencil shape, require values of some remote elements, i.e. elements that are located in other partitions. Figure 3.3(b) highlights dependent incoming region elements in grey.

As it is evident from the Figure, differently from the incoming independent region, the new one is not a compact set. Instead, the region features the properties of being gathered next to the partition borders, or in other words that at least one of the borders of the region is also the border of the partition.

We formally model and prove the previous property with the following Theorem:

**Theorem 3.2.1 (Incoming Dependent Region Placement).** Let the element \( e_{local} \) belong to the incoming dependent region of a partition \( P_\alpha \) with respect to a stencil \( \psi \). Let \( \delta \) be one of the partition borders that the \( \psi \) shape intersects when it is applied to \( e_{local} \). All partition points between \( e_{local} \) and \( \delta \) belongs to the incoming dependent region.

**Proof.** First of all the existence of \( \delta \) is ensured by the fact that the element \( e_{local} \) belongs to the incoming dependent region. In fact by definition of incoming dependent region when one of its element is considered as application point, the associated shape features at least one element that does not belong to the partition. We can conclude that the shape has to intersect at least one of the partition border. We
Figure 3.3: Graphical representation of stencil shape (3.3(a)), incoming dependent (in gray color) and independent regions (3.3(b)) and finally outgoing dependent (in gray color) and independent regions (3.3(c))
3.2. THE WORLD OF PARTITIONS

have therefore demonstrated the correctness of theorem statement, we now proceed proving its the validity.

The key point of the proof is given by the uniformity property of HUA stencils. Consider \( e^{\text{remote}} \) one of the elements that belongs to the shape applied to \( e^{\text{local}} \) but does not belong to the partition \( P_\alpha \); in another words \( e^{\text{local}} \) features a functional dependency with respect to \( e^{\text{remote}} \). We can assert that all the elements of \( P_\alpha \) between \( e^{\text{local}} \) and \( e^{\text{remote}} \) belong to the incoming dependent region. Suppose \( e_m \) is one of those elements. If the shape of the stencil applied to \( e^{\text{local}} \) features elements out of the partition bound, the same shape applied to \( e_m \), which is closer to the partition border by construction, can not be completely inside the partition. If the shape centered in \( e_m \) have also elements that do not belong to its partition, \( e_m \) is classified as element of the incoming dependent region.

It is worth mentioning that another main information comes from the incoming dependent region. The elements of the region define a set, let us call it the incoming set, composed by non local elements. The set is the union of the remote elements of all the shapes whose application point belongs incoming dependent region.

The incoming set models the non local information that a generic partition has to acquire in order to update all its elements. The way the set is scattered between the neighbour partitions defines what we call the set of the incoming partition dependencies. A formal definition follows.

**Definition 3.2.5 (The Incoming Partition Dependency Set).** Let \( \psi \) be a HUA stencil, and let \( NS(P_\alpha) \) the set of neighbours of a generic partition \( P_\alpha \). We define, The incoming partition set \( \Delta^\text{in} \) is defined as the following sub set of \( NS(P_\alpha) \):

\[
\Delta^\text{in} = \{ \forall P_\beta \in NS(P_\alpha) | \exists e^{\text{local}} \in P_\alpha, e^{\text{remote}} \in P_\beta \text{ and } e^{\text{remote}} \in S^\psi(e^{\text{local}}) \}
\]

Orthogonally with respect to the two previous regions, which have been characterized according to dependencies versus element of other partitions, we classify the partition into other two regions: outgoing dependent and outgoing independent.

**Definition 3.2.6 (Outgoing Dependent Region).** Let \( P_\alpha \) be a generic partition of a working domain and let \( \psi \) be a generic HUA stencil. We define the outgoing dependent region (ODR) of \( P_\alpha \) associated to \( \psi \) the following set:

\[
\text{ODR}(P_\alpha) = \{ \forall e^{\text{local}} \in P_\alpha | \exists e^{\text{remote}} \in S^\psi(e^{\text{local}}) \text{ and } e^{\text{remote}} \notin P_\alpha \}
\]

The region, that it is highlighted in Figure 3.3(c) with a gray color, groups elements that are required by other partitions to complete the update of all their elements. The value of each element of this region has to be provided to one or more neighbour partitions. The second region is characterized as follows.
Definition 3.2.7 (Outgoing Independent Region). Let $\mathcal{P}_\alpha$ be a generic partition of a working domain and let $\psi$ be a generic HUA stencil. We define the outgoing independent region (OIR) of $\mathcal{P}_\alpha$ associated to $\psi$ the following set:

$$OIR(\mathcal{P}_\alpha) = \{ \forall e_{\text{local}} \in \mathcal{P}_\alpha \mid \forall e_{\text{remote}} \notin \mathcal{P}_\alpha \Rightarrow e_{\text{local}} \notin S^\psi(e_{\text{remote}}) \}$$

This region is complementary to the outgoing dependent one with respect to the partition space. It grabs all the elements whose values are not going to be exchanged with any other partition. As the incoming independent region, this region is a compact set. The region is highlighted with a white color in Figure 3.3(c).

A main information is extracted from the outgoing dependent region. The way the elements of the region are requested by neighbour partitions defines what we call the set of the outgoing partition dependencies. A formal definition follows.

Definition 3.2.8 (The Outgoing Partition Dependency Set). Let $\psi$ be a HUA stencil, and let $\mathcal{N}(\mathcal{P}_\alpha)$ the set of neighbours of a generic partition. We define $\Delta^{\text{out}}$, the outgoing partition set, as follows:

$$\Delta^{\text{out}} = \{ \forall \mathcal{P}_\beta \in \mathcal{N}(\mathcal{P}_\alpha) \exists e_{\text{remote}} \in \mathcal{P}_\beta, e_{\text{local}} \in \mathcal{P}_\alpha, e_{\text{local}} \in S^\psi(e_{\text{remote}}) \}$$

From the comparison of Figure 3.3(b) and Figure 3.3(c), it is evident that the dependent incoming region and the dependent outgoing one are composed by different partition elements. This observation can be symmetrically reported for the other two remaining regions.

We will see in the next Section that for HUA stencils whose shapes feature a central symmetry with respect to the application point, the incoming and outgoing dependent regions coincide.
3.3 The World of Concurrency

A stencil based application at the concurrent level is defined by a set of communicating processes. The partition dependencies defined by $\Delta^{in}$ $\Delta^{out}$, the incoming and the outgoing partition dependency set, are transformed in or better resolved with communications between processes.

In this section we first describe $\mathcal{LC}$, which is the language we target for the concurrent language, then we present a cost model associated to the language that allows us to estimate the performance of a program especially for the communication impacts. Finally, we present some conventions and annotations that we use when working at the concurrent level.

3.3.1 The $\mathcal{LC}$ language

$\mathcal{LC}$ is the language we defined for the concurrent level; it is based on a message passing paradigm and describes abstract processes communicating according to a local environment model. $\mathcal{LC}$ was born as a tailored minimal formalism of the CSP of Hoare [21].

A $\mathcal{LC}$ parallel program statically declares a set of processes, that can also be parametrically described with respect to some identifiers. A in deep description of $\mathcal{LC}$ language and the presentation of an implementation for the Cell multi-core architecture are the issue of Appendix A.

A $\mathcal{LC}$ channel is unidirectional and typed: it represents a queue where a set of producer processes insert messages and a single consumer process extracts data. The type of the channel and the type of the messages in its queue must match. Channels can implement either a symmetric communications, when only one producer is defined, or asymmetric communications.

An important characteristic of a $\mathcal{LC}$ channels is the asynchrony degree, which is a static integer parameter defining the maximum number of non blocking send operations that can be performed when no receive operation is invoked. When the parameter is equal to zero, the channel is synchronous by definition. The asynchrony degree is statically defined and has to be guaranteed by the implementation of the language for the life time of the $\mathcal{LC}$ program.

In order to target high performance computing, an important aspect is the possibility of featuring communication mechanisms that can overlap communications and computations. This depends from the characteristics of the target architecture and from the implementation of the communication mechanisms. As it is proved by the experiments shown in Chapter A, our $\mathcal{LC}$ implementation on the Cell multi-core architecture, that is called Mammu$T$, guarantees the possibility of exploiting the best overlapping that the firmware architecture provide.

Finally $\mathcal{LC}$ includes an alternative guarded command, similar to the one of CSP, to manage non deterministic aspects; this is a relevant key construct for dynamic load balancing aspects.
3.3.2 Communication Cost Model

Associated to the concurrent level we consider a cost model. The aim of the model is to wrap the performance feature of a target firmware architecture in a few parameters. This approach allows an easy analytic analysis of the performance associated to a LC parallel program without requiring the knowledge of the specific firmware architecture.

We consider a simple and wide used cost model for communications. We model the communication latency of a send operation as a constant setup time plus a transfer time that is proportional to the length of the message. For a single send operation we associate the following parameter:

\[ T_{\text{send}}(msg) = t_{\text{setup}}^{\text{snd}} + t_{\text{transm}} \times s_{\text{msg}} \]

The \( msg \) is the message and \( s_{\text{msg}} \) its size. Moreover \( t_{\text{setup}}^{\text{snd}}, t_{\text{transm}} \) are two constants that depend on the specific physical architecture. In our analysis we consider the two constant as unknown variables of the system. In other words all our considerations on communication impacts are going to be parametric with respect to both \( t_{\text{setup}}^{\text{snd}}, t_{\text{transm}} \).

The model has to consider also communication and computation overlapping aspects. Indeed, for certain implementations, a part of the whole communication latency can be possible partially overlapped to the computation. For the sake of simplicity we suppose that in the defined \( T_{\text{send}}(msg) \) the component \( t_{\text{transm}} \times s_{\text{msg}} \) exactly corresponds to the latency that can completely overlapped to some computations.

Finally, concerning the receive operations we model their latency as a constant that is independent from the message size:

\[ T_{\text{recv}}(msg) = t_{\text{setup}}^{\text{recv}} \]

This approach is well suited for those implementations of a message passing communication language which guarantee a zero-copy communication protocols. A discussion about the relation between the expressiveness of communication languages and implementation featuring zero-copy protocols is reported in Appendix A.

3.3.3 LC and \( \mathcal{HUA} \) Stencils

When implementing a \( \mathcal{HUA} \) stencil based application three are the phases that has to be considered.

I Data are distributed in such a way that each process acquires its own data partition.

II Partition dependencies are resolved though inter processes communications and each process updates its element values.
III Final values are gathered to rebuild the result data structures.

The focus of our studies is on the second of the three phases. We are interesting into understand the possible ways in which we can transform, as most efficiently as possible, mechanism of the functional dependency level into mechanisms of the concurrent level.

way to transform in the most efficient way mechanisms of the functional level in mechanism of the concurrent level.

For the sake of simplicity we refer to processes exploiting the name of the their own partition. Therefore with $P_\alpha$ we identify both the partition and the process managing it.

Granted that, our studies at the concurrent level mainly focuses an the definition and the analysis of the incoming communication set $\Delta_{in}^{\text{com}}$ and the incoming communication set $\Delta_{out}^{\text{com}}$. The two sets define for a generic process $P_\alpha$ which are the neighbour processes interested in respectively incoming and outgoing communications in order to resolve the partition dependencies. Because partitions and processes share the same name we can claim the following relations:

\[
\Delta_{in}^{\text{com}} \subseteq \Delta_{in}^{in}
\]
\[
\Delta_{out}^{\text{com}} \subseteq \Delta_{out}^{out}
\]
3.4 A Nine Point Stencil at Work

In this Section we concentrate on the analysis of a particular stencil example defined over a two dimension space that in literature is usually called nine point stencil (for sake of simplicity we call it Nine). This stencil is useful when studying communication impacts, because it can be considered as a worst case test. A representation of a sequential application based on the nine point stencil is reported in pseudo code in Figure 3.5.

In the following, we are going to focus the discussion on the analysis of the nine point stencil application development, from the functional level to the concurrent one, as shown by Figure 3.4. The Figure also highlights for each architectural level the mechanisms we are going to define and study.

3.4.1 Functional Dependency Level

At the level of functional dependencies, the application is described by programmers in the \( \mathcal{HUA} \) model, and therefore the dependencies between spatial elements are defined. The \( \mathcal{HUA} \) description of the nine point stencil application is straightforward. Figure 3.1 reports the model of the step component, while Figure 3.6(a) shows a graphically representation of the stencil shape geometry.
3.4. A NINE POINT STENCIL AT WORK

```c
double J_in[1024][1024], J_out[1024][1024];
load_working_domain_values(J_in);
for (i_step = 0; i_step < 5; i_step++) {
    forall((x, y) ∈ J_in) {
        J_out[x, y] = (J_in[x, y + 1] + J_in[x, y - 1] +
                      J_in[x + 1, y] + J_in[x + 1, y + 1] +
                      J_in[x + 1, y - 1] +
                      J_in[x - 1, y] + J_in[x - 1, y + 1] +
                      J_in[x - 1, y - 1])/8;
    }
    swap(J_in, J_out);
}
return_working_domain(J_out);
```

Figure 3.5: Representation in pseudo code of a nine point stencil (Nine) based application on a two dimensional toroidal space. To keep the annotation light we suppose the indexes are automatically mapped on the toroidal space, i.e. \( J_{out}[-1, +1] \) is transformed into \( J_{out}[+1023, +1] \)

3.4.2 Partition Dependency Level

At the partition dependency level, a \( \mathbb{HUA} \) stencil description in terms of functional dependencies between elements is translated into a description where the main objects are the partitions. Without the programmer interventions, functional dependencies are automatically re-described in terms of partitions, regions, incoming and outgoing partition dependency sets.

Figure 3.6(b) highlights, in a generic partition of the nine point stencil, the incoming independent and dependent regions; the first one is colored in white while the second one in gray. Instead, Figure 3.6(c) highlights the outgoing dependent and independent regions, respectively in gray and in white.

In the case of the nine point stencil, the incoming and outgoing dependent regions are gathered next to all the partition bounds. Moreover, because the stencil shape features a central symmetry with respect to the application point, we have that the two regions are perfectly overlapped, i.e. they are composed by the same partition elements.

Finally, it is worth mentioning that all the neighbour partitions are included into both the incoming and outgoing partition dependency sets:

\[
\Delta^{in} = \Delta^{out} = NS(P_\alpha)
\]

Therefore for the nine point stencil, a generic partition has to both acquire and provide some information form and to all its neighbours. This is the characteristic that makes the nine point stencil a useful worst case example in the studies of communication impacts.
Figure 3.6: Graphical representation of stencil shape (3.6(a)), incoming dependent (in gray color) and independent regions (3.6(b)) and finally outgoing dependent (in gray color) and independent regions (3.6(c)) of a nine point stencil.
3.4. A NINE POINT STENCIL AT WORK

\[ \forall e = (x, y) \in M_{\text{nine}} \xrightarrow{\text{step}_{\text{nine}}} (F, S_{\text{ie}}^{\text{nine}}) \]

\[ R_{\text{nine}} = \{(x, y + 1), (x, y - 1), (x + 1, y - 1), (x + 1, y), (x + 1, y + 1), (x - 1, y - 1), (x - 1, y), (x - 1, y + 1)\} \]

\[ S_{(x,y)}^{\text{nine}} = (x, y) + R_{\text{nine}} \]

\[ F : \mathbb{R}^4 \mapsto \mathbb{R} \]

\[ F : (r_1, r_2, r_3, r_4) \mapsto \sum_{i=1}^{8} \frac{r_i}{8} \]

\[ M_{\text{nine}}^{i+1}[e] = F(M_{\text{nine}}^{i}[(x, y + 1)], M_{\text{nine}}^{i}[(x, y - 1)], M_{\text{nine}}^{i}[(x + 1, y + 1)], M_{\text{nine}}^{i}[(x + 1, y)], M_{\text{nine}}^{i}[(x, y - 1)], M_{\text{nine}}^{i}[(x + 1, y)], M_{\text{nine}}^{i}[(x - 1, y - 1)], M_{\text{nine}}^{i}[(x - 1, y)]) \]

**Table 3.1:** Step component in the structured stencil model of the nine point stencil application that is described in pseudo code in Figure 3.5.

### 3.4.3 Concurrent Level: The naive Method

Passing from the partition dependency level to the concurrent one, the dependencies featured by partitions are resolved with inter processes communication, for example though message passing mechanisms.

There are two ways to implement this information exchange. We first consider the easiest one that we call the *naive* method. The method resolves incoming and outgoing partition dependencies with direct communications between processes. Therefore each process, that manages its own partition of the spatial structure, has to both send and receive data from and to all its neighbours processes.

We report in Figure 3.7 the pseudo code representing the behaviour of the generic process. For sake of simplicity, in communication operations we just indicate the sender or the receiver process, while we leave to both Figure 3.8(a) and Figure 3.8(b) the burden of graphically highlighting which are exactly the elements that are involved in the communications.
```c
double J_in[512][512], J_out[512][512];
load_partition_values(J_in);
for(i_step = 0; i_step < 5; i_step + +){
    SEND(P(-1,-1)); SEND(P(-1,0)); SEND(P(-1,1)); SEND(P(0,-1));
    SEND(P(0,1)); SEND(P(+1,-1)); SEND(P(+1,0)); SEND(P(+1,1));

    COMPUTE(Incoming independent region);

    RECV(P(-1,-1)); RECV(P(-1,0)); RECV(P(-1,1)); RECV(P(0,-1));
    RECV(P(0,1)); RECV(P(+1,-1)); RECV(P(+1,0)); RECV(P(+1,1));

    COMPUTE(Incoming dependent region);
    swap(J_in, J_out);
}
return_partition(J_out);
```

**Figure 3.7:** Representation in pseudo code of a nine point stencil application described at the concurrent level exploiting the *naive* method.

**Figure 3.8:** Graphical representation of the incoming (fig. 3.8(a)) and outgoing (fig. 3.8(b)) communication of a *naive* implementation of the nine point stencil at the concurrent level.
Each step in the \textit{naive} method is composed by the following phases:

I Each process sends data to the eight neighbours. We suppose the communication are asynchronous with at least one degree of asynchronicity.

II The process computes the new values for the elements of the incoming independent region.

III The required data to complete the update for the element of the incoming dependent region are acquired

IV The values of the remaining elements are computed

The schema of the program is simple and does not introduce any particular difficulties about managing both computations and communications, also when targeting architectures supporting their overlapping.

The computations associated to the elements of the incoming independent region are the only ones that can possibly be overlapped to communication latency. Hence, in order to target the best efficiency, the region has to feature a data volume such that its computation time, i.e. the time required to update all its elements, is sufficient to completely masquerade the communication latency.

Considering that in a two dimension regular domain the number of possible neighbours is equal to 8, the simplest \textit{naive} approach features 8 incoming and as many outgoing communications per step. Because the previous observation can be extended to spaces featuring more dimensions we can claim the following Theorem.

\textbf{Theorem 3.4.1 (Naive Overhead).} Let $\psi$ be a generic \textit{HUA} stencil defined over a $n$ dimension spatial structure. A naive implementation of $\psi$ at the concurrent level, can required $3^n - 1$ incoming and $3^n - 1$ outgoing communication at most.

\textit{Proof.} Because in a naive implementation the partition dependencies are resolved though direct communication we can assert that we have:

$$\Delta^{in} = \Delta^{out} = \Delta^{in}_{com} = \Delta^{out}_{com}$$

Moreover, if we consider the nine point stencil, which is a worst case for communications, we can claim

$$|\Delta^{in}| = |\Delta^{out}| = |\Delta^{in}_{com}| = |\Delta^{out}_{com}| = 3^n - 1$$

\hfill $\Box$

Coming back again to our nine point stencil example, we focus on an analytic analysis of communication impact exploiting the $\mathcal{LC}$ cost model. We consider a specific configuration, where the computation time associate to the update of a single element is equal to zero. This approach allows as to avoid the introduction of
mechanisms in order to model the possible overlapping between computations and communications.

We can therefore model the communication overhead per each step \( T_{\text{com}}^{\text{naive}} \) as follows:

\[
T_{\text{com}}^{\text{naive}} = 8 \times (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \times s_{\text{total}}^{\text{naive}}
\]

The \( s_{\text{total}}^{\text{naive}} \) parameter is the total size of data that is sent by a process during a generic step. Let us suppose a square partition with \( l \) element for each border; we have that

\[
s_{\text{total}}^{\text{naive}} = 4 \times (l + 1) \text{size\_of\_one\_el}
\]

Extending the analytic analysis of the communication impacts to a \( n \) dimension space we can assert what follows.

\[
T_{\text{com}}^{\text{naive}} = (3^n - 1) \times (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \times s_{\text{total}}^{n \text{dim\_naive}} \quad (3.1)
\]

The \( s_{\text{total}}^{n \text{dim\_naive}} \) parameter is the total size of data that is sent by the generic process in a generic step of the extension to \( n \) dimension space of the nine point stencil.

### 3.4.4 Concurrent Level: The shift Method

An optimization of the previous naive schema can be introduced, exploiting the Plimpton’s shift method [41]. The strength of the approach is to avoid direct communications with diagonal neighbours; all data shift only along the main axes of the partition space.

We formally define a diagonal neighbour a neighbour partition whose associated movement vector \( \alpha \) (see Definition 3.2.2) contains more than one non null component:

\[
\alpha = (\alpha_1, \ldots, \alpha_n); \quad \exists i, j \quad \alpha_i \neq 0, \text{ and } \alpha_j \neq 0
\]

Symmetrically, we consider non diagonal neighbours those that are placed along the main axes of the partition reference system. Therefore, the generic partition, which is associated to the movement \( \alpha \), does not belong to the diagonal neighbour set when:

\[
\alpha = (\alpha_1, \ldots, \alpha_n); \quad \exists i \quad \alpha_i \neq 0
\]

The avoidance of diagonal communications implies that an explicit routing of the information from and to diagonal neighbours has to be implemented in the program.

In Figure 3.9, we report the pseudo code that implements the behaviour of a generic process that executes the shift method. The program schema is quite different from the naive one. In terms of communications, in the naive case each stencil step is characterized by one single send phase followed by a receive one. Instead, the
double J_{in}[512][512], J_{out}[512][512];
load_partition_values(J_{in});
for(i_{step} = 0; i_{step} < 5; i_{step}++){
SEND_1(P(1,0)); SEND_1(P(-1,0));
COMPUTE_A(Incoming independent region);
RECV_2(P(1,0)); RECV_2(P(-1,0));
SEND_3(P(0,-1)); SEND_3(P(0,1));
COMPUTE_B(Incoming independent region);
RECV_4(P(0,-1)); RECV_4(P(0,1));
COMPUTE(Incoming dependent region);
swap(J_{in}, J_{out});
}
return_partition(J_{out});

Figure 3.9: Representation in pseudo code a nine point stencil application described at the concurrent level exploiting the *shift* method.

![Graphical representation of the incoming (fig. 3.10(a)) and outgoing (fig. 3.10(b)) communication of a *shift* implementation of the nine point stencil at the concurrent level.](image-url)
Chapter 3. The HUA Complete Architecture

The shift method features a sequence of interleaved send and receive operations. Figure 3.10(a) and Figure 3.10(b) report respectively the incoming and outgoing communication patterns. In the Figures the arrows representing communications are labeled with indexes that establish the temporal order according which they are executed. The same indexes are reported on the pseudo code send and receive operations.

Considering a extension on a $n$ dimension space of the nine point stencil we can assert the following Theorem.

**Theorem 3.4.2 (Shift Overhead).** Let $\psi$ be a generic HUA stencil defined over a $n$ dimension spatial structure. An implementation of $\psi$ at the concurrent level, requires $2 \times n$ incoming and $2 \times n$ outgoing communication at most.

**Proof.** In a $n$ dimension space, the multidimensional extension of the nine point stencil requires information exchange with all its neighbours. Exploiting the Plimpton’s shift methods all the direct communications with the diagonal neighbours can be avoided. A process therefore communicate only with non diagonal neighbours:

$$\Delta_{in}^\text{com} = \Delta_{out}^\text{com} = \{ \forall P(\beta_1, \ldots, \beta_n) \in \Delta_{in}; \exists i \beta_i \neq 0 \}$$

Because a component of a movement vector can be equal to -1, 0 or 1, we can assert that:

$$|\Delta_{in}^\text{com}| = |\Delta_{out}^\text{com}| = 2 \times n$$

We now focus the discussion on the analysis of the communication impacts as we made for the naive method. Consider a configuration where the computation time associate to the update of a single element is equal to zero. In this case we have that the communication overhead per each step can be modelled as follows:

$$T_{com}^{\text{shift}} = 4 \times (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \times s_{\text{shift}}^\text{total}$$ (3.2)

The $s_{\text{total}}$ parameter is the total size of data that is sent by the generic process in a generic step. Instead, considering an extension in a $n$ dimension space of the nine point stencil, we have:

$$T_{com}^{\text{shift}} = 2 \times n \times (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \times s_{\text{dim}\_shift}^\text{total}$$ (3.3)

The different impacts that the two methods have with respect to communications is evident comparing the previous Equation 3.3 with Equation 3.1 at page 92. The two quantity $s_{\text{dim\_naive}}^\text{total}$ and $s_{\text{dim\_shift}}^\text{total}$ are equal: it is evident from the previous Figures in the two dimension case and it can be also proved for a space featuring any number of dimensions. The two methods send and receive in each step the same amount of data that however is differently packed. Indeed, the naive method is characterized by a high number of communications which can be also of small dimensions, while the shift one features less but bigger communications.
3.4. A NINE POINT STENCIL AT WORK

As we can claim from the previous discussion on the message sizes, the different impact of the two methods is restricted to the setup communication overhead. This fact guarantees that the improvement of the \textit{shift} method are going to be valid also when targeting \textit{LC} implementations that support the overlapping of computations and communications.

Finally it is worth mentioning that the weigh of the performance improvement of the shift method strongly depends on the size of the data sent in a single step. Indeed, if in $T_{\text{com}}^{\text{shift}}$ the terms of the setup overhead, i.e. $(2 \ast n) \ast (t_{\text{snd}} + t_{\text{rcv}}^{\text{setup}})$, is negligible with respect to the term of the transfer overhead, i.e. $t_{\text{transm}} \ast s_{\text{total}}^{n \ast \text{dim}_{\text{-shift}}}$, the improvement of the shift method is going to be negligible as well. Therefore the shift method is a technique that is useful when targeting fine grain computations.

3.4.5 Environments of the Experimental Tests

In this Section we describe the environment that we exploit to test the communication overhead of a stencil program on real architectures.

The architectures we exploited are the listed below.

- **I Pianosa**: a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100
- **II Ottavinareale**: an eight core Intel Xeon (CPU E5420 @ 2.50GHz)
- **III Siberia**: nine core Cell B.E.

On top of Siberia we exploited our implementation of \textit{LC} that is presented in Appendix A. Because up to now we do not have any \textit{LC} implementation for the other two architectures, we decide to target the MPI message passing library. On the Pianosa cluster we selected MPICH 2 version of the library. Instead for the Ottavinareale multi-core we selected an implementation of MPICH that is optimized for shared memory architectures (shared memory MPICH).

For each test we keep the number of partition fixed with respect to the number of dimensions of the space. For example in a two dimension space we considered a partition space represented as follows:

$$[0, 3] \times [0, 3]$$

This is the minimum configuration such that a generic partition can feature exactly eight \textbf{distinct} neighbours. In the general case of a n dimension space the smallest partition space we consider is therefore

$$[0, 3] \times [0, 3] \times \ldots \times [0, 3]$$
Fixed the number of partition we run the tests increasing the number of elements of each partition. We used this configuration because it represents the test model that can simulate the behaviours of the different implementations with different types of grain while exploiting the minimum number of resources.

3.4.6 Experimental Results

In this Section we present the result of test implementing the naive and the shift methods. As reference stencil-based application, we considered the nine point stencil, in a two dimension space, and its extension in a three dimension space (in this case the stencil is called twenty seven point stencil). As for the analytic study, we focus only on communication overhead, therefore each process in the presented test performs only communications avoiding the computation phase. As environment test we exploited the one described in the previous Section.

Figure 3.11 and Figure 3.12 report the performance results on the Pianosa cluster. Figure 3.13 and Figure 4.14 report the ones for Ottavinareale, i.e. the intel multi-core architecture. Finally, Figure 3.15 reports performance results on Siberia, the Cell B.E IBM multi-core architecture.

In the charts the time gain parameter $G_{com}$ is the result of the division of the step completion time of a reference method by the step completion time of a target implementation $\chi$. For all the tests we consider the naive method as the reference one. Formally we have:

$$G_{com} = \frac{T_{naive}}{T_{\chi}}$$

From our measures we can claim that in most of the cases the shift method performs better with respect to the naive one, more over the time gain of the first one is always higher when considering fine grain, i.e. partition with less elements. Indeed, smaller partitions imply a lower size of all the transferred messages. In a such configuration, the constant setup communication overhead, which the shift method aims to reduce, feature a greater impact on the overall latency with respect to the component that is proportional to the traffic size.

We moreover observe that the time gain of the shift method is always higher in the three dimension space with respect to the two dimension one. This aspect was also easily deducible by the presented performance model. Indeed the number of diagonal neighbours grows more than the number of non diagonal ones with respect to the increasing of the space dimensions.

The tests that register the performance of the nine point stencil on the cluster environment results in conclusion that are not compatible with the analytic studies of the communication impacts. By the way this results agree with the ones presented by Palmer and Nieplocha [39]. The two scientists conclude from their experimental studies that the optimal method between the shift and the naive depends both by the architecture and the size and the dimension of the problem.
Figure 3.11: Communication overheads featured by naive and shift implementations of the nine point stencil in a two dimension space performed on dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.
Figure 3.12: Communication overheads featured by \textit{naive} and \textit{shift} implementations of the twenty-seven point stencil in a three dimension space performed on dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.
3.4. A NINE POINT STENCIL AT WORK

Figure 3.13: Communication overheads featured by *naive* and *shift* implementations of the nine point stencil in a two dimension space performed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
Figure 3.14: Communication overheads featured by `naive` and `shift` implementations of the twenty-seven point stencil in a three dimension space performed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
3.4. A NINE POINT STENCIL AT WORK

Figure 3.15: Communication overheads featured by *naive* and *shift* implementations of the twenty-seven point stencil in a three dimension space performed on top of an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
Finally, it is worth to point out the steps featured by the curves in the charts of the intel multi-core architecture. This phenomena are observable for both naive and shift methods and also in both the two and three dimension cases. Indeed, they are linked to the relation between the cache line size and the largest message that is sent. The phenomena appear when the size of this last entity exceeds the cache line size.

Two observations supports the previous assertion. Firstly, the biggest messages associated to the two methods feature more or less the same size. Secondly, the steps on the curves appear for both the method at the same partition dimension.

3.4.7 Conclusions

The Theorem 4.3.2 represents the best result in literature for the minimization of the number of communications in $\mathcal{HUA}$ stencils. We want just to highlight that the shift method does not introduce any benefit for example in the Jacobi or Laplace cases, where the stencil does not feature any information sharing between diagonal neighbours. Therefore we can conclude that at the state of the art a generic $\mathcal{HUA}$ stencil require $2 \ast n$ incoming and outgoing communication and secondly that the Jacobi and Laplace can not be implemented with less than $2\ast n$ communications. In the next Chapter we are going to present some optimizations that, modifying the stencil at the functional level, break both the previous limits.
Chapter 4

$Q$–transformations

Abstract

One of the most used techniques exploited to describe stencil computations is the *owner-computes* rule. Although the rule makes the definition of a stencil parallelization an easier task, we discover that alternative solutions can lead to important new optimizations that reduce the number of dependencies between domain partitions.

The new transformations, which are classified as relaxed safe, reduce to $n$ the maximum number of communications required to implement a $HUA$ stencil, where $n$ is the number of dimensions of the targeted space. The optimizations achieved by the new introduced techniques represent the best results with respect to the solutions that are present in literature.

The Chapter is structured as follows. Section 4.1 introduces the $Q$–transformations in general and proves some of their properties.

Section 4.2 defines a particular case of $Q$–transformations called Positive $Q$–transformations and provides some Theorems highlighting the peculiarity of the new transformations.

Section 4.3 analyses the impact of Positive $Q$–transformations in the implementation of a nine point stencil at the concurrent level. Moreover it provides the proof that the $q$–shift method based on Positive $Q$–transformations feature the best result with respect to the solutions cited in literature.

Section 4.4 introduces an analysis on the impact of Positive $Q$–transformations in the case of stencils which do not feature partition dependencies with diagonal neighbours, like for the Jacobi one.

Section 4.5 presents Negative $Q$–transformations which feature antithetic characteristics with respect to the Positive ones. The Section prove the benefit of interleaving Positive and Negative $Q$–transformations in order to reduce the overhead to rearrange, if necessary, the value in their original position in the data structures.

Finally, Section 4.6 focuses on the extension of Positive and Negative $Q$–transformations to semi uniform stencils.
Contents

4.1  Q−transformations 105
  4.1.1 Defining Q−transformations 105
  4.1.2 Q−transformations are different from Skewing 107
4.2  Positive Q−transformations 110
  4.2.1 Defining Q+−transformations 110
4.3  A Nine Point Stencil at Work with Q+−transformations 113
  4.3.1 Functional Dependency Level 113
  4.3.2 Partition Dependency Level 113
  4.3.3 Concurrent Level: The q Method 115
  4.3.4 Concurrent Level: The q_shift method 119
  4.3.5 Experimental Results 121
  4.3.6 Conclusions 128
4.4  A Closer Analyses on Q+−transformations 129
  4.4.1 Analytic Analysis of the Nine Point Stencil 129
  4.4.2 The Jacobi Case 129
4.5  Negative Q−transformations 134
  4.5.1 Defining Q−−transformations 134
  4.5.2 Combining Positive and Negative Q−transformations 137
4.6  Extending Q−transformations to The Semi Uniform 138
4.7  Conclusions 140
4.1 \( Q \)-transformations

Since the Chapter 2 we have discussed about the concept behind the owner-computes rule and with the Definition 2.1.9 at page 27 we gave to the rule a formal revisitation in the terms of the structured stencil model. This rule is useful to define functional dependencies of stencil applications, nevertheless we observed that it can preclude same important optimizations. Hence, in this Chapter we introduce \( Q \)-transformations.

The \( Q \)-transformations compute some modifications of a \( \mathcal{HUA} \) stencil at the functional dependency level. The impact of the changes is observable at all the lower levels of the system architecture (see Chapter 3).

Exploiting some particular \( Q \)-transformations we are going to define two optimization methods that lower the maximum number of communications required to describe the stencil at the concurrent level. We already gave an preview example of this kind of \( Q \)-transformations in Section 2.5.2 of Chapter 3 at page 65.

4.1.1 Defining \( Q \)-transformations

Before formally presenting \( Q \)-transformations, we would like to recall and reinforce some important working hypotheses. In our studies we focus on a space invariant stencils which do not feature any loop-carried dependency inside one single step. In such configurations, as we will formally prove later on in this Chapter, a transformation which modifies the location of the application point of a \( \mathcal{HUA} \) stencil (such operation that is equivalent to brake the owner-computes rule) results in a relaxed equivalent program.

We start now from a formal definition of a generic \( Q \)-transformation.

**Definition 4.1.1 (Generic \( Q \)-transformation).** Let \( \psi \) be a \( \mathcal{HUA} \) stencil. A generic \( Q \)-transformation transforms \( \psi \) in the \( \mathcal{HUA} \) stencil \( Q[\psi] \) which is equivalent to the original one, except for the step model which is defined as follows:

\[
\forall e \in \mathcal{M} \quad Q[\psi] \quad (F^\psi, S^Q[\psi])
\]

\[
q = (q_1, \ldots, q_\text{dim})
\]

\[
R^Q[\psi] = R^\psi + q
\]

\[
S^Q[\psi] = e + R^Q[e]
\]

\[
\mathcal{M}^{i+1}_{Q[\psi]}[e] = F_i(\mathcal{M}^i_{Q[\psi]}[e + \gamma_1], \ldots, \mathcal{M}^i_{Q[\psi]}[e + \gamma_n])
\]

A generic \( Q \)-transformation introduces some modifications on the relative shape of a \( \mathcal{HUA} \) stencil. The changes can be geometrically represented in two equivalent ways with respect to the selected observation point. Indeed, the changes produce a rigid translation of all the relative shape elements or equivalently they result in a translation of the application point.
The Definition 4.1.1 asserts that the resulting stencil is still be a \( \text{HUA} \) stencil. To assure the correctness of the previous definition we report the following Property.

**Property 4.1.1** \( (Q^{-}\text{transformations are \text{HUA}}) \). \( Q^{-}\text{transformations} \) map \( \text{HUA} \) stencil in \( \text{HUA} \) stencils.

**Proof.** Trivially, we have that the only difference between the original and the transformed stencils is the step model. Because it is evident that the modified stencil step is compatible with the \( \text{HUA} \) model, we can conclude that the transformed stencil is still a \( \text{HUA} \) stencil.

Another interesting characteristic of a generic \( Q^{-}\text{transformation} \) is given by the following Theorem.

**Theorem 4.1.1** \( (Q^{-}\text{transformation relaxed safety}) \). A program resulting from a general \( Q^{-}\text{transformation} \) is relaxed equivalent with respect to the original one.

**Proof.** Considering a generic domain space \( \mathcal{M} \), let \( \mathcal{M}_\psi^i \) be the state of the domain when applying \( i \) times a generic stencil \( \psi \). Let \( \mathcal{M}_Q^i \) be the value of the domain when applying the stencil \( Q(\psi) \), which is the result of a generic \( Q^{-}\text{transformation} \).

In order to prove the Theorem, we have to demonstrate that in a generic step \( i \) the values of \( \mathcal{M}_Q^i \) are the same of \( \mathcal{M}_\psi^i \), apart from a space translation.

Let the \( \mathcal{M}^0 \) be the status of the domain before applying any stencil. We can write

\[
\mathcal{M}^0 = \mathcal{M}_S^0 = \mathcal{M}_Q^0 \tag{4.1}
\]

If we define the element \( a = e + q \), by Definition 4.1.1 at page 105 we can describe the domain state after the first step as:

\[
\mathcal{M}_Q^1[e] = F_0(\mathcal{M}_Q^0[e + \gamma_1], \ldots, \mathcal{M}_Q^0[e + \gamma_n]) = F_0(\mathcal{M}_Q^0[a + \beta_1], \ldots, \mathcal{M}_Q^0[a + \beta_n])
\]

Therefore for Equation (4.1) we have:

\[
\mathcal{M}_Q^1[e] = F(\mathcal{M}_S^0[a + \beta_1], \ldots, \mathcal{M}_S^0[a + \beta_n])
\]

At this point, according to the Definition 2.5.1 at page 64 of the \( \text{HUA} \) model, it comes that:

\[
\mathcal{M}_Q^1[a] = \mathcal{M}_Q^1[e + q] = \mathcal{M}_Q^1[e]
\]

Iterating the previous reasoning for various steps, we find that:

\[
\mathcal{M}_Q^i[a] = \mathcal{M}_Q^i[e + i \ast q] = \mathcal{M}_Q^i[e] \tag{4.2}
\]

\[
\mathcal{M}_Q^i[e] = \mathcal{M}_Q^i[e - i \ast q] \tag{4.3}
\]
Thanks to the previous Theorem we know that we can modify a \textit{HUA} stencil exploiting any kind of \textit{Q-transformations} and that the resulting stencil can be used, in place of the original one, in order to obtain the same output data values.

Up to now, the general definition of \textit{Q-transformations} does not provide useful information to define new optimization techniques. In the next Sections we focus on two specifications of the \textit{Q-transformations} that are called positive and negative. This transformations are at the core of the optimizations we are studying.

### 4.1.2 \textit{Q}-transformations are different from Skewing

Before proceeding with the study of \textit{Q-transformations}, we consider worthwhile a small digression about the \textit{Q-transformations} and loop skewing.

Skewing [23, 7] is a well known transformation in the area of data dependency optimization theory which was invented to handle wavefront computations like Gauss-Seidel one.

\textit{Q-transformations} and Skewing transformations have in common the fact that both change the form of some geometric representation of a stencil. By the way, it is important to understand that the two techniques work on completely different aspects of a stencil, with different hypothesis, and also different aims.

A description in pseudo code of one iteration of the Gauss-Seidel in a two dimension space is reported in Figure 4.1, while the skewed version is reported in Figure 4.2.

```plaintext
double J[101][101];
for(x = 1; x < 100; x + +){
    for(y = 1; y < 100; y + +){
        J[x][y] = (J[x][y + 1] + J[x][y - 1] + J[x + 1][y] + J[x - 1][y]) / 4;
    }
}
```

Figure 4.1: Pseudo code of one iteration of the Gauss-Seidel stencil on a two dimensional space. The stencil features \{(1, 1), (0, 1)\} as dependency vector and \(\mathcal{R} = \{(0, 1)(0, -1)(1, 0)(-1, 0)\}\) as shape (considering an extension of the shape element to non \textit{HUA} stencil)

About the features of skewing transformations, we can assert the following points.

I Skewing transformations were designed for wave front stencil applications, where each single step features loop-dependent dependencies.
```java
double J[101][101];
for(x = 1; x < 100; x + +){
    for(y = x + 1; y < 100; x + y + +){
        J[x][y-x] = (J[x][y-x+1] + J[x][y-x-1] + J[x+1][y-x] + J[x-1][y-x])/4;
    }
}
```

**Figure 4.2:** Pseudo code of one iteration of the skewed Gauss-Seidel stencil on a two dimensional space. The skewed stencil features \{ (1, 1), (0, 1) \} as dependency vector and \( R = \{ (0, 1)(0, -1)(1, 0)(-1, 0) \} \) as shape (considering an extension of the shape element to non HUA stencil)

II Skewing transformations work on the statement dependency space exploiting dependency vectors that are the main mechanisms of the classical optimization theory.

III Skewing is a safe transformation and not only a relaxed one. Indeed, the outputs of the original and the skewed programs are equivalent without any kind of exception.

IV The aim of skewing is to change the visit path that is exploited to update the data structure elements. The different path is selected preserving data dependencies.

V The relative shape of the skewed program, which represents the functional dependency of the stencil, is the equal to the one of the original program.

While about \( \mathcal{Q} \)-transformations we can say the following

I \( \mathcal{Q} \)-transformations work on functional dependencies between working domain elements, represented by the shape construct, in steps that do not feature loop-dependent dependencies. Indeed, we recall that Gaus-Seidel does not belong to HUA stencils because, differently from the similar Jacobi stencil, it features loop-dependent dependencies in the single step computation.

II The \( \mathcal{Q} \)-transformations are relaxed safe transformations as Theorem 4.1.1 proves. Indeed, the outputs of the original and the transformed stencil are equivalent, except for a rigid translation of all the elements in the spatial structure.

III \( \mathcal{Q} \)-transformations map HUA stencils into HUA stencils as Property 4.1.1 proves. Therefore, because a transformed stencil still belongs to the HUA
class any strategy for the data structures visit is acceptable. Indeed \textit{HUA} stencils do not feature any loop-carried dependency.

IV The relative shape of the transformed stencil is completely different from the original one. Indeed, \textit{Q-transformation}s aim the change of the functional dependencies between elements of the spatial structure with respect to the associated application point.

From the previous point is now completely clear that the two techniques are completely different.
4.2 Positive Q−transformations

We have seen in the previous Sections that Q−transformations can transform a program into a relaxed equivalent one. We are now going to focus on the definition of a set of particular Q−transformations which can automatically produce an equivalent relaxed program that features better performance with respect to the original one.

4.2.1 Defining Q+−transformations

Definition 4.2.1 (Positive Q−transformation). Let ψ be a HUA stencil. A positive Q−transformation transforms ψ in the HUA stencil $Q^+\psi$ which is equivalent to the original one, except for the step model that is defined as follows:

$$\forall e \in M \quad Q^+\psi \rightarrow (F^\psi, S^Q^+\psi)$$

$$q^+ = \langle q_1^+, \ldots, q_{dim}^+ \rangle$$

$$q^+_i = -\min \{\beta_\alpha * \epsilon_i | \epsilon_i \in \mathbb{R}\}$$

$$R^{Q^+\psi} = R^\psi + q^+$$

$$S^{Q^+\psi} = e + R^{Q^+\psi}$$

$$M_{i+1}^{Q^+\psi}[e] = F_i(M_i^{Q^+\psi}[e + \gamma_1], \ldots, M_i^{Q^+\psi}[e + \gamma_n])$$

(4.4)

$\epsilon = \{\epsilon_1, \ldots, \epsilon_{dim}\}$ is the set of the vectors in the natural basis of $\mathbb{N}^{dim}$, and $\beta_\alpha * \epsilon_i$ is the scalar product which returns the component of the vector $\beta_\alpha$ along the main space direction expressed by the vector $\epsilon_i$.

The vector $q^+$ is called the positive vector.

With respect to the original stencil, only the set $R$ has been changed into $Q^+(R)$: adding to each vector element of $R$ the constant positive vector $q^+$. The resulting new set, i.e. $Q^+(R)$, features a nice property that is the fulcrum of some optimizations that we can derive from Q−transformations.

Property 4.2.1 (Shape Orientation). The set $Q^+(R)$ can be represented as set of vectors featuring only positive components:

$$Q^+(R) = \{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_i * \epsilon_i \geq 0 \forall i \in \{1, \ldots, n\}\}$$

$\epsilon = \{\epsilon_1, \ldots, \epsilon_{dim}\}$ is the set of the vectors in the natural basis of $\mathbb{N}^{dim}$, and the operator * represents the scalar products between vectors.

Proof. By definition of positive Q−transformations we have the following equation.

$$R^{Q^+\psi} = \{\gamma_1, \gamma_2, \ldots, \gamma_n | \gamma_i = \beta_\alpha + q^+ \forall \alpha\}$$
4.2. POSITIVE $Q$–TRANSFORMATIONS

\[ (1, 0) \quad (0, 1) \quad (0, -1) \quad (-1, 0) \]

\[ (1, 0) \quad (2, 1) \quad (1, 2) \quad (0, 1) \quad (1, 2) \]

Figure 4.3: Graphical representations of the relative shapes of respectively $Jacobi$ and $Q^+[Jacobi]$ stencils

\[
\text{forall} ((x, y) \in J_{in}) \{
\begin{align*}
J_{out}[x][y] &= (J_{in}[x][y + 1] + J_{in}[x + 1][y] + J_{in}[x][y - 1] + J_{in}[x + 1][y - 1]) / 4; \\
\end{align*}
\}
\]

Figure 4.4: Jacobi pseudo code.

Therefore the condition $\gamma_\alpha \ast \epsilon_i \geq 0$ can be rewritten as:

\[
(\beta_\alpha + q^+) \ast \epsilon_i \geq 0 \\
(\beta_\alpha \ast \epsilon_i) + q_i^+ \geq 0 \\
(\beta_\alpha \ast \epsilon_i) - \min \{ (\beta_\alpha \ast \epsilon_i) \forall \beta_\alpha \in \mathcal{R} \} \geq 0 \\
(\beta_\alpha \ast \epsilon_i) \geq \min \{ (\beta_\alpha \ast \epsilon_i) \forall \beta_\alpha \in \mathcal{R} \}
\]

Because the last inequation is obviously valid, we have that the formula $\gamma_\alpha \ast \epsilon_i \geq 0$ is valid as well.

\[
\text{forall} ((x, y) \in J_{in}) \{
\begin{align*}
J_{out}[x][y] &= (J_{in}[x][y + 1] + J_{in}[x + 2][y + 1] + J_{in}[x + 1][y] + J_{in}[x + 1][y + 2]) / 4; \\
\end{align*}
\}
\]

Figure 4.5: $Q^+[Jacobi]$ pseudo code.
We consider once again the case of Jacobi stencil. We recall that the relative shape of the stencil is 
\[ R^{Jacobi} = \{(-1, 0), (1, 0), (0, -1), (0, 1)\} \]. Positive \( Q^- \) transformations associate to the Jacobi stencil the positive vector \( q^+ = (+1, +1) \) and define the relative shape of the stencil \( Q^+[Jacobi] \) as follows:

\[ R^{Q^+[Jacobi]} = \{(0, 1), (2, 1), (1, 0), (1, 2)\} \]

All the components of the new relative shape are not negative, as we expected from the previous Property. As we will show in the next Section, one of the effect of this geometric feature is going to be a kind of "orientation" of the communications.

A graphical comparison of both the original and the transformed relative shapes is reported in Figure 4.3. The shapes are geometrically the same, except for the location of the application point which is modelled by the reference system origin.

We also report the pseudo code of both the original Jacobi and \( Q^+(Jacobi) \) respectively in Figure 4.22 and Figure 4.5.
4.3 A Nine Point Stencil at Work with $Q^+\text{--transformations}$

In this Section we resume the nine point stencil, that we used in the previous Chapter as worst case for communication analyses, in order to to study the benefit of exploiting positive $Q\text{-transformations}$.

4.3.1 Functional Dependency Level

As we have seen, $Q\text{-transformations}$ introduce changes in the original stencil that are expressed directly at the functional dependency level. The result of $Q\text{-transformations}$ is a $\mathcal{HUA}$ stencil. We report the step model of the stencil $Q^+[\text{nine}]$ in Table 4.1 at page 114. The stencil description can be compared with the one of the original stencil that is reported in the previous Chapter in Table 3.1 at page 89. Finally a graphical representation of the new stencil shape is reported in Figure 4.6(a).

4.3.2 Partition Dependency Level

The benefit of exploiting $Q\text{-transformations}$ become clear when expressing the $\mathcal{HUA}$ stencil description at the partition dependency level. Figure 4.6(b) and Figure 4.6(c) highlight all the regions in which the generic partition is composed.

Because the vectors in the relative shape do not present negative components, the elements of both the incoming and the outgoing dependent regions are concentrated only over two edges of the partition borders.

As a consequence of the previous observation, we have that the incoming and outgoing partition sets are strict sub sets of the neighbour partition one:

$$\Delta^\text{in} \subset NS(P_\alpha)$$

$$\Delta^\text{out} \subset NS(P_\alpha)$$

Extending the previous assertion to a generic stencil we can define the following Property.

Property 4.3.1 (Partition Dependency Reduction with $Q\text{-transformations}$). Let $\psi$ be a $\mathcal{HUA}$ stencil and $Q^+[\psi]$ the stencil resulting from the application of a positive $Q\text{-transformation}$ to $\psi$. About the transformed stencil, we can assert the following relations between sets.

$$\Delta^\text{in}_{Q^+[\psi]} \subset NS(P_\alpha)$$
∀e = (x, y) ∈ \mathcal{M}_{\mathbb{Q}^{+}[nine]} \quad \xrightarrow{\mathbb{Q}^{+}[nine]} \quad \left( \mathcal{F}, S_{e}^{\mathbb{Q}^{+}[nine]} \right)

\mathcal{R}_{\mathbb{Q}^{+}[nine]} = \{(x, y + 1), (x, y - 1), (x + 1, y - 1),
(x + 1, y), (x + 1, y + 1), (x - 1, y - 1),
(x - 1, y), (x - 1, y + 1)\} + (+1, +1)

= \{(x + 1, y + 2), (x + 1, y), (x + 2, y),
(x + 2, y + 1), (x + 2, y + 2), (x, y),
(x, y + 1), (x, y + 2)\}

\mathcal{S}_{(x,y)}^{\mathbb{Q}^{+}[nine]} = (x, y) + \mathcal{R}_{\mathbb{Q}^{+}[nine]}

\mathcal{F} : \mathbb{R}^4 \mapsto \mathbb{R}

\mathcal{F} : (r_1, r_2, r_3, r_4) \mapsto \frac{\sum_{i=1}^{8} r_i}{8}

\mathcal{M}_{\mathbb{Q}^{+}[e]}^{(i+1)} = \mathcal{F}(\mathcal{M}_{\mathbb{Q}^{+}[(x + 1, y + 2)]}, \mathcal{M}_{\mathbb{Q}^{+}[(x + 1, y)]},
\mathcal{M}_{\mathbb{Q}^{+}[(x + 2, y + 2)]}, \mathcal{M}_{\mathbb{Q}^{+}[(x + 2, y + 1)]},
\mathcal{M}_{\mathbb{Q}^{+}[(x + 1, y)]}, \mathcal{M}_{\mathbb{Q}^{+}[(x, y + 2)]},
\mathcal{M}_{\mathbb{Q}^{+}[(x, y + 1)]}, \mathcal{M}_{\mathbb{Q}^{+}[(x, y)]},
\mathcal{M}_{\mathbb{Q}^{+}[(x + 2, y + 1)]}, \mathcal{M}_{\mathbb{Q}^{+}[(x, y + 1)]})

Table 4.1: Step component in the structured stencil model of the \(Q^{+}[nine]\) stencil application.
4.3. A NINE POINT STENCIL AT WORK WITH \( Q^+ − \text{TRANSFORMATIONS} \)

\[ \Delta^{\text{out}}_{Q^+[e]} \subset NS(\mathcal{P}_a) \]

\( \mathcal{P}_a \) is a generic partition in which the working domain has been divided and \( NS \) is the set of neighbour partitions of \( \mathcal{P}_a \).

Proof. Exploiting the Property 4.2.1 about the shape orientation introduced by \( Q − \text{transformations} \), we can define the incoming and the outgoing partition dependency set as follows:

\[ \Delta^{\text{in}} = \{ \forall \mathcal{P}_{(x,y)} \in NS(\mathcal{P}_a) \mid x \geq 0, y \geq 0 \} \]

\[ \Delta^{\text{out}} = \{ \forall \mathcal{P}_{(x,y)} \in NS(\mathcal{P}_a) \mid x \leq 0, y \leq 0 \} \]

Because all the neighbour partitions that feature an index with negative components do not belong to \( \Delta^{\text{in}} \), than the set is a strict sub set of \( NS(\mathcal{P}_a) \).

Symmetrically, we can assert the same for the outgoing partition dependency set. \( \square \)

4.3.3 Concurrent Level: The \( q \) Method

What we previously called the effect of the "orientation" of communications are analyzable at the concurrent level.

Considering that \( Q^+[\text{nine}] \) is a \( \mathcal{HUA} \) stencil, we recall that in the previous Chapter we show two different methods to implement the information exchange, that is necessary to resolve the partition dependencies.

We first consider the naive method that resolves incoming and outgoing partition dependencies with direct communications between processes. Each process, that manages its own partition of the spatial structure, has to send data to the processes belonging to the outgoing partition set (\( \Delta^{\text{out}} \)) and receive data from those belonging to the incoming partition set (\( \Delta^{\text{in}} \)).

We introduce a new nomenclature; in order to highlight that the naive method is applied to a stencil that is the result of a \( Q − \text{transformations} \), we rename the naive method as \( q \) method.

We report in Figure 4.7 the pseudo code representing the behaviour of the generic process implementing the \( q \) method for the nine point stencil application. As we did in the previous Chapter, in the code we exploited the name of the partitions to indicate the associated processes. Moreover, for sake of simplicity, in communication operations we just indicate the sender or the receiver process, while we leave to both Figure 4.8(a) and Figure 4.8(b) the burden of graphically highlighting which are exactly the elements that are involved in the communications.

Each step in the \( q \) method is composed by the following phases:
Figure 4.6: Graphical representation of stencil shape (4.6(a)), incoming dependent (in gray color) and independent regions (4.6(b)) and finally outgoing dependent (in gray color) and independent regions (4.6(c) ) for the $Q^+[nine]$ stencil.
4.3. A NINE POINT STENCIL AT WORK WITH $Q^+−TRANSFORMATIONS$

\begin{verbatim}
double $J_{in}[512][512], J_{out}[512][512];$
load_partition_values($J_{in}$);
for($i_{step} = 0; i_{step} < 5; i_{step} += 1$){
    SEND($P_{(0,1)}$); SEND($P_{(+1,0)}$); SEND($P_{(+1,+1)}$);
    COMPUTE($Incoming\ independent\ region$);
    RECV($P_{(0,1)}$); RECV($P_{(+1,0)}$); RECV($P_{(+1,+1)}$);
    COMPUTE($Incoming\ dependent\ region$);
    swap($J_{in}, J_{out}$);
}
return_partition($J_{out}$);
\end{verbatim}

\textbf{Figure 4.7:} Representation in pseudo code a $Q^+[nine]$ stencil described at the concurrent level exploiting the $q$ method.

\textbf{Figure 4.8:} Graphical representation of the incoming (fig. 4.8(a)) and outgoing (fig. 4.8(b)) communications of a $q$ implementation of the nine point stencil at the concurrent level.
I Each process sends data to the three neighbours that are associated with a movement vector with non negative components. We suppose the communications feature at least one degree of asynchronicity; otherwise the presented program would be going to stall in a deadlock situation.

II A process computes the new values for the elements of the incoming independent region.

III The data, required to complete the update for the elements of the incoming dependent region, are acquired from the three neighbour partitions that are associated with a movement vector featuring non positive components.

IV The values of the remaining elements are computed

As it was for the naive method, the schema of the program is simple and does not introduce any particular difficulties about managing both computations and communications, also when targeting architectures supporting their overlapping.

For the nine point stencil, which represents our reference worst case, we can assert that the $q$ method requires only three incoming and three outgoing communications. Therefore, the new $q$ method lower the number of communications with respect to both naive and shift method. Up to now, the $q$ method provides in a two dimension space the best optimizations technique.

The following Theorem extends the analysis to space with a generic number of dimensions.

**Theorem 4.3.1 (The q Method Impact on Communications).** Let $\psi$ be a generic $HUA$ stencil defined over a $n$ dimension spatial structure. A $q$ implementation of $\psi$ at the concurrent level, can required $2^n - 1$ incoming and $2^n - 1$ outgoing communications at most.

**Proof.** Because partition dependencies are resolved with direct communications, the number of incoming communications is equal to the cardinality of the incoming partition set:

$$\Delta_{com}^{in} = \Delta^{in} = \left\{ \forall P_{(\beta_1, \ldots, \beta_n)} \in NS(P) \| \forall \beta_i, \beta_i \geq 0 \right\}$$

Because in the definition of neighbour index set (see Definition 3.2.2 at page 78) we have that $\beta_i \in \{-1, 0, 1\}$ and that $(\beta_1, \ldots, \beta_n) \neq (0, \ldots, 0)$, we rewrite the previous formula of the incoming partition set as:

$$\Delta_{com}^{in} = \Delta^{in} = \left\{ \forall P_{(\beta_1, \ldots, \beta_n)} \in NS(P) \| \forall \beta_i \in \{0, 1\} \text{ and } (\beta_1, \ldots, \beta_n) \neq (0, \ldots, 0) \right\}$$

The cardinality of the set is equal, except for one unit less, to one of the set composed by all the possible strings of length $n$ that can be created by an alphabet of two digits: $2^n$. We can finally assert the following:
4.3. A NINE POINT STENCIL AT WORK WITH $Q^+$–TRANSFORMATIONS

\[ |\Delta_{in|com}^{|in|} = |\Delta_{in}^{|in|} | = 2^n - 1 \]

The same reasoning can be applied to the outgoing partition set which cardinality is equal to the number of outgoing communications.

\[ |\Delta_{out|com}^{|out|} = |\Delta_{out}^{|out|} | = 2^n - 1 \]

We report in Figure 4.11 a chart that represents, with respect to the number of space dimensions, the number of incoming communications for each of the studied method: naive, shift and q. What we can see is that the new q method provides a lower number of communications than the shift one only when the number of space dimension is less than two. Therefore the gain of the q method is limited to spaces with a low number of dimensions. We improve the reached results with another new method that is introduced in the following Section.

4.3.4 Concurrent Level: The $q$–shift method

As the naive method, the previous q one features communications with diagonal neighbours. Those can be avoided exploiting the Plimpton’s shift method. We call the mix of q and shift methods $q$–shift. The new method resolves the partition dependencies that the $Q^+$[nine] stencil implies with diagonal neighbours exploiting indirect communications.

In Figure 4.9, we report the pseudo code implementing the behaviour of a generic process that executes the $q$–shift method. The program features one couple of interleaved send and receive operations. Figure 4.10(a) and Figure 4.10(b) report respectively the incoming and outgoing communication patterns. In the Figures the arrows representing communications are labeled with indexes that establish the temporal order according which they are executed. The same indexes are exploited in the pseudo code to tag the corresponding send and receive operations.

We can assert that the implementation of the nine point stencil that exploits the $q$–shift method requires only two incoming and two outgoing communications. In the two dimension space the new method provides the best result with respect to all the other methods.

The following Theorem extends the result to spaces with a general number of dimensions.

Theorem 4.3.2 (The $q$–shift Method Impact on Communications). Let $\psi$ be a generic $HUA$ stencil defined over a n dimension spatial structure. Exploiting the $q$–shift method an implementation of $\psi$ at the concurrent level can required n incoming and n outgoing communication at most.
double $J_{in}[512][512], J_{out}[512][512]$;
load_partition_values($J_{in}$);
for($i_{step} = 0; i_{step} < 5; i_{step} += 1$) {
    SEND_1($P_{(1,0)}$);
    COMPUTE_A($\text{Incoming independent region}$);
    RECV_2($P_{(1,0)}$);
    SEND_3($P_{(0,1)}$);
    COMPUTE_B($\text{Incoming independent region}$);
    RECV_4($P_{(0,1)}$);
    COMPUTE($\text{Incoming dependent region}$);
    swap($J_{in}, J_{out}$);
} 
return_partition($J_{out}$);

**Figure 4.9:** Representation in pseudo code a $Q^+[nine]$ stencil described at the concurrent level exploiting the $q-shift$ method.

**Figure 4.10:** Graphical representation of the incoming (fig. 4.10(a)) and outgoing (fig. 4.10(b)) communications of a $q-shift$ implementation of the nine point stencil at the concurrent level.
4.3. A NINE POINT STENCIL AT WORK WITH $Q^+ - TRANSFORMATIONS$

Proof. In order to prove the Theorem, we consider $\psi$ the multidimensional extension of the nine stencil. Indeed, we proved that nine is a worst case stencil for the number of communications.

The partition dependency featured by the stencil are represented by the incoming and outgoing partition sets:

$$\Delta^\text{in}(P_\alpha) = \left\{ \forall P_{(\beta_1,\ldots,\beta_n)} \in NS(P_\alpha) | \forall i, \beta_i \geq 0 \right\}$$

$$\Delta^\text{out}(P_\alpha) = \left\{ \forall P_{(\beta_1,\ldots,\beta_n)} \in NS(P_\alpha) | \forall i, \beta_i \leq 0 \right\}$$

At the concurrent level, exploiting the Plimpton’s shift methods, partitions dependencies with diagonal neighbours are resolved without direct communications. Therefore, $\Delta^\text{com\_in}(P_\alpha)$, the set of the processes that feature send operation where $P_\alpha$ is the receiver, can be modeled as follows.

$$\Delta^\text{com\_in}(P_\alpha) = \Delta^\text{in}(P_\alpha) \cap \left\{ \forall P_{(\beta_1,\ldots,\beta_n)} \in NS(P_\alpha) | \exists! i \beta_i \neq 0 \right\}$$

We can rewrite the previously formula as:

$$\Delta^\text{com\_in}(P_\alpha) = \left\{ \forall P_{(\beta_1,\ldots,\beta_n)} \in NS(P_\alpha) | \exists! i \beta_i \neq 0 \text{ and } \beta_i \geq 0 \right\}$$

$$\left\{ \forall P_{(\beta_1,\ldots,\beta_n)} | \forall j \beta_j \in \{0, 1\} \text{ and } \exists! i \beta_i \neq 0 \right\}$$

About the number of incoming communication, we easily conclude with the following formula:

$$|\Delta^\text{com\_in}| = n$$

The same reasoning can be applied to the outgoing communications, in order to assert the following result:

$$|\Delta^\text{com\_out}| = n$$

4.3.5 Experimental Results

We considered a nine point stencil and its extension in a three dimensional space (twenty-seven points stencil) as tests for comparing all the four presented methods: naive, shift, q, and q_shift.

For all the tests we considered the environment presented in Section 3.4.5. As for the previous tests of the naive and shift method, in order to focus only on the
communication overhead we do not consider any computation associated to each element of the spatial structure.

Charts in Figure 4.12(a) and Figure 4.13(a) report results of the tests that were run on the intel multi-core architecture. It is interesting to observe that on this chip-multiprocessor architecture, featuring intra-chip communications, the results strongly respect the forecasts that can be extracted from the chart in Figure 4.11. The chart reports, for each method and for a set of space dimensions, the number of communications.

Indeed, the measured performances are characterized by the following invariant: the method which features a lower number of communications also supports a lower communication overhead. The observation is stressed by the comparison of the \textit{shift} and \textit{q} methods behaviour in the two dimension case and the three dimension one.

In two dimensions, \textit{q} method features a number of communications lower than the \textit{shift} ones and this is also reflected by performance test where \textit{q} presents a lower overhead. When we goes to analyse the three dimension case, the difference of the number of communications is inverted and as well the performance trend: the \textit{shift} method performs a step iteration faster than \textit{q}.

A comment has to be done on the steps featured by curves of all the methods in both performance charts. The causes are caching effects on message copying phase as it was mentioned in the performance of \textit{naive} and \textit{shift} presented in the previous Chapter.
4.3. A NINE POINT STENCIL AT WORK WITH $Q^+–TRANSFORMATIONS$

![Graph a](image1.png)

![Graph b](image2.png)

Figure 4.12: Performance result of a nine point stencil in a two dimension space, executed on top of an eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
Figure 4.13: Performance result of twenty-seven point stencil in a three dimension space, executed on top of a eight core Intel(R) Xeon(R) CPU E5420 @ 2.50GHz exploiting shared memory MPICH.
4.3. A NINE POINT STENCIL AT WORK WITH Q⁺–TRANSFORMATIONS

Figure 4.14: Nine point stencil in a two dimension space, performed on Cell B.E. IBM multicore exploiting the MammuT implementation of LL.
Figure 4.15: Performance result of a nine point stencil in a two dimension space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.
4.3. A NINE POINT STENCIL AT WORK WITH $Q^+-TRANSFORMATIONS$

Figure 4.16: Performance result of a twenty-seven point stencil in a three dimension space, executed on a dedicated thirty node cluster with Intel(R) Pentium(R) III CPU 800MHz and Ethernet Pro 100 exploiting MPICH library.
Charts in Figure 4.12(b) and Figure 4.13(b) represent the time gain of shift, 
q and q_shift with respect to the naive method. Regardless the number of space 
dimensions, the q_shift features the best time gain for fine grain parallelization; 
it is more than four times faster than the naive in the two dimension case and 
nine times in the three dimensional one. In the same condition, the shift method 
features a time gain of respectively of only two and five.

Charts in Figure 4.19(a) and Figure 4.19(b) report the tests run on the Cell B.E. 
multi-core. The results, which are presented only for the two dimension case, confirm 
the comments made for the previous architecture. The time gain reported in Figure 
4.19(b) are lower than the one registered for the intel architecture. This phenomenon 
come from the fact that we are targeting two completely different architecture and 
also that the LC support for the Cell is highly optimized and the setup overhead of 
the communication is reduced to the minimum (see Appendix A).

Charts in Figure 4.15 and Figure 4.16 report communication overheads regis-
tered on our cluster Pianosa. In the two dimensional case, all the methods present 
a comparable overhead, but optimizations based on elimination of diagonal commu-
nications, shift and q_shift methods, perform worst than the other two.

In the three dimension case the differences between the methods are more rel-
evant. Performance chart in Figure 4.16(b) shows that q_shift method reaches a 
time gain up to seventy with respect to naive while the shift one does not exceed 
the value of sixty.

4.3.6 Conclusions

Analyzing all the previous performance results on both cluster and multi-core ar-
chitectures, we can assert that optimizations based on Q−transformations, q or 
q_shift methods, perform the lowest communication overhead, regardless the num-
ber of dimension space and the target architecture.
4.4 A Closer Analyses on $Q^+$—transformations

It is the scope of this section to take a deep investigation on the flow of information that is exchanged between processes at the concurrent level. In particular we want to analyse if the total size of messages sent and received is an invariant between the different method implementations.

4.4.1 Analytic Analysis of the Nine Point Stencil

We go back to the cost model of communication we presented for the $\mathcal{LC}$ language. In the previous Chapter, exploiting the model, we analyzed the communication overheads of the naive and shift methods in the case of the nine point stencil and its multidimensional extensions. We report here the resulting formulas.

\[
T_{\text{com}}^{\text{naive}} = (3^n - 1) \cdot (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \cdot s_{\text{total}}^{n \_\text{dim} \_\text{naive}}
\]

\[
T_{\text{com}}^{\text{shift}} = (2 \cdot n) \cdot (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \cdot s_{\text{total}}^{n \_\text{dim} \_\text{shift}}
\]

When we compared the results we claimed that the two quantities $s_{\text{total}}^{n \_\text{dim} \_\text{naive}}$ and $s_{\text{total}}^{n \_\text{dim} \_\text{shift}}$, which represent the sum of the size of all the sent messages, are equal and that the only difference between the two methods resides in the communication setup components. In the following we refer to the sum of the size of all the sent messages as the outgoing information flow.

If we extend the analysis of communication overhead also to $q$ and $q\_\text{shift}$ methods we get the following results:

\[
T_{\text{com}}^{q} = (2^n - 1) \cdot (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \cdot s_{\text{total}}^{n \_\text{dim} \_q}
\]

\[
T_{\text{com}}^{q \_\text{shift}} = (n) \cdot (t_{\text{snd}}^{\text{setup}} + t_{\text{rcv}}^{\text{setup}}) + t_{\text{transm}} \cdot s_{\text{total}}^{n \_\text{dim} \_q \_\text{shift}}
\]

In the case of the nine point stencil we can claim that the outgoing flow is an invariant for all the four method:

\[
s_{\text{total}}^{n \_\text{dim} \_\text{naive}} = s_{\text{total}}^{n \_\text{dim} \_\text{shift}} = s_{\text{total}}^{n \_\text{dim} \_q} = s_{\text{total}}^{n \_\text{dim} \_q \_\text{shift}}
\]

Therefore the explicit routing of the information between processes that is defined by Plimpton’s shift method does not imply any extra data exchange with respect to the original stencils. As we see in the next section this is not true for all the stencils.

4.4.2 The Jacobi Case

We have seen many times the Jacobi stencil in the previous Chapter. We briefly analyze it in the light of the previous implementation methods. The shape of the
Figure 4.17: Shape 4.17(a) of the Jacobi stencil. Incoming (fig. 4.17(b)) and outgoing (fig. 4.17(c)) communications of a \textit{naive} implementation at the concurrent level.
4.4. A CLOSER ANALYSES ON $Q^+$-TRANSFORMATIONS

Figure 4.18: Shape 4.18(c) of the $Q^+$ [Jacobi] stencil. Incoming (fig. 4.18(d)) and outgoing (fig. 4.18(e)) communications of a $q$ implementation at the concurrent level. Incoming (fig. 4.18(a)) and outgoing (fig. 4.18(b)) communications of a $q$–shift implementation at the concurrent level.
Jacobi stencil is reported in Figure 4.17(a) and a representation of its communication pattern at the concurrent level is shown in Figure 4.17(b) and Figure 4.17(c).

It easily to see that, differently from the nine point stencil, Jacobi does not feature information exchange between neighbours. Hence, the naive method is the only one that can be exploited to implement the stencil at the concurrent level. More precisely the shift and the naive method result in the same implementation.

As presented in Figure 4.17(b) and Figure 4.17(c) for each step four incoming and four outgoing communications are required. No other optimization can be introduced with the classical approach.

Instead, if we consider relaxed safe solution we can exploit both $q$ and $q_{\text{shift}}$. This last does not result in the same implementation of $q$, because for the stencil $Q^+[Jacobi]$, whose relative shape is shown in Figure 4.18(c), a generic partition features information exchange also with some diagonal neighbours.

Exploiting this two methods, both of which come from $Q-\text{transformations}$, we can implement the stencil with respectively only three and two communications as it is shown in Figure 4.18(a) and Figure 4.18(b) for the $q$ method and in Figure 4.18(a) and Figure 4.18(e) for the $q_{\text{shift}}$ one.

The results that we get about the number of communications of the Jacobi stencil are aligned with the theoretic results we presented and proved. Nevertheless, it is worth pointing that in the case of Jacobi we have that the outgoing information flows associated to one step of the $q$ and $q_{\text{shift}}$ are higher that the one of the naive.

Consider the example of Jacobi of Figure 4.17 and Figure 4.18, where square partition are exploited. Suppose that each partition features $l$ elements for each edge. In this configuration we can assert the following.

\[
\begin{align*}
S_{total}^{n_{\text{dim naive}}} &= 4 \ast l \\
S_{total}^{n_{\text{dim shift}}} &= 4 \ast l \\
S_{total}^{n_{\text{dim q}}} &= (4 \ast l) + 1 \\
S_{total}^{n_{\text{dim q shift}}} &= (4 \ast l) + 2
\end{align*}
\]

The cost of exploiting the $q$ method is the augment the outgoing information flow of one element with respect to the naive, while it is of two elements when exploiting the $q_{\text{shift}}$. Because the augment can not be grater than the shape volume, it provide a negligible impact.

In Figure 4.19 we report the charts of the performance registered by the Jacobi example implemented in the three methods. As it is evident, the augment of the flow does not impact. The optimization associated to $Q-\text{transformations}$ still provide the best performance.
4.4. A CLOSER ANALYSES ON $Q^+\text{TRANSFORMATIONS}$

Figure 4.19: Jacobi stencil in a two dimension space, performed on Cell B.E. IBM multicore exploiting the *MammuT* implementation of $L\mathcal{C}$. 
4.5 Negative $Q^-$-transformations

In the previous Section we have introduced positive $Q^-$-transformations. From them it is possible to define some optimizations, i.e. $q$ and $q_{shift}$ methods, that reduce the number of communications require to resolve partition dependencies.

Those optimizations are based on relaxed safe transformations, therefore the output of the optimized program is equal to the original one, except for a different organization of the valuea in the spatial structure (See Definition 2.5.2).

We recall that from the Equation (4.3) of the proof of Theorem 4.1.1 we have:

$$M^t_{\psi}[e] = M^t_{Q^{-}}[e - i \ast q]$$

This is the relation that links the positions of the values in the data structure of the original stencil to the ones of the transformed stencil. In Figure 4.20(a), we show the evolution of the element positions in the working domain step after step for a nine point stencil implemented according the $q$ method. The generic element translate in each step of a quantity define by the vector $q^+$. It is evident that a complete rearrangement of all the elements is necessary in order to report all the elements in their original positions. Two possible solutions to the problem are the following.

I A sequential post processing computation can rearrange elements in their original positions according to Equation (4.2).

II The features of toroidal space can be exploit to solve the problem. A number of steps, after the end of the computation, can be performed without modifying any value, until the elements are again in the initial positions. This approach implies the execution of a number of "empty" steps that depends both on the lengths of the domain and on the number of steps previously performed by the application.

4.5.1 Defining $Q^-$-transformations

For the rearrangement problem, a powerful alternative is possible for the class of those stencils featuring a relative shape with a central symmetry with respect to the origin. Most of the stencils presented in literature belong to this class, as Jacobi and the nine point stencil.

This alternative solution, which requires at most one "empty" step to rearrange the domain elements, consists in the definition of a transformation featuring antithetical characteristics with respect to positive $Q^-$-transformation.

**Definition 4.5.1 (Negative $Q^-$-transformation).** Let $\psi$ be a $\mathcal{HUA}$ stencil. A negative $Q^-$-transformation transforms $\psi$ in the $\mathcal{HUA}$ stencil $Q^-[\psi]$ which is
4.5. NEGATIVE $Q^{-}$TRANSFORMATIONS

Figure 4.20: Evolution of the element value positions in the spatial structure in the case of a Jacobi stencil defined over a non-axial domain space: 4.24(a) with positive $Q^{-}$transformation, 4.24(b) with the interleaving of positive and negative $Q^{-}$transformation.
equivalent to the original one, except for the step model that defined as follows:

\[
\forall e \in \mathcal{M} \xrightarrow{Q^{-}} (\mathcal{F}^\psi, \mathcal{S}^{Q^{-}|\psi})
\]

\[
q^- = <q^-_1, \ldots, q^-_{\dim}>
\]

\[
q^-_i = + \max \{\beta^* \epsilon_i \forall \beta \in \mathcal{R}\}
\]

\[
\mathcal{R}^{Q^{-}|\psi} = \mathcal{R}^\psi + q
\]

\[
\mathcal{S}^{Q^{-}|\psi} = e + \mathcal{R}^{Q^{-}}
\]

\[
M^{i+1}_{Q^{-}|\psi}[e] = \mathcal{F}_i(M^{i}_{Q^{-}|\psi}[e + \gamma_1], \ldots, M^{i}_{Q^{-}|\psi}[e + \gamma_n])
\]

where \(\epsilon = \{\epsilon_1, \ldots, \epsilon_{\dim}\}\) is the set of the vectors in the natural basis of \(\mathbb{N}^{\dim}\) and \(\beta^* \epsilon_i\) is the scalar product which returns the component of the vector \(\beta\) along the main space direction expressed by the vector \(\epsilon_i\). The vector \(q^-\) is called the negative vector.

The key idea is the same of positive \(Q^{-} - transformations\), but the fundamental geometric feature of the negative ones is that all the components of the \(\gamma^-\) are not positive instead of not negative. Once again we exploit the Jacobi as tutorial stencil.
and we reported in Figure 4.21 the representation of both Jacobi and $Q^-$ relative shapes and in Figure 4.22 and Figure 4.23 the corresponding sequential pseudo codes.

4.5.2 Combining Positive and Negative $Q^-$ transformations

Stencils with central symmetric shapes present the following nice characteristic:

$$q^- = -q^+$$

In other words, a component of $q^-$ is the negative of the corresponding one of $q^+$. This feature can be exploited to reduce the effect of the element movements, introduced by $Q^-$ transformations, as presented in the following Theorem.

**Theorem 4.5.1 (Interleaving Negative and Positive $Q^-$ transformations).** Considering a generic domain space $M$, let $M_i^\psi$ be the state of the domain when applying $i$ times a generic stencil $\psi$, featuring a central symmetry. Let then $M_{Q_i}^\psi$ be the state of the domain when applying the stencil $Q^+(\psi)$ on odd steps and the stencil $Q^-(\psi)$ on even ones.

At the beginning of odd steps all elements are in their original positions, while at the beginning of even ones all elements are translated of a quantity $q^-$. 

**Proof.** Let the $M^0$ be the state of the domain before applying any stencil, we have

$$M^0 = M_\psi^0 = M_Q^0$$

At the first step we apply the positive $Q^-$ transformation: $M_{Q_1}^\psi = M_{Q_1}^0$. Considering the Equation (4.3), which is at the core of the proof of Theorem 4.1.1 for general $Q^-$ transformations relaxed safety, we obtain the following Equation for positive $Q^-$ transformations:

$$M_{\psi}^1[e] = M_{Q}^1[e - q^+]$$

After the first step the domain elements are translated with respect to the original positions of a quantity $q^-$, indeed, because of the central symmetry of the stencil $\psi$, we have $-q^+ = q^-$. 

At the second step the negative $Q^-$ transformation is applied and so always for Equation (4.3) at page 106 for negative $Q^-$ transformations, we get:

$$M_{\psi}^2[e] = M_{Q}^2[(e - q^+) - q^-] = M_{Q}^2[e + q^- - q^-] = M_{Q}^2[e]$$

The domain elements after two steps, with a positive and a negative $Q^-$ transformations, are back in the original positions.
Iterating the reasoning, we obtain:

\[ M'_\psi[e] = M'_Q[e + (\text{mod}_2(i) \ast q^-)] \]

At the end of even steps the data are in the original positions, while in odd steps they are translated of a quantity \( q^- \).

A graphical presentation of the results coming from Theorem 4.5.1 is given in Figure 4.20(b) for the case of the nine point stencil in a two dimensional space.

From the previous property it is clear that, exploiting the interleaving of positive and negative \( Q^-\) transformations, the domain does not need any rearranging when the application ends after an even number of steps. In the other cases, one "void" step is sufficient.

It is worthwhile highlighting that negative \( Q^-\) transformations feature the same characteristics of positive \( Q^-\) transformations in terms of impact on partition dependencies. Therefore as from the positive, it is possible to define \( q \) and \( q\) shift method for negative \( Q^-\) transformations. This means that while interleaving positive and negative \( Q^-\) transformations, we still exploit the optimization on the number communications in each step.

### 4.6 Extending \( Q^-\) transformations to The Semi Uniform

The constraint of toroidal domain space featured by \( \Euclid \) stencil makes translations exploited by \( Q^-\) transformations always possible. More precisely, the toroidal space guarantees that, referring to Equation (6.2) at page 191, the generic element \( e + \gamma_\alpha \) (which is the element \( e \) translated by \( q \)) is always in \( M \).

Concerning \( \Euclid \) stencil belonging to the semi uniform class, all the \( Q^-\) transformation formalization can be redefine if we consider the extended version of the stencil. We recall that the existence of this extension is a necessary condition for a stencil to be classified as semi uniform. Therefore, all the results we have obtained up to now can be directly reused on the extended version because it is a \( \Euclid \) stencil.

One of the most important results in the extension of \( Q^-\) transformations to semi uniform stencils is the fact of targeting also stencils which are not defined over a toroidal space; for instance the Jacobi stencil we presented in Figure 2.17 at page 55.

Figure 4.24 represents the evolution of the values of the bound elements, which are highlight in gray, in the spatial structure for the case Jacobi. The Figure compares both the techniques: the one that exploits only positive \( Q^-\) transformations and the other that uses the interleaving method.
4.6. EXTENDING Q–TRANSFORMATIONS TO THE SEMI UNIFORM

Figure 4.24: Evolution of border elements for the Jacobi stencil defined over a non-toroidal domain space: 4.24(a) with positive Q− transformation, 4.24(b) with the interleaving of positive and negative Q− transformation.
4.7 Conclusions

In this Chapter we presented and formally proved the powerful features of \(Q-\text{transformations}\): a set of transformations applicable to stencils whose data dependencies can be represented in terms of affine space translations.

In stencil based parallel applications, communications represent the main overhead, especially when targeting a fine grain parallelization in order to reduce the completion time. Techniques that minimize the number and the impact of communications are clearly relevant.

We proved that the reduction of the number of communications featured by \(Q-\text{transformations}\) based optimizations is greater than those provided by methods presented in literature.

Moreover our experiments, both on multi-core and cluster architectures, show that implementations exploiting \(Q-\text{transformations}\) perform the lowest communication overhead when targeting fine grain parallelizations.
Chapter 5

Step Fusion Transformations

Abstract

The aim of this Chapter is the introduction of a new and powerful class of stencil transformation called $Q-$Step-$Fusion$ ($QSF$).

Data parallelisms is based on data distribution and function replication. A well known technique, called $ghost$ cell expansion, aims the reduction of the communication overheads though the exploitation of data replication. The transformation, which in literature is expressed as a transformation at the concurrent level, introduces a interesting thread-off between the reduction of communication overheads and the increase of the computational load.

The $QSF-$transformations expand and restructure the main concept of the $ghost$ cell expansion up to the definition of a new class of stencil transformations which are defined at the functional dependency level instead of at the concurrent one. The new in-deep point of view of the $ghost$ cell expansion technique, makes possible the exploitation of both of the results of $Q-$transformations, for communication overhead reduction, and of new optimizations which are focused on lowering the computation load.

In the Chapter we analytically demonstrate the benefit of the new transformations and we validate the results with a complete set of experiments on different kind of computational architectures.

The Chapter is structured as follows. Section 5.1 introduce the oversending technique and explain how data replication is exploited to reduce the communication overhead.

Section 5.2 introduce $SF-$transformations, which come from the revisitation of oversending as a stencil transformation defined at the functional dependency level.

Finally, Section 5.3 analyses the performance improvements of $SF-$transformations on sequential computational kernels that comes from effects of temporal locality.
## Contents

**5.1 Data Replication in HLA Stencil**  
5.1.1 The Oversending Method 143  
5.1.2 Oversending Performance Model 145  
5.1.3 Oversending and Q-transformations 149

**5.2 Step Fusion Transformations**  
5.2.1 A Structured Interpretation of Oversending 152  
5.2.2 Formal Definition of $SF$ Transformations and Their Properties 154  
5.2.3 Step Fusion for Linear Step Functions 158  
5.2.4 $SF$ and Oversending 158

**5.3 $SF$ and Sequential Computations**  
5.3.1 Relation between Shape Cardinality and $SF$ Level 161  
5.3.2 Temporal Locality Factor in $SF$-transformations 162  
5.3.3 Asymptotic Analysis of both Computations and Communications 163  
5.3.4 Taking into account Cache Memory Hierarchy 168  
5.3.5 Experimental Results 169  
5.3.6 Conclusions 174
5.1 Data Replication in HUA Stencil

The data parallel paradigm is based on data partitioning and replication of functions. In the terms of the HUA model, the working domain is distributed among multiple processes and the computational kernels, represented by the step functions, are replicated.

In the previous Chapters we have seen the overheads that in data parallel programs is introduced by communications between processes. We have presented a set of optimizations (some already cited in literature but other new as the \(Q\)-transformations) targeting the reduction of those overheads.

A common characteristic of all the optimization methods is the exploitation of some strategies that pack smaller incoming or outgoing messages together. The benefit comes from the reduction of the setup time component of the communication overheads.

Ding and He in [15] propose a technique that we call overspending, but it is also known as Ghost Cell Expansion. The technique, which is expressed at the concurrent level, exploits data replications to reduce the mean number of communications per step.

In Sections 5.1.1 and Section 5.1.2 we present the oversending method and we analyze its drawbacks. Moreover we compare the new technique with the \(Q\)-transformations and we present a fusion of the two transformations.

5.1.1 The Oversending Method

We start the explanation of the oversending method exploiting the Jacobi stencil example in a two dimension space. For sake of simplicity, we break a clause of the Working Hypothesis 3.1 and we consider a row partitioning.

In a row partitioning the naive and shift methods are equal, therefore in the rest of the Section we can consider only the naive one. Nevertheless all the following issues can be easily extended to block partitioning independently from the fact that a naive or a shift method has been targeted.

The pseudo code of a program that at the concurrent level implements the Jacobi stencil exploiting a row partition strategy is reported in Figure 5.1. As previously done, instead of making the pseudo code annotations heavier, we relay on Figure 5.5 at page 150 for the burden of graphically highlighting the elements that are involved in the communications.

The oversending method is based on the principle of sending, and symmetrically receiving, more data during some steps in such a way that communications between processes are not required at each stencil step.

In a generic step \(i\), a process exchanges bigger messages with respect to the naive implementation. The messages are composed by two set \(\lambda\) and \(\delta\) of values of elements in \(M\).
$\text{int } c = 512;$
$\text{int } r = 50;$
$\text{double } J_{in}[r][c], J_{out}[r][c];$
load_partition_values($J_{in}$);

for($i_{step} = 0; i_{step} < 10; i_{step}++$)

SEND($P_{(-1)}$); SEND($P_{(+1)}$);
COMPUTE($\text{Incoming independent region}$);
RECV($P_{(-1)}$); RECV($P_{(+1)}$);
COMPUTE($\text{Incoming dependent region}$);
swap($J_{in}, J_{out}$);

return_partition($J_{out}$);

\begin{figure}
\begin{center}
\begin{verbatim}
int $c = 512$;
int $r = 50$;
double $J_{in}[r][c], J_{out}[r][c]$;
load_partition_values($J_{in}$);

for($i_{step} = 0; i_{step} < 10; i_{step}++$)

SEND($P_{(-1)}$); SEND($P_{(+1)}$);
COMPUTE($\text{Incoming independent region}$);
RECV($P_{(-1)}$); RECV($P_{(+1)}$);
COMPUTE($\text{Incoming dependent region}$);
swap($J_{in}, J_{out}$);

return_partition($J_{out}$);
\end{verbatim}
\end{center}
\end{figure}

\textbf{Figure 5.1:} Representation in pseudo code of a Jacobi stencil implemented at the concurrent level, exploiting a naive method and a row partitioning strategy.

I The $\delta$ set represents the same information that is exchanged though the messages that are defined by the \textit{naive} method. The values of $\delta$ are required to compute the values of the incoming dependent region at the time $i + 1$.

II The set $\lambda$ is exploited to update the values of $\delta$ elements at the time $i + 1$ too. Indeed, the new updated values of $\delta$ can resolve the functional dependencies of the incoming dependent region also for the step $i + 1$. In such a way it is possible to compute the values at the time $i + 2$ for all the partition elements.

All the received data during one step resolve the partition dependencies for two steps. This important result does not came for free. The elements of $\delta$ have to be computed, or better to be updated, before they can be exploited for the step featuring no communications. The update computation represents the supplementary computational overhead with respect to the naive implementation.

The elements in $\delta$ represent replicated data. Both the sender and the receiver of those data are going to independently update them according to the computational kernel defined by the stencil step function. In other words, the same set of values are computed in the same step by more than one process.

A pseudo code of a Jacobi stencil implementation that has been optimized with the oversending method is reported in Figure 5.2. Once again, we relay on Figure 5.5
at page 150 for the burden of graphically highlighting the elements that are involved in the communications.

The oversending, as it is clear from the method description in pseudo code, increases the data sent in one step in order to replace the communications of the following step with some computations. The method can be extended in order to avoid communications in more than one consecutive steps. Hence, we say that the oversending method features different levels of optimizations.

The levels give to the method, differently from the previous ones, a parametric feature: it is possible to decide how long is the sequence of steps that do not perform communications. Nevertheless, the higher the level of optimization is (i.e. the more the number of steps that avoid communications is), the more the extra computation load is. This relation crate an interesting trade-off between communication and computation.

5.1.2 Oversending Performance Model

We now focus on the performance studies of the oversending method. We analyze computation and communication times. For the sake of simplicity we do not take in consideration the possible overlapping between the two components.

In Figure 5.3 we report a graphical representation of the evolution of data structures and communications of the oversending program of Figure 5.2. The representation can be used to easily understand how the performance model of the program is extracted.

Starting with the computation component, we have that in two steps, of which only one features communications, a process computes two times the elements of its partition and moreover it updates the elements of two received rows.

Consider $t_{\text{one-el}}$ the time to update one single element according to the Jacobi stencil and moreover let the variables $r$ and $c$ be respectively the number of rows per partition and the number of elements in a row. We can model the mean computation time for a single Jacobi step as follows:

$$T_{\text{comp}} = t_{\text{one-el}} \cdot c \cdot (r + 1)$$

About the communication component, each process features two send and two receive operations, whose associated messages consist in two rows of elements. The send and receive operations are computed only in one step out of two. Therefore, referring to the size of a row with the term $s(R)$, we can model the mean communication overheads for the single Jacobi step as follows:

$$T_{\text{com}} = (t_{\text{snd}} + t_{\text{rcv}}) + 2 \cdot t_{\text{transm}} \cdot s(R)$$

We extend the model for the Jacobi example to a generic level $L$ of oversending. As convention, we consider $L = j$ the level of optimization where only one step out of $j$ features communications.
int  c = 512;
int  r = 50;
double  J_{in}[r][c], J_{out}[r][c];
load_partition_values(J_{in});

for(i_{step} = 0; i_{step} < 10; i_{step} + +){
    if(i_{step}%2 == 0){
        SEND(P_{0}); SEND(P_{+1});
        COMPUTE(Incoming independent region);
        RECV(P_{0}); RECV(P_{+1});
        COMPUTE(Incoming dependent region);
    } else{
        SEND(P_{0}); SEND(P_{+1});
        COMPUTE(Incoming independent region);
        UPDATE_REPLICATED_DATA();
        COMPUTE(Incoming dependent region);
    }
    swap(J_{in}, J_{out});
}
return_partition(J_{out});

Figure 5.2: Representation in pseudo code of a Jacobi stencil implemented at the concurrent level, exploiting a oversending method and a row partitioning strategy.
5.1. DATA REPLICATION IN HUA STENCIL

Figure 5.3: Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized with the oversending method according to the pseudo code in Figure 5.2.
Figure 5.4: Representation of the evolution of the data structure elements and communications during the computation of a Jacobi stencil that has been optimized with the q-oversending method.
5.1. DATA REPLICATION IN \( \mathcal{HUA} \) STENCIL

\[
T_{\text{comp}} = t_{\text{one-el}} \cdot c \cdot (r + \mathcal{L} - 1)
\]

\[
T_{\text{com}} = \frac{(t_{\text{send}} + t_{\text{recv}})}{\mathcal{L} - 1} + 2 \cdot t_{\text{trans}} \cdot s(R)
\]

From the previous formula the following two main aspects come out.

I As all the previous methods for communication optimization, the oversending one target the reduction of the setup overhead of communications. The mean flow, i.e. the amount of data sent or received, per step do not decrease.

II The reduction of the communication overheads comes from the increasing of computation load: more elements have to be updated. Therefore, for each configuration there is an optimum value of \( \mathcal{L} \). The value depends on the performance of the physical architecture and on the computational characteristics associated to the target stencil.

5.1.3 Oversending and \( Q - \) transformations

In Table 5.1, we compare the performance results of the oversending method for the Jacobi example with the ones of naive and \( q \) methods. Figure 5.5 shows for each of the compared methods the communication pattern.

We recall that because for the sake of simplicity we are considering row partitioning strategy, the implementation of the \( q \) and the \( q - \) shift methods are equivalent.

Observing accurately the data reported in the Table 5.1, it comes out that the performance of the \( q \) method resumes the advantages featured by the other two methods. More formally we can write the two following equations:

\[
T_{\text{com}}^q = \min \{ T_{\text{com}}^{\text{oversending}}, T_{\text{com}}^{\text{naive}} \}
\]

\[
T_{\text{comp}}^q = \min \{ T_{\text{comp}}^{\text{oversending}}, T_{\text{comp}}^{\text{naive}} \}
\]

Therefore we can claim that in the case of row partitioning, the \( q \) method get the same performance improvement of the oversending one without increasing the computation time. We can extend the previous observation with the following Theorem.

**Theorem 5.1.1 (Comparing Oversending and \( Q - \) transformations).** In the case of block partitioning and with a stencil \( \psi \) that features partition dependencies with all its neighbours, the \( q - \) shift method obtains the same reduction of communication overheads of the oversending one without introducing extra computational load.
CHAPTER 5. STEP FUSION TRANSFORMATIONS

**Figure 5.5:** Graphical representation of the communication patterns of *naive*, *q*, *oversending* and *q − oversending* methods exploited to implement a Jacobi stencil in a row partition configuration.

<table>
<thead>
<tr>
<th>Jacobi</th>
<th>$T_{com}$</th>
<th>$T_{comp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>$2 \ast (t_{snd} + t_{rcv}) + 2 \ast t_{transm} \ast s(R)$</td>
<td>$t_{one_el} \ast c \ast r$</td>
</tr>
<tr>
<td>Oversending($L=2$)</td>
<td>$\left( t_{snd} + t_{rcv} \right) + 2 \ast t_{transm} \ast s(R)$</td>
<td>$t_{one_el} \ast c \ast \left( r + 1 \right)$</td>
</tr>
<tr>
<td>q</td>
<td>$\left( t_{snd} + t_{rcv} \right) + 2 \ast t_{transm} \ast s(R)$</td>
<td>$t_{one_el} \ast c \ast r$</td>
</tr>
<tr>
<td>q − oversending($L=2$)</td>
<td>$\left( \frac{t_{snd} + t_{rcv}}{2} \right) + 2 \ast t_{transm} \ast s(R)$</td>
<td>$t_{one_el} \ast c \ast \left( r + 1 \right)$</td>
</tr>
</tbody>
</table>

**Table 5.1:** Performance model of the communications and computations of *naive*, *q*, *oversending* and *q − oversending* methods exploited to implement a Jacobi stencil in a row partition configuration. The parameter are considered as mean time per step.
Proof. We known that the \( q-shift \) method implies, for a stencil \( \psi \) that is defined over a \( n \) dimension space and features partition dependencies with all its neighbours, a number of communication equal to \( n \).

The oversending method, with a optimization level \( L \) equal to two, requires communications at each step out of two. If we suppose to exploit along with the oversending the shift method for the reduction of diagonal communications, the resulting implementation requires a number of communication equal to \( 2 \times n \) each two steps. Therefore on average it requires \( n \) communication per step.

Therefore both the two solutions feature the same number of communication per step. Nevertheless the solutions associated to the oversending methods also introduces extra computational overhead in order to update replicated data.

As we stressed at the beginning of the Section, the oversending method is an optimization that in literature is defined at the concurrent level. We can therefore mix \( Q-transformation \), which are defined at the functional dependency level, with the oversending one without any problem. We call the resulting method \( q - oversending \).

In the case of Jacobi stencil, a graphical representation of the communication pattern and the communication implied by \( q - oversending \) method is reported in Figure 5.4 and its performance model is reported on the last row of Table 5.1.

As result, the new method halves the communication setup with respect of both the other two methods and increase the computation load as the oversending one. Thanks to the multiple level of optimization, The \( q - oversending \) method enhances both \( q-shift \) and \( q \) with a parametric feature that they do not have.
5.2 Step Fusion Transformations

In the previous Sections we have introduced and analyzed the oversending method as it is defined in literature: an optimization at the concurrent level for communications. In this Section, we focus on the definition of a new model for oversending. Indeed, we are going to redefine the method as a stencil transformation at the functional dependency level. As we will see later in the Chapter, this solution provides on sequential computation kernels new optimizations which better exploit cache hierarchy with a higher temporal locality.

5.2.1 A Structured Interpretation of Oversending

We introduce the \textit{Step–Fusion} transformations (SF), which are a structured redefinition of the oversending technique. The main concept of the new transformations is to merge one or more steps of a \textit{HUAA} stencil into a single one.

Consider the Equation that is at the core of the classic definition of the \textit{Jacobi} stencil:

\[
\mathcal{M}^i[x][y] = \left( \mathcal{M}^{i-1}[(x, y + 1)] + \mathcal{M}^{i-1}[(x, y - 1)] + \mathcal{M}^{i-1}[(x + 1, y)] + \mathcal{M}^{i-1}[(x - 1, y)] \right) / 4
\]  

(5.1)

The formula claims that the computation of an element value at the time \( i \) depends on the evaluation of some elements at the time \( i - 1 \). In turn, the requested values can be indirectly estimated applying the definition of the \textit{Jacobi} stencil and exploiting the evaluations of elements at time \( i - 2 \).

For example the value \( \mathcal{M}^{i-1}[(x, y + 1)] \), which is a component of the Equation (5.1), can be express as:

\[
\mathcal{M}^{i-1}[(x, y + 1)] = \left( \mathcal{M}^{i-2}[(x + 1, y + 1)] + \mathcal{M}^{i-2}[(x - 1, y + 1)] + \mathcal{M}^{i-2}[(x + 2, y)] + \mathcal{M}^{i-2}[(x, y + 2)] \right) / 4
\]

The new \textit{Jacobi} Equation (5.2) can be interpreted as a new alternative stencil which fuses two steps of the original \textit{Jacobi} into one.
5.2. STEP FUSION TRANSFORMATIONS

Figure 5.6: Graphical representation of the step fusion transformations which is applied to a Jacobi stencil.
The procedure can be iterated more times, therefore it is possible to define a
generic level of fusion. Merging $k$ steps of a stencil we can compute the values of
the working domain elements from the knowledge of their values at time $i - k$ as it
follows.

$$M^i[x][y] = \mathcal{F} \left( M^{i-k}[..][..] \ldots M^{i-k}[..][..] \right)$$

We model this new stencil, that come from the fusion of one or more steps of
the original one, as the result of a transformation that is defined at the functional
dependency level: $SF$-transformation.

### 5.2.2 Formal Definition of $SF$ Transformations and Their
Properties

**Definition 5.2.1** *(Step Fusion transformations).* The $SF^k$ is Step Fusion of
$k$ levels, where a generic stencil $\psi$ (expressed into the $HUA$ model) is transformed
into the stencil $SF^k[\psi]$, recursively defined as:

$$
\begin{align*}
SF^k[\psi] &= SF^2[SF^{k-1}[\psi]] \\
SF^2[\psi] &= SF^2[\psi] \\
SF^1[\psi] &= \psi
\end{align*}
$$

$SF^2[\psi]$ is a specific support transformation which depends on a defined stencil, in
this case $\psi$, and maps $HUA$ stencils to $HUA$ stencils.

**Definition 5.2.2** *(Step Fusion Support Transformations).* Let $\psi$ and $\chi$ be
two $HUA$ stencils which are respectively characterized by the couples $(F^\psi, S^\psi)$ and
$(F^\chi, S^\chi)$ and let be both $S^\chi(e) = \{g_1, \ldots, g_n\}$ and $S^\psi(e) = \{\omega_1, \ldots, \omega_m\}$. The transformed stencil $SF^2[\chi]$ is a $HUA$ stencil characterized by the following step model.

$$
\forall e \in M, SF^2[\chi] = SF^2[SF^2[\chi]]
$$

$$
\begin{align*}
SF^2[\psi][S^\chi] &= \left\{ \forall \omega \in M | \omega \in \bigcup_{g_\alpha \in S^\chi(e)} S^\psi_{i-1}(g_\alpha) \right\} \\
M^{i+1}[e] &= \mathcal{F}_i \left( \mathcal{F}^{\psi}_{i-1}(M^{i-1}[^\omega_1], \ldots, M^{i-1}[^\omega_m]), \ldots, M^{i-1}[^\omega_1], \ldots, M^{i-1}[^\omega_m] \right) \\
&\text{where } \{\omega_1, \ldots, \omega_m\} = S^\psi_{i-1}(g_\alpha)
\end{align*}
$$
By definition of a HUA stencil, we have that at the step $i + 1 \chi$ updates an element $e$ of the working domain with the result of the computation of the step function $F^\chi$ with the evaluations of the $S^\chi(e)$ set elements as parameters. In this scenario, $SF^\psi_2[\chi]$, a support transformation for the $\psi$ stencil applied to $\chi$, defines the shape set $SF^\psi_2[S^\chi_i]$ replacing each element $g_\alpha$ in $S^\chi(e)$ with the domain element set $S^\chi(g_\alpha)$. $S^\chi(g_\alpha)$ is the domain element set which defines the shape of $\psi$ featuring $g_\alpha$ as its application point. The resulting set $SF^\psi_2[S^\chi_i]$ constitutes the shape of transformed stencil.

Figure 5.6 graphically presents the previous concept describing how support transformations work when $QSF-$transformations are applied to a Jacobi stencil.

In the case of a two level of step fusion, the stencil that we obtain is $SF^2[Jacobi] = SF^2_{Jacobi}[Jacobi]$, which features nine elements, five more than the original one. One single step of $SF^2[Jacobi]$ corresponds to two iterations of the original Jacobi. Considering the time model of the original Jacobi stencil, we have that, in order to produce element values at time $i + 1$, $SF^2[Jacobi]$ requires values at time $i - 1$.

When applying a $SF$ with three levels of step fusion, we get the sixteen element stencil $SF^3[Jacobi]$, whose single computation step is equivalent to three applications of Jacobi step. Therefore to produce values at the step $i$ the transformed stencil requires domain values at the step $i - 2$.

From Figure 5.6 it is also possible to well appreciate the role of the support transformation. We review the formal definition of support function for a three levels of step fusion applied to Jacobi. By definition we have:

$$SF^3[Jacobi] = SF^2_{Jacobi}[SF^2_{Jacobi}[Jacobi]]$$

Therefore according to the annotations used in the definition, $\psi$ and $\chi$ are respectively Jacobi and $SF^2_{Jacobi}[Jacobi]$, while $S^\psi_i$ and $S^\chi_i$ are the two stencil shapes.

The procedure to obtain $SF^2_{\psi}[S^\chi_i]$ consists in replacing each element in $\chi$ shape ($S^\chi_i$) with the elements of the Jacobi shape ($S^\psi_i$) featuring the replaced element as application point. The union of all the new elements gives as result the shape of the sixteen element stencil defined by $SF^3[Jacobi]$.

In Figure 5.7 and Figure 5.8 we also report the graphical representation of $SF$-transformations for the Laplace and the nine point stencils. As it is evident the higher the level of fusion is the wider the stencil shape becomes.

In all the Figures representing stencil which are the result of $SF$-transformations, a parameter is associated to each shape element. The parameter is the integer part of the coefficient used by the step function. To better understand the previous explanation it is sufficient to compare the coefficients of $SF^2[Jacobi]$ in Figure 5.6 and the Equation (5.2). We just consider the integer part for a sake of simplicity in the graphical representations.

In the following we focus on the definition and on the prove of two significant properties of the $QSF-$transformations.
CHAPTER 5. STEP FUSION TRANSFORMATIONS

Figure 5.7: Graphical representation of $S\mathcal{F}^i$[Laplace] stencils for $i$ from one to three.

Figure 5.8: Graphical representation of $S\mathcal{F}^3$[Nine] stencils for $i$ from one to three.
Theorem 5.2.1 (SF’s Correctness). Let \( \psi \) be a generic stencil and let \( SF^k[\psi] \) its transformation with respect to \( SF \)-transformations. The values of a domain \( M \) after \( k \) steps of \( \psi \) stencil are the equivalent to those after one step of \( SF^k[\psi] \).

Proof. The proof comes from the definition of \( M^{i+1}[e] \) in Equation (5.4).

The previous property asserts that the transformation that we defined behaves as we targeted: it returns a stencil which merges two or more steps of an original stencil into one.

Theorem 5.2.2 (Step Fusion Support Transformations are HUA). Let \( \psi \) and \( \chi \) two generic HUA stencils. \( SF^2[\psi][\chi] \), result of the support transformation of \( \psi \) applied to \( \chi \), is a HUA stencil.

Proof. We demonstrate that the set \( SF^2[\psi][\chi] \) can be expressed in the form \( e + \{ \beta_1, \beta_2, \ldots, \beta_n \} \) where \( \forall \alpha \ (e + \beta_\alpha) \in M \). Exploiting the definition of \( S^\chi_i \) and \( S^\psi_{i-1} \) in the HUA model we get:

\[
S^\chi_i(e) = \{ \alpha_1, \ldots, \alpha_n | \forall j \alpha_j = (e + \beta_{\alpha_j}) \in M \}
\]

\[
\forall \alpha \in S^\chi_i(e), \ S^\psi_{i-1}(\alpha) = \alpha + \{ \delta_1, \ldots, \delta_m \} \text{ where } \forall j \ (\alpha + \delta_j) \in M
\]

\[
= e + \{ \delta_1 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha \} \quad (5.5)
\]

Replacing the Equation (5.5) into the definition of \( SF^2[\psi][\chi] \) (see Equation 5.4), we get:

\[
SF^2[\psi][\chi](e) = \left\{ \forall \omega \in M | \omega \in \bigcup_{\alpha \in S^\chi_i(e)} \left\{ e + \{ \delta_1 + \beta_\alpha, \delta_2 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha \} \right\} \right\}
\]

\[
\overset{\psi}{=} \left\{ \forall \omega \in M | \omega \in e + \bigcup_{\alpha \in S^\chi_i(e)} \left\{ \delta_1 + \beta_\alpha, \delta_2 + \beta_\alpha, \ldots, \delta_n + \beta_\alpha \right\} \right\}
\]

Each \( \omega \in SF[S_i](e) \) has the form \( e + \beta_\omega \); we can conclude that the set \( SF[S_i](e) \) is compatible with the HUA model. Because all the remaining elements are compatible with the HUA, the demonstration is complete.

The previous property is essential for the definition of SF-transformation; it asserts that the support transformation of a HUA stencil maps HUA stencils into HUA stencils. Therefore firstly we can conclude the following:

I The series of transformations defined in Equation (5.3) is well defined.

II The result of a SF-transformation is still a HUA stencil: results of Q-transformations can be directly exploited on the transformed stencils.
5.2.3 Step Fusion for Linear Step Functions

A more specific definition of $\mathcal{SF}$—transformations can be suggested for those stencils whose functions $\mathcal{F}$ are a linear composition of the input parameters.

Exploiting associative and commutative properties, which are featured by linear computations, the function composition defined by the support transformation can be unrolled and presented as a linear combination of domain elements values.

Most of the stencil computations feature linear functions: all the stencil used in partial differential equation solver or in image processing. The same Jacobi, which computes the arithmetic mean over a set of four elements, has a function which is a linear composition of the shape element values. The Equation (5.2) associated to $\mathcal{SF}^2$[Jacobi] shows that a stencil that feature a linear step function is transformed by $\mathcal{SF}$—transformations in a stencil that still feature a linear step function. The linearity is obviously an invariant with respect to the $\mathcal{SF}$—transformations.

Because stencil characterized by linear step function are widespread, we introduce for them a specific definition of the support functions.

**Definition 5.2.3** (Step Fusion Support Transformations for Linear Functions).

$$\forall e \in \mathcal{M} \quad \mathcal{SF}^2_\psi[S_{\chi_i}] \rightarrow \left(\mathcal{F}_{\chi_i}^{\psi}, \mathcal{F}_{\chi_{i-1}}^{\psi}, \mathcal{SF}^2_\psi[S_{\chi_i}]\right)$$

$$\mathcal{SF}^2_\psi[S_{\chi_i}] = \left\{ \forall \omega \in \mathcal{M} | \omega \in \bigcup_{g_{\alpha} \in S_{\chi_i}(e)} S_{\psi}^{\chi_i}(g_{\alpha}) \right\}$$

$$\mathcal{M}^{i+1}[e] = \sum_{\omega_j \in \mathcal{SF}^2_\psi[S_{\chi_i}]} \lambda_j \ast M^{i-1}[^{\omega_j}] \quad (5.6)$$

It is important to notice that the coefficients $\lambda_j$ are constants that can be determined statically when calculating the shape of $\mathcal{SF}^k[\psi]$. In the rest of the Chapter we consider only stencil featuring a linear step function.

5.2.4 $\mathcal{SF}$ and Oversending

Summing what we have seen in the Chapter until now, we first presented the oversending method stressing the that in literature it is expressed at the concurrent level. Then we presented $\mathcal{SF}$—transformations a transformation defined at the functional level which result in stencils that compute more steps in a single one.

What remains now is the demonstration that the communication pattern implied by a stencil transformed with $\mathcal{SF}$—transformations coincides with the one associated to the oversending method.

**Property 5.2.1** ($\mathcal{SF}$—transformations and Oversending). Let $\psi$ be a HUA stencil and $\mathcal{SF}^k[\psi]$ the result of a $\mathcal{SF}$—transformation applied to $\psi$. At the concurrent level, the implementation of $\psi$, according to the oversending method of level
Figure 5.9: Graphical representations of shapes and communication patterns of Jacobi, $SF^2[Jacobi]$, $Q[Jacobi]$, $QSF^2[Jacobi]$ stencils.
$k$, and the one of $\mathcal{SF}^k[\psi]$, according to the naive method, feature the same commu-
nication pattern.

Proof. Oversending avoid communication for a set of consecutive steps. This means
that in the step featured by communications all the information has to be exchanged
to update locally the partition for the following $k - 1$ steps.

The communications of $\mathcal{SF}^k[\psi]$ imply an information exchange to compute $k - 1$
steps of the original stencil.

We can conclude that the information flow is the same in both the implementa-
tion and also the communication pattern. A graphical prove of this can be derived
comparing the communication patterns reported in Figure 5.9 and Figure 5.5. □

Finally for completeness we define, as we symmetrically did for oversending, a
new transformation that mix both $\mathcal{SF}$—transformations and $\mathcal{Q}$—transformations.
The new transformations, that we call $\mathcal{QSF}$—transformations, feature the same
communication performance model of the method that we called $q$—oversending.
5.3 $S\mathcal{F}$ and Sequential Computations

In the previous Section we presented the class of $QSF$–transformations, where transformed stencils are found from the original ones by merging two or more steps into one. In this section we focus the discussion on the benefit of exploiting $QSF$–transformations in sequential conditions.

The class of $QSF$–transformations, in which stencil can be transformed with different levels of step fusion, provides different sequential and parallel implementations of the same stencil. For example, one way to implement Jacobi is the classical one, that is exploiting the four elements stencil. Another version is the transformed stencil $SF^2[Jacobi]$, whose single step is equivalent to two steps of the classical Jacobi. Other more implementations can be defined by rising the level of step fusion. We therefore are interested in the features of the different implementations and in particular we study when a transformed stencil can provide, on sequential executions, a greater speed up, with respect to the classic implementation.

We recall that in this Section we consider only stencils featuring a linear step function.

5.3.1 Relation between Shape Cardinality and $S\mathcal{F}$ Level

We start the studies of $S\mathcal{F}$–transformations from the analysis of the relation between the shape cardinality and the $S\mathcal{F}$ levels. This relation is important because the computation load, as it is easily to understand, in most cases depends linearly with the number of elements of the shape.

Table 5.2 reports the number of elements of the three stencils analyzed with different level of step fusion. We remember that by definition we have $SF^1[\psi] = \psi$.

<table>
<thead>
<tr>
<th></th>
<th>Jacobi</th>
<th>Laplace</th>
<th>Nine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SF^1$</td>
<td>4</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>$SF^2$</td>
<td>9</td>
<td>13</td>
<td>25</td>
</tr>
<tr>
<td>$SF^4$</td>
<td>16</td>
<td>25</td>
<td>49</td>
</tr>
<tr>
<td>$SF^4$</td>
<td>25</td>
<td>41</td>
<td>81</td>
</tr>
</tbody>
</table>

Table 5.2: Number of elements of the stencils resulted by the applications of different $S\mathcal{F}$–transformations to Jacobi, Nine and Laplace

To better understand the trend of the relation between levels and shape elements for different levels of fusion and different stencils, we refer to Figure 5.10 where the
Element Increasing Factor (EI factor) is plotted for Jacobi, Nine and Laplace. The EI is defined parametrically respect to the level of fusion as:

\[ E_{I}(i) = \frac{|S_{F}[S^{i}(\psi)]|}{|S_{F}[S^{i-1}(\psi)]|} \]

In the chart the dotted line represents the bisector of the first quadrant. We can see that the more the fusion level increases the more the curves move away from the bisector.

Let \( \psi \) be one of the three stencil that we have plotted. If the shape of \( \psi \) features a cardinality equal to \( s \) than the shape of \( S_{F}[\psi] \) features a number of elements that is higher than \( k \times s \).

Recalling that the computation load is linked to the shape cardinality, we can assert that, in order to update the working domain to a certain time, the higher the step fusion level is the more computations are required.

**Figure 5.10:** Element Increasing Factor for Jacobi, Laplace and Nine.

### 5.3.2 Temporal Locality Factor in \( S_{F} - transformations \)

If the computation overhead increases with the increasing of the level of step fusion, there is another component that has to be taken into account: the temporal locality.

A transformed stencil features a different shape, that means that the function dependencies are changed.
The figure 5.11(a) analyzes the computation of a Jacobi along a matrix row. The temporal locality is easy to highlight considering the effect on registers.

While for the computation of the first and the second elements of the row all the shape elements have to be loaded, for the remaining computations only three elements of the shape can be loaded: indeed one value has been already load.

Similarly for $SF^2[Jacobi]$, after a small transitory, the computation saves four loads operations. In general, as presented in Figures 5.12(a),5.12(c) and 5.12(b) for different stencils, the number of reused elements can be establish subtracting from the total number of elements the number of rows on which the stencil is scattered. This number depends linearly with the level of fusion.

We can conclude that there are two important factors that are at stake: both of them increase with the level of step fusion but only one impacts positively on performance. A model to study the results of the two factors is presented in the next Section.

### 5.3.3 Asymptotic Analysis of both Computations and Communications

We focus on a really simple model to predict some properties of the performance trend featured by transformed stencils. We consider an architectural model with an infinite number of registers, a memory and no cache hierarchy.

Let $T_{el}^\psi$, the mean time required to compute one element according to a stencil $\psi$, be given by the equation

$$T_{el}^\psi = T_{mem}^\psi + T_{op}^\psi$$

The entity $T_{mem}^\psi$ is the time spent in loading operations while $T_{op}^\psi$ in arithmetic ones. $T_{mem}^\psi$ and $T_{op}^\psi$ are respectively proportional to the number of load operations ($N_{load}$) and of the elements of the stencil shape ($|S^\psi|$). Therefore, we define

![Graphical representation of time locality for Jacobi and SF\{Jacobi\} stencils](image)

**Figure 5.11**: Graphical representation of time locality for Jacobi and SF\{Jacobi\} stencils
CHAPTER 5. STEP FUSION TRANSFORMATIONS

![Graphical representation of time locality for different stencils](image)

Figure 5.12: Graphical representation of time locality for different stencils
Figure 5.13: Performance trend for $SF$–transformations for Jacobi 5.13(a) and Laplace 5.13(b)
\[ T_{\text{mem}}^\psi = \lambda \ast N_{\text{load}} \]

\[ T_{\text{op}}^\psi = \kappa \ast |S^\psi| \]

The \( \lambda \) and \( \kappa \) are two constants modelling the impacts of respectively load and arithmetic operations on the physical target architecture.

We are not interesting in investigating \( \lambda \) and \( \kappa \) values, our analysis focus in discover some trends in the performance, leaving the two constant unspecified.

Focusing on Jacobi and Laplace we can define, parametrically with respect to the level of fusion \( i \), the mean time to compute one element as:

\[
T_{\text{el}}^{\text{Jacobi}}(i) = \lambda \ast \left[ 2i + 1 \right] + \kappa \ast \left[ (i + 1)^2 \right]
\]

\[
T_{\text{el}}^{\text{Laplace}}(i) = \lambda \ast \left[ 2i + 1 \right] + \kappa \ast \left[ (i + 1)^2 + i^2 \right]
\]

We are interested to study the improvement associated to SF–transformations, therefore we analyze the time gain factor of the two computations. We start to analyze before the two components, \( T_{\text{op}} \) and \( T_{\text{mem}} \), separately. The advantage of this strategy is that the \( \lambda \) and \( \kappa \) variables disappear in the time gain formula.

We recall that we have defined the time gain as the time of a reference computation divided by the time of a equivalent computation that we want to study. A time gain higher than one imply a gain in performance while in the other case a performance lost.

In this analyses we consider as reference time the one required to compute one step of the original stencil \( (T^\psi) \). We want to compare the computation of \( \psi \) with the ones of the stencils \( SF^i[\psi] \) \((T^{SF^i[\psi]}(i))\). Because in one step \( SF^i[\psi] \) computes \( i \) steps of the original stencil we normalize the time dividing it by \( i \), therefore the time gain formula is given by:

\[
G^\psi_{\text{mem}}(i) = \frac{i \ast T^\psi(1)}{T^{SF^i[\psi]}(i)}
\]

Coming back to the separated analysis of the memory and computation impacts we can assert the following points.

I The time gain associated to the only load operations are defined as:

\[
G^{\text{Jacobi}}_{\text{mem}}(i) = \frac{i \ast T_{\text{mem}}^{\text{Jacobi}}(1)}{T_{\text{mem}}^{\text{Jacobi}}(i)} = \frac{3 \ast i}{2i + 1}
\]
5.3. \( SF \) AND SEQUENTIAL COMPUTATIONS

\[
G_{\text{Laplace}}^i = \frac{i \cdot T_{\text{Laplace}}^i (1)}{T_{\text{mem}}^i (i)} = \frac{3 \cdot i}{2i + 1}
\]

Because, as it was highlighted before, the number of loads is equivalent to the number of rows on which the stencil shapes are spread. Both \( \text{Jacobi} \) and \( \text{Laplace} \) feature the same time gain on load operations.

With the increasing of the level of step fusion, the time gain (which is plotted in Figures 5.13(a) for the \( \text{Jacobi} \) case and in Figure 5.13(b) for the \( \text{Laplace} \) one. The curves are labeled with the name \( \text{memory} \)) increases as well but present an horizontal asymptote: it never overpass the value \( 3/2 \) as demonstrate by the limit:

\[
\lim_{i \to \infty} G_{\text{mem}}^{\text{Jacobi}} (i) = \lim_{i \to \infty} G_{\text{mem}}^{\text{Laplace}} (i) = \frac{3}{2}
\]

II From the other side, the time gain related to the arithmetic operations are defined as:

\[
G_{\text{op}}^{\text{Jacobi}} (i) = \frac{i \cdot T_{\text{op}}^{\text{Jacobi}} (1)}{T_{\text{op}}^{\text{Jacobi}} (i)} = \frac{4 \cdot i}{(i + 1)^2}
\]

\[
G_{\text{op}}^{\text{Laplace}} (i) = \frac{i \cdot T_{\text{op}}^{\text{Laplace}} (1)}{T_{\text{op}}^{\text{Laplace}} (i)} = \frac{5 \cdot i}{(i + 1)^2 + i^2}
\]

Differently from the other case, the two components are different for \( \text{Jacobi} \) and \( \text{Laplace} \) stencils. Their curves, labeled as \( \text{operations} \) are reported in Figures 5.13(a) and Figure 5.13(b).

As it was previously analyzed, with the increasing of the level of step fusion the computation load is heavier and therefore the time gain is always less than one. More precisely it is monotonically going to zero as \( O(1/i) \) as demonstrated by the following limits.

\[
\lim_{i \to \infty} G_{\text{op}}^{\text{Jacobi}} (i) = \lim_{i \to \infty} 4 \cdot \frac{1}{i}
\]

\[
\lim_{i \to \infty} G_{\text{op}}^{\text{Laplace}} (i) = \lim_{i \to \infty} 5 \cdot \frac{1}{2i}
\]

Without investigating on \( \lambda \) and \( \kappa \) specific values for a target architecture, we now focus on the asymptotic trend of \( G_{\psi} \):

\[
\lim_{i \to \infty} G_{\psi}^{\text{Jacobi}} = \lim_{i \to \infty} \frac{T_{\psi}^{\text{Jacobi}} (1) \cdot i}{T_{\psi}^{\text{Jacobi}} (i)} = \lim_{i \to \infty} \frac{3 \lambda + 4 \kappa}{\kappa i} = o \left( \frac{1}{i} \right)
\]
\[
\lim_{i \to \infty} G^\text{Laplace}_\psi = \lim_{i \to \infty} \frac{T^\text{Laplace}_\psi(1) \ast i}{T^\text{Laplace}_\psi(i)} = \lim_{i \to \infty} \frac{3\lambda + 5\kappa}{(\kappa + 1)i} = o\left(\frac{1}{i}\right)
\]

From the limit definition, we can conclude that, independently from the value of \(\lambda\) or \(\kappa\), there is a level \(\tilde{i}\) of step fusion after which there is no performance improvement. Both \(\lambda\) and \(\kappa\) impact on the value of \(\tilde{i}\).

We have therefore two possible scenarios.

I \(\tilde{i}\) can be equal to one, which means there is no performance improvement by exploiting \(\mathcal{SF}\) for sequential kernels.

II \(\tilde{i}\) can be bigger than one, that is \(\mathcal{SF}\) can be used to lower the computation time.

To visualize the trend of \(G_\psi\) we reported in Figure 5.13(a) and Figure 5.13(b) the curves associated to the case \(\kappa = \lambda = 1\).

It is important to notice that in our analysis we did not consider any limits to the number of registers of the architecture.

5.3.4 Taking into account Cache Memory Hierarchy

The temporal locality we analyzed in the previous Section features an important impact also on caches memory hierarchy. The principle is the same one that we analyzed for the register temporal reuse, but the explanation is slightly more complex, this is why we started the analysis of temporal locality from the registers prospective.

In this Section we do not aim to an in depth analytic analysis of the temporal locality impact on caches, as we did previously in the case of registers reuse. We wan just to simply give a prove that the temporal locality introduced by higher level of step fusion can influence also the performance on caches.

We target a simple model with only one level of caching between the memory and registers. We suppose each cache line to be of length \(\kappa\) and for the sake of explanation we do not consider any bounds about the number of lines. Indeed, we consider that the number of lines is sufficient to exploit completely the benefit of the temporal locality.

We analyze first the \(Jacobi\) stencil effects on caching and than we compare them with the ones of \(\mathcal{FS}^2[Jacobi]\). More precisely we compare one step of \(\mathcal{FS}^2[Jacobi]\) with two steps of \(Jacobi\).

In our analysis we take as reference the sequential naive implementation; the one that features two matrix, one for the input data and one for the output one. We suppose that the matrices have a number of rows equal to the variable \(\lambda\) and that each row is equivalent in size to a number of \(\eta\) cache lines.

On average, an element to be computed according to the Jacobi stencil requires that three cache lines has been previously filled up with the right portions of the
input matrix. Hence, to compute the entire matrix one time we can say that the number of cache lines that are read is approximately equal to:

\[ 3 \times \lambda \times \eta \]

A second iteration of Jacobi would require the same number of cache line read operations. In conclusion in order to complete the computation of two steps, the total number of cache lines that are read is approximately equal to:

\[ 6 \times \lambda \times \eta \]

If we consider the stencil \( \mathcal{SF}^2[Jacobi] \) in one step, which as we recall is computationally equivalent to two computation steps of \( Jacobi \), the total number of cache lines that are read is approximately equal to:

\[ 5 \times \lambda \times \eta \]

Therefore in order to update the working domain according to the computation of two iterations of the Jacobi stencil, the implementation which exploit the \( \mathcal{SF} - transformations \) performs approximately \( \lambda \times \eta \) cache lines read operations less that the naive sequential implementation.

### 5.3.5 Experimental Results

We have experimentally evaluated the time gain of different levels of step fusion, considering a wide range of cases. We examine the three example stencils \( Jacobi \), \( Laplace \) and \( Nine \) and four target architectures featuring different processor frequency and cache sizes.

The test were run targeting a size of the working domain that could not completely fit the cache memory levels. For the two dimension case we target matrices of 72 MB. The achieved results are presented in charts of Figures 5.14(a), 5.15(a), 5.16(a), 5.17(a).

For the \( Nine \) stencil, which features a higher increase of number of operations associated to the rise of the step fusion level, no benefits has been detected by the use of \( \mathcal{SF} - transformations \). This means that the impact of the arithmetic overhead is stronger than reuse effects.

Different are the results for the other two stencils. In all the targeted architectures, \( SF^2[Jacobi] \) and \( SF^2[Laplace] \) provide better performance with respect to the original stencils. In some cases the time gain is close to two: 2.1 for \( Jacobi \) and 1.8 for \( Laplace \). Therefore the sequential implementation of the transformed stencil is closed to be up to time faster than the original sequential version.

Some test were also performed exploiting the three dimension extension of Jacobi, Laplace and Nine. The charts in Figure Figures 5.14(b), 5.15(b), 5.17(b). For the three dimension cases we selected a working domain of 216MB.
Figure 5.14: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB
Figure 5.15: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB
Figure 5.16: Intel(R) Xeon(R) CPU 5150 @ 2.66GHz cache size 4096 KB
Figure 5.17: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB
The results are slightly different from the ones of the two dimension cases. The registered time gains are reduced with respect the previous ones. In a test configuration, also the three dimension Laplace features, for level of step fusion grater than one, a time gain that is less than one; which means performance lost. This behaviour is given by the increasing of the impact of the computational component with respect to the other one.

As a support for the previous assertion, we report in Table 5.3 the number of shape elements featured by the stencil in a three dimension space and in Figure 5.18 the chart of the Element Increasing Factor.

<table>
<thead>
<tr>
<th></th>
<th>Jacobi</th>
<th>Laplace</th>
<th>Nine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SF^1$</td>
<td>6</td>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td>$SF^2$</td>
<td>19</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>$SF^3$</td>
<td>44</td>
<td>63</td>
<td>343</td>
</tr>
<tr>
<td>$SF^4$</td>
<td>85</td>
<td>129</td>
<td>729</td>
</tr>
</tbody>
</table>

Table 5.3: Number of elements of the stencils resulted by the applications of different $SF-$transformations to the three dimension extension of Jacobi, Nine and Laplace

5.3.6 Conclusions

The previous analysis, made on the two dimension cases, are confirmed by the experimental results: all the measured speed up for step fusion transformation above second level are monotonically decreasing. We conclude that sequential code optimized with $SF-$transformations and a low level of fusion can provide in some cases an outstanding performance improvement. The improvement are reduced when targeting three dimension spaces.
Figure 5.18: Element Increasing Factor for Jacobi, Laplace and Nine extended in a three dimension space.
Chapter 6

Space Overlapping Transformations

Abstract

In the previous Chapters we focused on techniques to optimize both communication and computational overheads, indeed for the two problems we introduce respectively \( Q \)-transformations and \( SF \)-transformations. In this Chapter, we concentrate on the memory requirements for the implementation at the concurrent level of a \( HUA \) stencil.

Implementations of \( HUA \) stencil which approximately halve the memory requirement to represent the working domain are well known, but they suffer of the drawback of increasing the computation overhead. Indeed, those techniques require a copy operation for each element of the working domain.

In this Chapter, we present and formally prove the existence of a specific \( Q \)-transformations called \( QM \)-transformations. The stencil resulting from this new transformations can be associated to an in-place implementation that halves the memory requirements without introducing other computational overheads.
Contents

<table>
<thead>
<tr>
<th>6.1</th>
<th>The Implementation of the Working Domain</th>
<th>179</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.1</td>
<td>Naive Implementation</td>
<td>179</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Support Buffer Implementation</td>
<td>180</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Space Overlapping Implementation</td>
<td>182</td>
</tr>
<tr>
<td>6.2</td>
<td>$Q_M$–transformations</td>
<td>187</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Positive $Q_M$–transformation</td>
<td>187</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Negative $Q_M$–transformation</td>
<td>191</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Performance Tests</td>
<td>193</td>
</tr>
</tbody>
</table>
6.1 The Implementation of the Working Domain

We have seen in the previous Chapters the importance of the working domain component in both the structured and the HUA models.

At the functional level, the working domain models statically, thanks to the temporal dimension associate to the evaluation map, the complete evolution of the stencil computation. Indeed, the evaluation of an element is possible in any computation time because it results in a parametric static formula.

At the concurrent level, the aim of the data structures implementing the working domain is to store, at the end last step of the application, the final values of all the spatial structure elements.

During the run time computation, the temporal dimension, that is featured by the working domain at the functional level, is lost and only the values at the current instant are stored. We would like to highlight that the implementation of the working domain includes both the definition of the data structures for the values and the strategies to update the values from the beginning to the end of one step computation.

For the sake of exposition simplicity, all the considerations that we are going to make in the rest of the Chapter focus on the implementation of a sequential program. Nevertheless, all the results can be reused also to implement a partition of the working domain.

In the following we first present and analyze two standard implementations of the working domain. Then we pass to the introduction of a new implementation strategy that, as it is going to be clear after a deep analysis, exploits $Q-$transformations. The changes that the transformations make on the stencil shape are such that it is possible to minimize the memory requirements without introducing a copy overheads.

To present the three different implementations we refer once again to the mono dimension Jacobi stencil.

6.1.1 Naive Implementation

The easiest way to implement a working domain is to exploit two different data structures. One stores the working domain values at time $i$ and the other is going to store the values at time $i + 1$. We report in Figure 6.1 the pseudo code of the algorithm. This is the same strategy we used in all the previous examples. This method does not introduce any constraint on the pattern visit of the data structures.

In order to analyze the presented strategy from the point of view of the memory requirements, let we consider $|M_{Jacobi}|$ the number of elements in the spatial structure and $d(D_{Jacobi})$ the function that returns the size of a value of the computational domain. Finally, modelling the memory requirements with the parameter $MR$ we can claim the following formula.
double $J_{in[100]}, J_{out[100]}$;
load_working_domain_values($J_{in}$);
for ($i_{step} = 0; i_{step} < 4; i_{step} + +$) {
    forall ($x, \in J_{in}$) {
        $J_{out[x]} = (J_{in[x + 1]} + J_{in[x - 1]})/2$;
    }
    swap($J_{in}, J_{out}$);
}
return_working_domain($J_{out}$);

Figure 6.1: Description in pseudo code of a naive implementation of the Jacobi stencil on a toroidal space.

$MR_{naive} = 2 \times |M_{Jacobi}| \times d(D_{Jacobi})$

The memory allocated for the implementation of the working domain is equal to two times the one required to store all values of the spatial structure elements.

6.1.2 Support Buffer Implementation

During a step computation $i$, when an element value at time $i$ is not any more necessary for the computation of some value at time $i + 1$, its memory can be reused to store values at time $i + 1$. In other words, we are free to target in-place computation strategies, which in a step reuse the input data structures to store output values.

We can consider an in-place computation if we target the following points.

I A specific visit for the update of the working domain elements is exploited.

II Some support buffers are introduced in order to resolve the data dependencies which are the result of the introduction of in-place computations.

We report in Figure 6.2 the pseudo code of the algorithm that exploits a standard strategy for in-place computations.

The base concept is to compute all the updates in-place and to relay on support buffers in order to save necessary values that would be lost. We call the presented method for implementing the working domain the buffered strategy.

We want to highlight three important points of the buffered strategy.

I The dimension of the support buffers depends directly from the geometry of the stencil shape. The more the shape area is extended the more higher is the buffer space.
double J[100];
double buffer_support_head;
double buffer_support_present;
double buffer_support_future;

load_working_domain_values(J);

for(i_step = 0; i_step < 4; i_step ++){
    copy(buffer_support_head, J[0];)
    copy(buffer_support_present, J[-1];)

    for(x = 0; x < 999; x ++){
        copy(buffer_support_future, J[x];
        J[x] = (J[x + 1] + buffer_support_present) / 2;
        swap(buffer_support_present,
             buffer_support_future);
    }
    J[-1] = (buffer_support_head +
             buffer_support_present) / 2;
}
return_working_domain(J);

Figure 6.2: Description in pseudo code of a buffered implementation of the Jacobi stencil on a toroidal space.
II All the elements of the matrix are copied. The strategy therefore imply a complete copy of all the elements of the working domain.

III The copy associated to buffer_support_head is mandatory to break the toroidal feature. In the case of stencil defined on non toroidal spaces this copy is not necessary.

Analyzing the memory requirements of the algorithm we can extract the following formula.

\[
MR_{\text{buffered}} = (|M_{\text{Jacobi}}| + \phi_{\text{buffer}}(S_{\text{Jacobi}})) \times d(D_{\text{Jacobi}})
\]

The function \( \phi_{\text{buffer}} \) returns, with respect to a target stencil shape, the size of the required buffers.

Because one of our working hypothesis is that the size of a working domain is considerably wider than shape one, we can consider the buffer size as negligible in terms of memory requirement.

We conclude that the buffered method, at the price of introducing a computational overhead per step associated to the copy of all the values, halves the memory requirements with respect to the naive implementation. We can therefore claim the following formula

\[
MR_{\text{buffered}} \approx |M_{\text{Jacobi}}|
\]

### 6.1.3 Space Overlapping Implementation

In this Section, we present a specification of \( Q - \) transformations that, at the functional dependency level, transform the stencil in a relaxed equivalent one. The resulting stencil can be implemented with the same requirements of memory of the buffered implementation but avoiding a copy overhead of the whole working domain.

For the sake of clarity, we decide to firstly present and analyze the algorithm for the Jacobi case and then to highlight from its structure the most important aspects that can lead us to the definition of what we will call \( QM - \) transformations.

Figure 6.3 reports the pseudo code of the new strategy for implementing the working domain. The program is more simple than it could appear at a first look. Indeed, if we respectively replace \( V_{\text{odd}}^J \) and \( V_{\text{even}}^J \) with \( J_{\text{in}} \) and \( J_{\text{out}} \) the program, except for the pattern of visit, is similar to the naive one. In each step, values are read only from one vector and written on the other.

By the way, \( V_{\text{odd}}^J \) and \( V_{\text{even}}^J \) are not real vectors. We define them as virtual vectors, i.e. vectors that are mapped in other vectors. Virtual vectors can be seen as a index renaming mechanisms.

Both the \( V_{\text{odd}}^J \) and \( V_{\text{even}}^J \) are mapped into the same vector \( J \). The first virtual vector is mapped on index interval \([0, 99]\), while the second one on \([1, 100]\). A graphic representation of the overlapping is reported in Figure 6.4(a).

Because in the mapping, the two virtual vectors are partially overlapped, we are in a configuration where loop-carried dependencies are present inside a single
double \( J[100 + 1]; \)
VIRTUAL \( V_{even}^{J}[100] \rightarrow J[0,99] \)
VIRTUAL \( V_{odd}^{J}[100] \rightarrow J[1,100] \)
double buffer_support;

load_working_domain_values(\( J \));

\[
\begin{align*}
&\text{for}(i_{step} = 0; i_{step} < 4; i_{step} += )
\quad \text{if}(i \% 2 == 0) \{ \\
&\quad \quad \text{copy}(buffer\_support, V_{odd}^{J}[−1];) \\
&\quad \quad \text{for}(x = 99; x > 1; x −−) \\
&\quad \quad \quad V_{even}^{J}[x] = \left(V_{odd}^{J}[x + 1] + V_{odd}^{J}[x − 1]\right)/2; \\
&\quad \quad \} \\
&\quad \quad \text{V}_{even}^{J}[x] = \left(J[x + 1] + buffer\_support\right)/2; \\
&\quad \text{if}(i \% 2! = 0) \{ \\
&\quad \quad \text{copy}(buffer\_support, V_{odd}^{J}[0];) \\
&\quad \quad \text{for}(x = 0; x < 99; x + +) \\
&\quad \quad \quad V_{odd}^{J}[x] = \left(V_{even}^{J}[x + 1] + V_{even}^{J}[x − 1]\right)/2; \\
&\quad \quad \} \\
&\quad \quad \text{V}_{odd}^{J}[x] = \left(J[x + 1] + buffer\_support\right)/2; \\
&\quad \} \\
&\text{return\_working\_domain(V_{odd});}
\end{align*}
\]

\textbf{Figure 6.3:} Description in pseudo code of a implementation with space overlapping of the Jacobi stencil defined over a toroidal space.
Figure 6.4: Mapping between virtual and real vectors for the overlapping implementation
6.1. THE IMPLEMENTATION OF THE WORKING DOMAIN

Figure 6.5: Jacobi, $Q^+[Jacobi], Q^-[Jacobi]$ shapes.

step. It is extremely important to exploit a visit of the two virtual vector that can guarantee the program semantics.

The two visits that are implemented in the algorithm, one for odd steps and one for even, avoid any conflicts between read and write operations. Indeed, it easy to see that all the element storing values at the time $i$ are exploited before they are rewritten with the new value at time $i + 1$.

Figure 6.4(b) gives a graphical representation of the previous assertion for the odd steps. Following the sequence of operation forced by the visit pattern, we can register what follows.

I The value of the last element of the vector $V^J_{odd}$, which stores element values at time $i$, is exploited to compute the value of the next to last element of $V^J_{even}$.

II The calculated value is stored in the next to last position of $V^J_{even}$. Because of the overlapping of the virtual vectors, the store operation changes the value also of the last element of $V^J_{odd}$.

III No other accesses for read operations are performed on the last element $V^J_{odd}$.

What we claimed for the read and write operations on the last element of the $V^J_{odd}$ vector on odd steps can be extended to all the elements of the vector and symmetrically also to all the elements of $V^J_{even}$ in even steps.

Because we asserted that an element which stores a value at time $i$ is rewritten with a value at time $i + 1$ only when the previous value is no more necessary for the stencil computation, we have proved that the previous algorithm respect the semantic of the computation.

From the prospective of the memory requirement, we can model the obtained results for the Jacobi example with following formula.

$$MR_{QM} = (|M_{ext}^{Jacobi}| + \phi_{overlapping}(S^{Jacobi})) * d(D_{Jacobi})$$

The function $\phi_{overlapping}$ returns with respect to a target stencil shape the size of the buffer required to break the toroidal constraint. As for the previous implementation the buffer size is negligible with the total size of the working domain. The component $M_{ext}^{Jacobi}$ represents the extended working domain.
In terms of memory requirements the presented method and the previous one are equivalent. The extended dimension of the working domain compensates the reduction of the buffer size. We can finally claim that this last implementation, without introducing copies of the values of the entire spatial structure, halves the memory requirements. Formally we can claim the following formula.

\[ MR_{Q,M} \cong |M_{\text{ext}}^{\text{Jacobi}}| \ast d(D_{\text{Jacobi}}) \]

Let us focus once again on the two virtual vectors and on their relation with both the stencil shape and the application point. According to the relative indexes of each of the two virtual vectors, the shape and the application point are the ones defined by the original Jacobi stencil as reported in Figure 6.5(a). Nevertheless if we take into consideration the absolute reference system of the non virtual vector \( J \), the shapes of odd and even steps are the result of a \( Q - \text{transformations} \) as shown in Figures 6.5(b) and 6.5(c). The implementation of the working domain that halves the memory requirements can be modelled as a \( Q - \text{transformation} \).
6.2 QM-transformations

In this Section we give a formal definition of QM-transformations: specific Q-transformations that allow the implementation of a program at the concurrent level that is optimized in order to reduce the memory requirements. Indeed, we prove that the transformed stencil along with a specific pattern of visit can be implemented in a program at the concurrent level that halves the memory requirements without introducing copy overhead.

As in the case of Q-transformations for the reduction of communications, QM-transformations are based on two symmetric transformations: one that is going to be associated to odd steps and one to even steps. We present and prove positive QM-transformations and we demonstrate that with the right pattern of access it is possible to implement odd step featuring in-place computations.

6.2.1 Positive QM-transformation

Definition 6.2.1 (Positive QM-transformation). Let $\psi$ be a HUA stencil. A positive QM-transformation is a Q-transformations that transforms $\psi$ in the HUA stencil $QM^+[\psi]$ which is equivalent to the original one, except for the step model that is defined as follows:

$$\forall e \in M \xrightarrow{QM^+[\psi]} (F^\psi, S^{Q^+[\psi]})$$

$$q^+_m = <q_1^+, 0, \ldots, 0>$$

$$q^+_{\text{limit}} = -\min \{\beta * \epsilon_1 \forall \beta \in R^\psi\}$$

$$q^+_1 = \begin{cases} q^+_{\text{limit}} \text{ if } (\beta * \epsilon_1 = -q^+_{\text{limit}}) \Rightarrow (\forall j > 1 \beta * \epsilon_j = 0) \\ q^+_{\text{limit}} + 1 \text{ in the other cases} \end{cases}$$

$$R^{QM^+[\psi]} = R^\psi + q^+_m$$

$$S^{QM^+[\psi]} = e + R^{Q^+}$$

$$M^{i+1}_{QM^+[\psi]}[e] = F_i(M^{i}_{QM^+[\psi]}[e + \gamma_1], \ldots, M^{i}_{QM^+[\psi]}[e + \gamma_n])$$  \hspace{1cm} (6.1)$$

where $\epsilon = \{\epsilon_1, \ldots, \epsilon_{\text{dim}}\}$ is the set of the vectors in the natural basis of $N^{\text{dim}}$. Moreover, $\beta * \epsilon_i$ is the scalar product which returns the component of the vector $\beta \epsilon_i$ along the main space direction expressed by the vector $\epsilon_i$.

Informally, the transformation moves the application point only on the first dimension. With respect to the resulting relative shape two situation can be possible:

I the first coordinate of all the shape element is strict positive
II at most the first coordinate of only one element is equal. Moreover the element is the result of a translation of the application point only along the first axe.

We reported the transformed shape of a set of stencil in Figure 6.6. The darker element in shape represent the application point or equivalently the center of the reference system used to describe the relative shape.

Both Laplace and Jacobi fall in the second of the described configurations; there is at most one element whose first coordinate is equal to zero. In the other case instead there is the nine point stencil; indeed the all the element feature the first coordinate that is strictly positive. This difference is going to be extremely important in the proof of the following Theorem.

**Theorem 6.2.1 (In-Place Computation with $QM$-transformations).** Let $\psi$ be a generic HUA stencil. One step of $QM^+[\psi]$ can be implemented at the concurrent level exploiting an in-place computation without copy overhead.

*Proof.* Let the spatial structure of $\psi$ defined as follows:

$$M_\psi = \mathbb{Z}^{m_1} \times \ldots \times \mathbb{Z}^{m_n}$$

For the implementation of the working domain at the concurrent level we consider the following matrix:

$$J[m_1 + q_1^+][m_2] \ldots [m_n]$$

We suppose that, at the beginning of the step, the working domain values are mapped in the first $m_1 - 1$ rows of the $J$ matrix.

The first operation we consider is a copy that, exploiting some buffers, breaks the toroidal functional dependencies. In other words we cut, if any, the functional dependencies between elements on the top of the matrix and those on the bottom. This copy operation is mandatory to target an in-place computation and can not be avoided.

It is worth to mention that the number of the copied elements depends on the stencil volume and is negligible with respect to the copy of the entire working domain.

To prove the Theorem, we claim that the visit pattern schematized in Figure 6.8 respects the semantic of the $QM^+[\psi]$ stencil computation.

Before proceeding, it is important to notice that the reference system of C-like matrices (the one used in the sequential aspects the concurrent language) features the first dimension with an opposite orientation with respect to the shape reference system. Consider the property of the relative shape of featuring elements with a first component that is strictly non negative. In terms of the C-like matrices, the properties asserts that the first index of each element is smaller that the one of the application point.

Because we are targeting an in-place computation, the matrix $J$ is going to store both input and output values for the computation of a step. Hence, the computation
Figure 6.6: Graphical representations of shapes and application points of a set of stencils transformed with $QM$–transformations.
inside the forall loop obviously features loop-carried dependencies with respect to the outermost loop controlled by the variable \( x_1 \).

We have to demonstrate that the order of read and write operations, that is forced by the specific pattern of visit, respects the semantic of the program. For the sake of simplicity we first consider the case \( q_1^+ = q_{\text{limit}} + 1 \).

The selected visit pattern forces a decreasing order of visit for the first component. In other words given two matrix indexes \((\alpha_1, \ldots, \alpha_n)\) and \((\beta_1, \ldots, \beta_n)\), if \( \alpha_1 > \beta_1 \) then the first index is visited before the second one.

Recalling that the pattern visit selects the application points for the computation, we analyze the relation between the shape and the pattern visit.

By definition, \( QM^+[\psi] \), in the case of \( q_1^+ = q_{\text{limit}} + 1 \), features a relative shape whose first component is strictly positive. In terms of the C-like reference system, the application point features a first component that is higher than the one of the shape elements. Because only the shape elements are targeted for read operations, a matrix element is visited (i.e. considered as application point) only when there are no other elements that requires its value.

Resuming, all the functional dependencies are oriented on indexed featuring a strict lower value in the first component. A bottom-up visit along the first dimension guarantees the program semantic. This proves the Theorem for the case \( q_1^+ = q_{\text{limit}} + 1 \).

The other case, \( q_1^+ = q_{\text{limit}} \), is just an optimization for those stencils that, as Jacobi and Laplace, feature only one element of the shape with the lower first component. The element has moreover the characteristics of laying on the axe of the first dimension.

In this cases, it is possible to avoid the addition of one unit because no data dependency can arise between elements of \( QM^+[\psi] \) that feature the same value in the first component.

We can therefore claim also the following Theorem.

**Theorem 6.2.2** (Memory Requirement with Positive QM–transformations). Let \( \psi \) be a generic HUA stencil. One step of \( QM^+[\psi] \) can be implemented at the
concurrent level exploiting a memory size that is equal to
\[ |\mathcal{M}_{Q\mathcal{M}^+}[\psi]| \ast d(D_{Jacobi}) \]
plus some memory for buffer support that according to the Working Hypothesis 3.2 can be considered as negligible.

Proof. The proof come directly from the implementation of \( Q\mathcal{M}^+[\psi] \) that is proposed in the proof of the previous Theorem. \( \square \)

### 6.2.2 Negative \( Q\mathcal{M} \)-transformation

We have seen that positive \( Q\mathcal{M} \)-transformations can be used to implement one single step exploiting an in-place computation and without introducing a copy overhead of the complete working domain.

Nevertheless, positive \( Q\mathcal{M} \)-transformations can not be applied for two consecutive steps. Indeed we recall that in the proof of the Theorem 6.2.3 we suppose that the working domain elements are mapped at the beginning of the step in the upper part of the matrix. After one step the domain values are stored in the lower part of the matrix.

We need some mechanisms to move the elements back to the top of the matrix. Or better, we can exploit the same technique that we used with the positive \( Q \)-transformations. We therefore define an antithetic transformation with respect to the positive one.

**Definition 6.2.2 (Negative \( Q\mathcal{M} \)-transformation).** Let \( \psi \) be a \( HU\mathcal{A} \) stencil. A positive \( Q\mathcal{M} \)-transformation is a \( Q \)-transformations that transforms \( \psi \) in the \( HU\mathcal{A} \) stencil \( Q\mathcal{M}^+[\psi] \) which is equivalent to the original one, except for the step model that is defined as follows:

\[
\begin{align*}
\forall \mathcal{e} \in \mathcal{M} & \quad \mathcal{M}_{Q\mathcal{M}^+}[\psi] \xmapsto{} (\mathcal{F}_\psi, \mathcal{S}_{Q^+}[\psi]) \\
q_m & = < q^-_1, 0 \ldots, 0 > \\
q^-_{\text{limit}} & = -\max \{ \beta_\alpha \ast \epsilon_1 \forall \beta_\alpha \in \mathcal{R} \} \\
q^-_1 & = \begin{cases} 
q^-_{\text{limit}} \text{if } (\beta_\alpha \ast \epsilon_1 = -q^-_{\text{limit}}) & (\forall j > 1 \beta_\alpha \ast \epsilon_j = 0) \\
q^-_{\text{limit}} - 1 \text{ in the other cases}
\end{cases} \\
\mathcal{R}_{Q\mathcal{M}^-}[\psi] & = \mathcal{R}^\psi + q^-_m \\
\mathcal{S}_{Q\mathcal{M}^-}[\psi] & = \mathcal{e} + \mathcal{R}^- \\
\mathcal{M}^{i+1}_{Q\mathcal{M}^+}[\psi][\mathcal{e}] & = \mathcal{F}_i(\mathcal{M}^i_{Q\mathcal{M}^-}[\psi][\mathcal{e} + \gamma_1, \ldots, \mathcal{M}^i_{Q\mathcal{M}^-}[\psi][\mathcal{e} + \gamma_n])
\end{align*}
\]
where $\epsilon = \{\epsilon_1, \ldots, \epsilon_{\dim}\}$ is the set of the vectors in the natural basis of $\mathbb{N}^{\dim}$. Moreover, $\beta_\alpha \cdot \epsilon_i$ is the scalar product which returns the component of the vector $\beta_\alpha$ along the main space direction expressed by the vector $\epsilon_i$.

As for the positive $QM -$ transformations we can claim the following Theorem for the negative ones.

**Theorem 6.2.3 (In-Place Computation with Negative $QM -$ transformations).** Let $\psi$ be a generic HUA stencil. One step of $QM^-[\psi]$ can be implemented at the concurrent level exploiting an in-place computation without copy overhead.

**Proof.** Let the spatial structure of $\psi$ defined as follows:

$$M_\psi = \prod_{i=1}^{\dim} \mathbb{Z}_{m_i}$$

For the implementation of the working domain at the concurrent level we consider the following matrix:

$$J[m_1 - q^+_{1}] [m_2] \ldots [m_n]$$

We suppose that, at the beginning of the step, the working domain values are mapped in the last $m_1 - 1$ rows of the $J$ matrix.

The First operation we consider is a copy that, exploiting some buffers, breaks the toroidal functional dependencies. In other words we cut, if any, the functional dependencies of the elements on the top of the matrix with those on the bottom. This copy operation is mandatory to target a in-place computation and can not be avoided.

It is worth to mention that the number of the copied elements depends on the stencil volume and is negligible with respect to the copy of the entire working domain.

To prove the Theorem, we claim that the visit pattern schematized in Figure 6.8 respects the semantic of the $QM^+[\psi]$ stencil computation.

<table>
<thead>
<tr>
<th>for($x_1 = 0; x_1 &lt; m_1; x_1 + + )$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall((x_2, \ldots, x_n)</td>
<td>(x_1, x_2, \ldots, x_n) \in J)</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>compute $QM^-[\psi]$ with $(x_1, x_2, \ldots, x_n)$</td>
<td>4</td>
</tr>
<tr>
<td>as application point</td>
<td>5</td>
</tr>
<tr>
<td>}</td>
<td>6</td>
</tr>
</tbody>
</table>

**Figure 6.8:** Visit pattern forced by the in-place computation.

From this point on, the demonstration is completely symmetric with respect to the one that were given for positive $QM -$ transformations.
It is easy to see that at the end of the one step computation the values of the working domain are now stored in the upper part of the matrix, i.e. in the first $m_1$ rows.

We can therefore conclude with the following Theorem.

**Theorem 6.2.4 (Memory Requirement with Positive QM-transformation).**

Let $\psi$ be a generic HUA stencil. A generic number of steps $\psi$ can be implemented at the concurrent level exploiting an interleaving of positive and negative QM-transformations: on odd steps $\mathcal{QM}^+[\psi]$ and $\mathcal{QM}^-[\psi]$ in even ones.

The resulting implementation features a memory requirement equal to

$$|\mathcal{M}_{\mathcal{QM}^+}[\psi]| \times d(D_{Jacobi})$$

plus some memory for buffer support that according to the Working Hypothesis 3.2 can be considered as negligible. Finally the implementation does not imply the complete copy of all the working domain elements.

**Proof.** The proof comes directly from the union of all the other Theorems proof.

6.2.3 Performance Tests

We tested $\mathcal{QM}$-transformations for different stencils, architectures and also combining $\mathcal{SF}$-transformations. The targeted stencil are Jacobi, Laplace and Nine plus all their transformations with the step fusion method up to a level of four.

We exploited three different architectures whose characteristics are reported in the following list.

I Intel Pentium III CPU - M @ 800MHz cache size 512 KB

II Intel Pentium 4 CPU @ 2.00GHz cache size 512 KB

III Intel Xeon CPU E5420 @ 2.50GHz cache size 6144 KB

For each of the three stencil we plotted we reported a chart plotting the following functions:

$$G^{\mathcal{SF}[\psi]}(i) = \frac{i \times T^\psi}{T^{\mathcal{SF}[\psi]}}$$

$$G^{\mathcal{QM}[\mathcal{SF}[\psi]]}(i) = \frac{i \times T^\psi}{T^{\mathcal{QM}[\mathcal{SF}[\psi]]}}$$

The variable $T^\psi$ is the mean time to compute one time the entire working domain, exploiting the stencil $\psi$. For the test, we obviously considered the computation of a working domain whose size would fit completely into the main memory, but not into the cache levels.
The first function defines the time gain of the stencil $S\mathcal{F}[\psi]$ with respect to the naive implementation $S\mathcal{F}^1[\psi] = \psi$. This is the same parameter we presented in the chart of the previous Chapter to study $S\mathcal{F} - transformations$.

The second function defines the time gain of all the stencil $QM[S\mathcal{F}[\psi]]$ always with respect to the naive implementation $S\mathcal{F}^1[\psi] = \psi$. Therefore both the two time gains target the same reference implementation.

From all the charts reported in Figures 6.11, 6.12, 6.13 and 6.14 it is evident a well defined trend. In all the cases the exploitation of $QM - transformations$ improves the performance of the computation for low levels of step fusion. More precisely the best performance configuration, which in the charts is represented by the point with the higher time gain value, is always associated to a stencil that has been transformed with the $QM - transformations$.

Therefore the transformation both reduce the memory requirements and improve the performance. The performance benefit comes from the impact of the algorithm on the cache hierarchy; more precisely of the write operations.

On write allocate architectures, a write miss will necessitate the allocation of a cache line. Therefore before the execution can proceed, the contents of the line is read from main memory. In the case of a stencil computation, this is a waist of time because the entire line is going to be rewritten.

A possibility to avoid this situations, cache initialization and cache bypass instructions can be used. According to Datta et. al. [14] the use of this instruction can increase the performance of a 50%.

Without exploiting cache bypass instructions the $QM-transformations$ provide an implementation of Laplace and Jacobi that avoid the overhead of unnecessary cache misses. Indeed, the results are stored in cache lines that have been previously read.
Figure 6.9: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB
Figure 6.10: Mobile Intel(R) Pentium(R) III CPU - M @ 800MHz cache size 512 KB
Figure 6.11: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB
Figure 6.12: Intel(R) Pentium(R) 4 CPU 2.00GHz cache size 512 KB
Figure 6.13: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB
Figure 6.14: Intel(R) Xeon(R) CPU E5420 @ 2.50GHz cache size 6144 KB
Chapter 7
Conclusions

In the previous Chapters we presented a set of innovative transformations for stencil-based applications.

- **Q-transformations** provide optimizations on the communication overhead. With the presented transformations the number of communications required to implement a generic stencil, defined over a $n$ dimension space, can be reduced to $n$, with respect to $2 \times n$ which is the best result provided by solutions cited in literature.

  Experimental tests on clusters and multi-core architectures prove that **Q-transformations** provide better performance than other implementations, especially when targeting fine grain parallelization. In worst case tests, where communications with all neighbours are required, the reduction of the communication overhead can has been quantified with respect to a “naive” implementation in a value of time gain that is up to $4.5$.

- **QSF-transformations** are a formalization and a notable extension of those techniques which exploit replication of data to reduce communications. One notable useful side effect of studying **QSF-transformations** is the definition of politics for the memory hierarchy management in stencil computational kernel.

  The technique, which is based on a revisitation of loop fusion classic optimizations, provides a value of time gain up to $2.1$, in a sequential environment.

- **QM-transformations** result in optimizations targeting the reduction of memory constraints. This reduction is obtained without performance lost; the type of memory accesses defined by **QM-transformations** rather provide notable performance benefit.

  By experimental results, we prove that **QM-transformations** almost halve the memory constraints and in some cases provide a value of time gain up to $2.2$. 
All the presented transformations are the result of the definition of a new optimization theory based on a relaxed concept of computational equivalence.

Differently from the classic vision, we consider safe, or better relaxed safe, a transformation when the original program and the transformed one produce, for the same input data structures, the same output values which can however feature a different spatial organization in the output data structures.

The relaxed optimization theory along with a structured approach in studying stencil-based application give us the basis to the definition of a structured stencil model which is exploited to introduce and formally prove all the previous transformations.

Formally the relaxed optimization theory is an extension of the classical one. Indeed each transformation that is safe in the classical meaning can be obviously classified also as relaxed safe; the implication in the other way round is not valid.

For the previous assertions we claim that we can merge classical optimization techniques with the new presented ones in order to obtain another relaxed transformations. We consider this prospective an interesting and promising subject for our future works.
Appendix A

MammuT

Abstract

Structured parallel programming is a parallel software development methodology that aims at delivering programmability, portability and interoperability, along with scalability and performance. To achieve those goals it is important to define both a suitable set of high level parallel constructs and a communication language whose mechanisms provide both very high performance and low overhead, for efficient implementation of parallel construct runtime, and a clear cost model that allows for parallel construct composition optimization.

We describe our experience in defining abstract and concrete communication protocols optimized for structured parallel programming on single chip multi-core architectures. We implement and test our mechanisms on the IBM Cell BE chip multi-core. We detail a comprehensive cost model of the communications, that is a requirement for supporting automatic optimization in a structured parallel framework, and we report the achieved performance. Our implementation reaches best possible bandwidth and latency figures on this architecture: measured performance numbers are extremely close to the actual hardware limits.

The Chapter is structured as follow. Section A.1 describes our structured parallel programming framework.

Section A.2 introduces the semantic and the syntax of the communication language $\mathcal{LC}$; the section also briefly compares some important characteristics of $\mathcal{LC}$ and $\textit{MPI}$.

Section A.3 and A.4 describes an abstract protocol and platform independent optimization for $\mathcal{LC}$ channels.

Section A.5 details the actual implementation of the communication support on the Cell architecture.

Finally Section A.6 reports achieved performance, comparing it with the MPI for Cell implementation and with hardware $\textit{DMA}$ transfer mechanisms. This section characterizes also the communication cost model.
Contents

A.1 Structured Parallel Programming 206
A.2 \( \mathcal{LC} \) Language Semantics and Syntax 209
   A.2.1 \( \mathcal{LC} \) Channel API 209
   A.2.2 \( MPI \) and \( \mathcal{LC} \) 210
A.3 \( \mathcal{LC} \) Channel Abstract Protocol 212
A.4 Channel Abstract Optimization 214
   A.4.1 Static Refilling: the \textit{w} protocol 214
   A.4.2 The \textit{K} _{\text{plus_one}} Optimization 216
A.5 Concrete Implementation on the Cell 218
   A.5.1 The Cell Architecture 218
   A.5.2 Channel Implementation on Cell 218
   A.5.3 Signal-based Implementation 222
   A.5.4 DMA1 Implementation 222
   A.5.5 DMA2 Implementation 223
A.6 The Cost Model 225
A.7 Conclusion and Feature Works 227
Nowadays every CPU design is based on integrating multiple homogeneous or heterogeneous cores inside a single chip. This technology shift, from accelerating a single core performance to integrating multiple simpler cores, originates from the inability of scaling application performance using only hardware improvements and instruction level parallelism. The current chip production technology suffers from severe physical limitations, which can be summarized as power, frequency and memory walls: we have an hard limit on usable power in integrated circuits, diminishing returns from deeper pipelines that allow faster clock speed and the access speed of DRAM memory is severely limited.

CPU designers attack these power, frequency and memory issues by requiring programmers to explicitly express thread level parallelism and exploit this parallel behavior on multiple (more or less independent) cores. Software developers are forced to use techniques similar to those developed for traditional high performance computing (HPC) architectures. What changes from the past is the packaging: these new architectures are now integrated inside a single chip. In this scenario, as it has always been in HPC field, the burden of achieving software performance shifts from hardware designers to software developers: in order to gain any advantage from modern CMP every software component must be coded with explicit parallelism.

New software development methodology are clearly fundamental to be able to provide desirable characteristics, such as programmability, portability and interoperability, along with performance and scalability. Structured parallel programming is an approach aiming to deliver those goals by “restricting” the parallel program structure to a set of parametric and configurable parallel constructs. This approach drives programmability by defining a methodology that guide programmers in designing the parallel application, while hiding low level architectural aspects and allowing application portability on different architectures. On the other side, structured parallel programming allows for the definition and implementation of tools that automatically perform static and run-time optimization, both for performance and for scalability. These tools are strictly based on the definition and validation of a cost model for all mechanisms used in the language runtime implementation.

We describe our experience in developing a set of low level mechanisms, based on message passing paradigm, that constitute an intermediate “language” for inter process communication and are able to support developing a runtime for structured parallel programming constructs. The Appendix details two achievements:

I. We define a set of communication mechanisms featuring a clear cost model for inter-process interactions.

II. We show how to exploit the knowledge about structured parallel constructs in order to implement optimized strategies for communications mechanisms.

We refer to the set of communication mechanisms with the terms communication support while we name communication language the language that manages them.
We therefore present a generic description of a communication support for structured parallel programming and its implementation on the IBM Cell BE CMP architecture. Performance tests show that bandwidth and latency obtained for one to one communication between processes on different cores are similar to the actual hardware performance limit. We show that how our implementation achieves the lowest possible overhead and we compare our results with the IBM Cell BE MPI implementation showing that our solution provides one order of magnitude improvement.

Finally we define a cost model of the communications mechanisms, which makes possible to implement parallel constructs, defining their specific cost model and study algorithms for automatic optimization.

A.1 Structured Parallel Programming

The key idea of structured parallel programming (SPP) is to describe a parallel application using only a specific set of parallel mechanisms or parallel constructs (PCs), and combining them. PCs are based on concepts like replicating or partitioning of function or of data, e.g. farm, map and stencil data parallelism, divide&conquer, etc.[12, 18, 29, 6, 47, 5]

This methodology provides a straightforward strategy for analyzing and parallelizing applications, thanks both to the separation of parallel and sequential aspects and to the hiding of all low level details on communication and synchronization.

SPP methodology alone is not a complete solution for CMP programming or parallel programming in general. While PCs allow for an easy description of a parallel application, the development, configuration and tuning of the corresponding program is not a simple task. An efficient implementation of a parallel construct on an architecture requires a deep knowledge of important low level aspects of the specific target architecture, as well as good skills for design and management of synchronization and communication.

Moreover a structured parallel application must take into account configurations that better overlap communication and computation. A load balancing strategy is mandatory when managing irregular problems. Communications between processes should be vectorized (i.e. grouped) to obtain a lower overhead. Finally the individual sequential processes, which set up the parallel application, must be mapped and scheduled on the physical resources. Algorithm that find exact solutions for most of these problems are proved NP-hard.

We study SPP with the purpose of defining algorithms to find reasonable approximate solutions for the previous NP-hard problems, exploiting information that can be extracted from a deep knowledge of the PCs semantic and a PCs cost model over a target architecture.

Our approach to parallel programming consists in defining a complete framework based on SPP methodology. The framework includes a system implementing a set of
PCs and a set of algorithms to manage both optimization and configuration required to run an efficient parallel program.

We structure our framework in four hierarchical levels (see Figure A.1); from top to down we find application, structured, concurrent and hardware-firmware levels. Each level defines, and at the same time is based on, a programming language featuring a particular abstraction degree. We recall that we use the term “level” and “language” in an interchangeable way. The architecture we present now is a generalization of the one that we introduced in Chapter 3.

The structured language provides to application developers high level constructs to design and implementation of parallel applications according to the SPP methodology. Developers declare and aggregate PCs, composing them to achieve the desired application behavior. Programmers can therefore rely on a high degree of abstraction since all low level details are completely hidden inside the individual PC implementation.

The concurrent level, which represents the communication support, is characterized by a language that express parallel processing as a set of interacting sequential computational kernels. In general threads can interact using either a shared memory or a message passing mechanism; in our framework we use a message passing paradigm. This language is designed in such a way to facilitate implementing, modeling and configuring a system for SPP.

Following the hierarchical relation between levels, all the parallel constructs of the structured language are implemented by strategies defined in the concurrent language; for example a farm construct can be described as process (master) which uses some communication mechanism to distribute tasks to a set of workers processes.
and then collects results. Since we require the strategy (i.e. the implementation) of PCs to be portable and independent from the specific target architecture, we use the concurrent language to completely mask the characteristics of the underline concrete architecture.

The lowest level of the system is the hardware-firmware, where the basic support for managing sequential processes and communication is implemented. At this level a parallel application is described by a sets of operations on hardware registers of the target architecture.

Every system level must be characterized by a cost model, which is an essential element for optimization and configuration purpose. In this Appendix we focus on the two bottom layers: concurrent and hardware-firmware layers.

Several different message passing libraries are available in industry and academia, for example *MPI* is the most used communication support in HPC. Most of these solutions provide enough mechanism to define PCs. Our team has an extensive experience in developing SPP environments on a wide range of architectures: from cluster to grids In those environment we used standard communication supports for many reasons, as portability, compatibility and saving of developing time.

In a *CMP* environment standard mechanism are too heavy weight, since they were designed for classical MPP architectures, so we decided to distill the required mechanism into a simple and low overhead interface.
A.2 \( \mathcal{LC} \) Language Semantics and Syntax

\( \mathcal{LC} \) is the language we defined for the concurrent level; it is based on a message passing paradigm and describes abstract processes communicating according to a local environment model. \( \mathcal{LC} \) was born as a tailored minimal formalism of the CSP of Hoare [21].

A \( \mathcal{LC} \) parallel program statically declares a set of processes, that can also be parametrically described with respect to some identifiers. Processes do not share any data structure: channels, statically defined in the program (no channel can be created at run time), are the only mechanism that describe interaction between entities.

A channel is unidirectional and typed: it represents a queue where a set of producer processes insert messages and a single consumer process extracts data. The type of the channel and the type of the messages in its queue must match. Channels can implement either a symmetric communications, when only one producer is defined, or asymmetric communications.

An important characteristic of a \( \mathcal{LC} \) channels is the asynchrony degree, which is a static integer parameter defining the maximum number of non blocking send operations that can be performed when no receive operation is invoked. When the parameter is equal to zero, the channel is synchronous by definition. The asynchrony degree is statically defined and has to be guaranteed by the implementation of the language for the life time of the \( \mathcal{LC} \) program.

It is important to observe that semantically asynchronous MPI communications are equivalent to \( \mathcal{LC} \) channels with an infinite asynchrony degree. MPI programmers don’t need to define an asynchrony degree but, as consequence, they prevent the MPI support to exploit important information for optimization as we will show later.

Finally \( \mathcal{LC} \) includes an alternative guarded command, similar to the one of CSP, to manage non deterministic aspects; this is a relevant key construct for dynamic load balancing aspects.

A.2.1 \( \mathcal{LC} \) Channel API

We developed the first version of \( \mathcal{LC} \) implementation on the IBM Cell BE architecture, implementing a C++ library called MammuT (cell Multicore Architecture coMMUnication supportT). We used C++ language rather than C to exploit an object oriented interface and to benefit from template mechanisms both for static type checking and compile time optimization.

The MammuT library includes a set of methods that describe, also in a parametric way, the graph of an application and therefore to define processes, channels and bindings. In this article we do not describe the declaration of processes and channels: we focus only on the channel construct interface, implementation and performance.
In MammuT library a channel is a C++ template object with the following signature:

```cpp
template <int Role, type msg_type,
           int Async_degree, int Window_length>
Spe_channels <Role, msg_type,
              Async_degree, Window_length>
```

The `Role` template parameter sets the channel as in-coming or out-going, `Type` parameter establishes the type of messages that can be sent over the channel and the `Async_degree` determines the asynchrony degree. The `Window_length`, which is detailed in section A.4, defines a logical life time on receives messages on a specific channel.

The runtime interface of a MammuT channel includes the following methods:

- `int transfer_id send(msg_type * msg)`: is used to send messages over a channel. The method returns an identifier for the transfer.
- `void wait(int transfer_id)`: is used to wait for the completion of the send operation identified by the `transfer_id`
- `msg_type * receive()`: is used to retrieve a message from an in-coming channel.

### A.2.2 MPI and LC

We are not studying LC as a language to replace MPI. We investigate LC because its expressiveness is sufficient to express PCs and it presents static features that can be exploited to reach high performance.

While comparing the MPI interaction model with LC, it is evident that the second one requires several static information about the communications: in LC you must specify both end points of every channel, the message type and the fixed asynchrony degree. These static information are relevant to introduce optimizations and to target more efficient communication protocols. The same information are required also when using MPI communication, but they are available only at run time.

Along with performance optimization, the complete knowledge of the communications between processes, and especially the characteristics of a fixed degree of asynchrony is enough to characterize the LC program memory requirement for asynchronous communications. This is a sensitive aspect especially while targeting architecture with memory limitations such as the SPE cores inside the Cell BE multi-core.

The MPI standard states that a program is "safe" only if it relies on synchronous communication [45]. This limitation is due to the inability to guarantee that the
memory required for buffering asynchronous communication can be allocated at runtime; if memory is not available the parallel application will deadlock.

In MPI model, the Buffer Allocation Problem (BAP), which is the problem of determining the minimum number of buffer required to ensure a deadlock free execution, it is proved to be NP-hard independently from the actual communication buffer management strategy [9]. Moreover the Buffer Sufficiency Problem (BSP), which is the decision problem to determine if a given buffer assignment is sufficient for deadlock free executions, is still intractable when exploiting receiver buffer allocation strategy [9], which is the only mechanism that provides zero copy communication.

In contrast to MPI, LC solves by design the problem of buffer allocation. If it is not possible to allocate at start-up time enough space for the required buffers, whose dimension is known at compile time thanks to LC static semantics, the program cannot be executed. Thus once the program starts, programmers don’t need to take into considerations deadlock situation due to buffer allocation.

LC features make it possible the introduction of several optimizations (detailed in Section A.3 and A.4), but we are not proposing that LC should be used instead of MPI: LC semantic has been defined to support the definition of parallel skeletons only. We use the LC language for the concurrent level instead of MPI, in order to exploit the performance gain that can be extracted from more rigid semantics.
A.3 LC Channel Abstract Protocol

In this Section we detail the abstract implementation of a one-to-one LC channel, independently from the specific architecture. This abstract description helps us to define important characteristics of our channel support and to show some optimization and the LC channel peculiarities that allow for them. This abstract description is an introduction for the concrete implementation on the Cell architecture.

LC one-to-one channel are typed and asynchronous: a send operation can terminate before the corresponding receive is called. To implement this behavior it is mandatory to provide some memory locations where messages can be written. The number of messages already inserted into the channel plus the number of predefined memory location to store future messages must be equal or greater than the channel asynchrony degree $k$. We call this the $k$ – property.

A one to one communication channel is normally represented as FIFO queue

![Diagram](image-url)

**Figure A.2:** Pseudo code and data structure used by the LC channel abstract protocol

```cpp
send(ch_type the_msg){
    pop(Void_msg_queue, new_msg);
    write(new_msg, the_msg);
    push(Written_msg_queue, new_msg);
}

ch_type receive( ){
    pop(Written_msg_queue, new_msg);
    get_void_msg(Refilling_mechanism,void_msg);
    push(Void_msg_queue, new_entry);
    return new_msg;
}
```
containing the messages inserted into it. In our representation, as shown in figure A.2(a), we consider a second queue to model the preallocated messages, required by the \( k \) – property. We name the first queue `Written_msg_queue`, the second one `Void_msg_queue` and we define respectively \( q \) and \( m \) their lengths.

Both queues are shared between sender and receiver processes, and are managed using only push and pop operations. The protocols followed by sender and receiver are described in pseudo code in figure A.2(b), which shows that the behavior of the two are somehow dual.

The sender first pops from `Void_msg_queue` a reference to a memory location where the message can be written and, after writing the data, it pushes the reference back into the `Written_msg_queue`. The receiver starts popping from the `Written_msg_queue` a reference to a sent message and then, before returning the message, it defines and inserts a new memory location into `Void_msg_queue`.

As previously highlighted by the \( k \) – property, the sum of \( q \), the number of messages that have been sent but not yet received, and \( m \), number of memory location predefined to store future messages, has to be always equal or greater than \( k \).

This protocol preserves the \( k \) – property. During initialization phase, we fill the `Void_msg_queue` with \( k \) references and we empty the `Written_msg_queue`; as consequence, we have \( q + m = 0 + k = k \). From this point on, both send and receive procedures maintain the sum of \( q \) and \( m \) equal to \( k \). At each call, the send procedure decreases \( m \), by extracting an element from `Void_msg_queue`, but increases \( q \), by inserting an element in `Written_msg_queue`. The reverse happens when handling a receive operation, which decreases \( q \) and increases \( m \).

It is important to highlight two protocol features: first it requires that pop operations are blocking only when the queue is empty; and secondly it does not require any queue length check for push operations: push operation are always allowed.

Finally, sender push and pop operations are performed on a different queue with respect to receiver push and pop operations. These queues are then characterized by one producer and one consumer: their implementation can use a lock-free algorithm, even without mechanism such as fetch-and-add or compare-and-swap, as shown by Lamport [36, 17]. This property is relevant in order to target high performance especially on the Cell, where atomic operations for SPE core exhibits a high latency.

It is important to underline that most of the nice characteristics of the protocol we have presented are strictly linked to channel static typed and fixed asynchrony degree: the asynchrony degree gives information about the minimum number of preallocated messages required to guarantee the \( k \) – property, while the type information specifies the memory required for every message.

The `get_void_msg` method wraps the possible mechanism for defining memory location where future messages can be written. Because of the utility of those mechanisms we refer to them as refilling mechanisms; several refilling solutions are possible and we'll present and analyze them in the following section.
A.4 Channel Abstract Optimization

A.4.1 Static Refilling: the \textit{w\_protocol}

We begin the abstract channel performance optimization section with the definition of an efficient \textit{refilling} mechanisms.

A first possible solution is based on dynamic memory allocation: at each receive a new memory chunk is allocated and its reference is inserted into the \textit{Void\_msg\_queue}. This \textit{refilling} mechanism implies an allocation overhead at each receive and requires \textit{LC} programmers to manage the deallocation of received messages. This solution also implies that the parallel program implementation will not be deadlock free.

A second potential solution requires the programmers to pass as parameter of the receive method a reference to a new memory location for receiving the message. With respect to the first solution, this \textit{refilling} mechanism avoids the dynamic allocation overhead, but both solutions manage message references that are not statically defined.
A third possible implementation is based on statically defined references. This last solution is obviously not compatible with a zero copy strategy: at the receiver side messages have to be copied from the static buffer to another memory location. The mechanism therefore reduces data traffic but increases the receive latency with the message copy overhead.

We selected an alternative solution that can be implemented with static messages references, to avoid additional communication between sender and receiver, but allows for a zero-copy feature that avoids unnecessary messages copy. The mechanism is based on the introduction of additional constraint on the semantic of the received messages.

In most of the PCs, such as farm, pipe and also often in data parallel constructs, the behavior of the threads implementing the pattern can be schematized as “activations”. During each activation few messages are received and the data are used as parameters by the computational kernel; after the computation is completed a set of response messages is sent. The messages received can also be used to modify the state of the process but at the end of the processing step they are not needed any more and can be deallocated.

To take advantages from this characteristic parallel construct behavior, we introduce a refilling mechanism based on buffer entry reuse, which introduces a usage time restriction on received messages.

We start from the abstract representation of the channel protocol in Figure A.2(b) and we consider a third queue, called Window_msg_queue, which is used only by the sender (see Figure A.3). We define $w$ as the number of element in this queue and we require it to be a static constant defined by programmers. At channel initialization we allocate $k + w$ messages and we push $k$ messages reference into the Void_msg_queue and $w$ into Window_msg_queue.

The new protocol on the receiver side (Figure A.4) pops the reference of the message to be received and pushes it directly into the Window_msg_queue. The receiver refilling mechanism simply pops a reference from Window_msg_queue and pushes it into Void_msg_queue; the $k – property$ is still guaranteed.

Following the path of a single message reference for a set of send call, it is evident that it is periodically reused to write new messages. Therefore a received message is characterized by a logical life time, which depends by the $LC$ channel parameter $w$. A received message can be used, without introducing incoherence, until its reference remains into the Window_msg_queue.

From the programmers prospective, a received message can be accessed for a limited time. When the $w + 1$ receive function call is done, the message data will not be valid any more since its reference is extracted from the Window_msg_queue and pushed into Void_msg_queue.

Programmers can statically set the value of $w$ depending on the parallel construct behavior in accessing incoming messages. The higher the value $w$ is, the longer the messages time life is. In most cases, where only a single message is used to send parameters to a computational kernel, a value of $w$ equal to one is enough.
to guarantee zero copy. It is relevant to underline that the parameter $w$ does not modify the semantic of the asynchrony degree.

We call the abstract protocol optimized with the refilling mechanism the $w$-protocol.

### A.4.2 The $K\_plus\_one$ Optimization

![Data structured used by the $K\_plus\_one$ optimization](image)

**Figure A.5:** Data structured used by the $K\_plus\_one$ optimization

```plaintext
send(ch_type the_msg){
    pop(Plus_one_queue, new_msg);
    write(new_msg, the_msg);
    push(Written_msg_queue, new_msg);
    pop(Void_msg_queue, next_msg);
    push(Plus_one_queue, new_msg);
}
```

**Figure A.6:** Pseudo code of the $K\_plus\_one$ optimization

We now describe an optimization strategy which allows for a better overlapping between communication and computation of the send procedure. Although this optimization concerns only the sender side, and therefore it is completely independent from the specific refilling mechanism, we present it in the contest of the $w$-protocol. This new optimization uses another queue called $Plus\_one\_queue$ (see Figure A.5). The queue is managed only on the sender side and the number of references it stores, represented by the variable $p$, is constantly equal to one.

In the new send procedure (figure A.6), the sender first pops from $Plus\_one\_queue$ a message reference, which is used to write the message to be sent, and pushed it
back into $\text{Written}_\text{msg}_\text{queue}$. Finally to keep $p$ equal to one, the send support pops a reference from the $\text{Void}_\text{msg}_\text{queue}$ and pushes it directly into the $\text{Plus}_\text{one}_\text{queue}$.

As in the other protocol version, at initialization $k$ references are defined and inserted into $\text{Void}_\text{msg}_\text{queue}$, but another reference is defined and inserted in $\text{Plus}_\text{one}_\text{queue}$. The total number of references requested at the channel setup is equal to $k + w + 1$; for this reason we refer to this optimization with the name $k\_\text{plus\_one}$.

The $k - \text{property}$ is still guaranteed by the length of $\text{Written}_\text{msg}_\text{queue}$ and $\text{Void}_\text{msg}_\text{queue}$: in other words, because the patterns of access to the shared queues has not changed, we still have $q + m = k$.

The benefits of introducing the $\text{Plus}_\text{one}_\text{queue}$ are evident comparing the send pseudo codes in fig A.4 and A.6. The improved performance comes from the fact that the message copy in the $k\_\text{plus\_one}$ version is computed before the pop operation on the $\text{Void}_\text{msg}_\text{queue}$. The pop on the shared queue is the only operation that can potentially require a wait. The pop on $\text{Plus}_\text{one}_\text{queue}$ never waits since, between a send and the next one, $p$ is always equal to one: the queue is never empty.

Thus a possible stall, due to the blocking semantic of the pop operation, is postponed in time, allowing the computation of some useful work. When a blocking pop returns the send procedure in the $k\_\text{plus\_one}$ version must still copy the message.

Moreover, neglecting the latency of the pop operation on the private $\text{Plus}_\text{one}_\text{queue}$, the write is the first operation to be computed. If the architecture supports asynchronous writes, the operation overhead is overlapped to the rest of the protocol.
A.5 Concrete Implementation on the Cell

A.5.1 The Cell Architecture

The Cell chip contains nine heterogeneous computational elements linked through a component called the Element Interconnection Bus (EIB). This element is composed by four 16-byte-wide data rings, each allowing up to three concurrent data transfers (if there’s no physical overlap). The EIB acts like a connection oriented network and supports transfers of blocks up to 128 bytes. See [4] for an in-depth presentation of the EIB.

The other Cell elements are: the Power Processing Element (PPE), eight Synergistic Processor Element (SPE), one Memory Interface Controller (MIC) and two I/O units. The PPE is an “in-order” 64-bit IBM Power 4 processor with two threads. Each SPE features a Synergistic Processor Unit (SPU), a memory flow controller (MFC) and a very fast 256-Kbyte SRAM memory (called local store - LS) with a 6 clock latency; the SPU is a RISC-style CPU optimized for vector operations.

The MFC acts as a memory management unit and a DMA engine; DMA is the only way to move data between the local store and other SPE or the main memory. The DMA engine schedules operations asynchronously and in an unordered way but some DMA commands (fence and barrier) are provided to force an ordering.

A.5.2 Channel Implementation on Cell

In this section we describe the implementation of LC channels for the Cell architecture based on the w-protocol with the k_plus_one design.

The w-protocol allows us to statically allocate a vector of messages ($r_msg_vect$) of length $k + w + 1$; this is the only buffering support that is going to be exploited. To guarantee zero copy, the buffer is allocated into the LS of the receiver process. A vector of references ($s_ref_vect$), each one pointing to a corresponding location in $r_msg_vect$, is instantiated on sender side. A reference can be used to write into the associate message through a DMA operations.

As the receiver message buffers are statically allocated at initialization time and its entries are reused during the program run, $s_ack_bit_vect$ is initialized once and never changed: no information exchange from receiver to sender is required to keep message references up to date, therefore saving both latency and bandwidth.

To avoid synchronizations trough lock mechanisms, which in IBM Cell BE architecture exhibits high latency and can not be completely delegated in an asynchronous way to the MFC, we follow an alternative implementation of the Lamport algorithm [36].

Lamport technique works for one consumer one producer queues and it is based on the management, in a non atomic way, of a top reference, modified only by the receiver, and a bottom one , modified only by the sender. The read of a not up to date value, that can result in a stall due to full or empty queue status, is followed
Figure A.7: Cell \( \mathcal{LC} \) channel data structures A.7(a) and its concrete protocol compared with the abstract one (A.8(a)).
send(ch_type the_msg){
   WRITE_DMA(new_msg, s_ref_vect[index]);
   set_bit(r_syn_vect[index]);
   ... 

ch_type received(){
   set_bit(r_syn_vect[(index+k)%(w+k+1)]);
   wait_bit(r_ack_vect[index]);
   msg_received = r_msg_vect[index];
   index=(index+1)%(w+k+1);
   return msg_received;
}

ch_type receive(){
   pop(Window_queue, new_entry);
   push(Void_msg_queue, new_entry);
   pop(Written_msg_queue, new_msg);
   push(Window_queue, new_msg);
   return new_msg;
}

Figure A.8: Cell LC channel data structures A.7(a) and its concrete protocol compared with the abstract one (A.8(a)).
by retries. This solution well fit the IBM Cell BE environment where SPE are mono-programmed and can efficiently use busy waiting.

We follow a slight different approach, that is semantically equivalent to Lamport. Each couple, formed by an element of $r_{msg\_vect}$ and the corresponding one of $s_{ref\_vect}$, is associated with two bits of information. Bits are logically structured, as presented in the Figure A.7(a), in two vectors: $s_{ack\_bit\_vect}$, allocated into sender LS, and $r_{syn\_bit\_vect}$, in the receiver LS.

The semantic of a bit is straightforward: a bit of $s_{ack\_bit\_vect}$ is equal to one when its message belongs to the Void$_{msg\_queue}$. Symmetrically when a $r_{syn\_bit\_vect}$ is equal to one the corresponding message belongs to the Written$_{msg\_queue}$.

We need two operations on bits in these vectors: wait$_{bit}$ and set$_{bit}$. The first operation, which will be used only on local data, blocks until the bit value is equal to one and resets it to zero: the operation stalls until the message is into the Void$_{msg\_queue}$ and automatically extracts it. The set$_{bit}$, that is used only on remote bits, sets a bit value to one; the operation is used to insert a message into the Void$_{msg\_queue}$.

At initialization we specify the bit values as shown in figure A.7(a) which describes the mapping between the data structures entries and the queue of the abstract protocol. Since at the program startup no message is present in the Written$_{msg\_queue}$, all $r_{syn\_bit\_vect}$ bits are set to zero.

On the sender side we associate the first entry with the Plus$_{one\_queue}$, thus its associated bit is set to zero. The following $k$ entries are mapped into the Void$_{msg\_queue}$ and their bit is set to one. Finally the rest of the vector is mapped with the Window$_{msg\_queue}$ and all the bits are set to zero.

Both sender and receiver work on the data structures in a circular way incrementing a local reference (called index) that points at the vector entries that must be used. At initialization the value of index are equal to zero for both sender and receiver: for the sender it points to the entry associated to the Plus$_{one\_queue}$. In the receiver the variable points to the first position that is going to be inserted into Written$_{msg\_queue}$. The complete protocol to manage the all the data structures is shown in Figure A.8(a), where the WRITE$_{DMA}$ represents a DMA transfer.

From an architectural prospective, set$_{bit}$ is an operation is called by the program on the SPU but its execution is delegated to the MFC. A nice feature on the sender side is that both the first two operations, WRITE$_{DMA}$ and set$_{bit}$, are delegated to the MFC in an asynchronous way: the sender process does not wait for their completion. The remaining part of the send protocol is overlapped with the actual MFC transfer of the message and the associated set$_{bit}$.

The protocol described in previous section can be implemented on the Cell architecture using different low level mechanisms for set$_{bit}$ and wait$_{bit}$. The Cell provides two main techniques for synchronizing SPEs.

The first option is using point to point communication over a special MFC registers, called signal register. We call the channel implementation that exploit this kind
of mechanism signal-based implementation \((DMAS)\). This solution leverage both the blocking semantics of the signal registers (when reading a zero value) and its \(OR\) writing semantic: a write operation of a value on the signal register stores in the register the result of an \(or\) operation between the old value and the new one. The \(or\) behavior, which can be set when loading a program on a \(SPE\), is useful to introduce optimized computation based on bit masks. The \(SPE\)-\(SPE\) communications though signal register are performed by \(DMA\) transfer. To distinguish those \(DMA\) communications from the others transfer we refer to them as signal.

The second synchronization technique is equivalent to the previous one but it performs polling on local storage locations and not on special registers. We can define two different channel implementation for this strategy depending on the number of transfer used: a single transfer \((DMA1)\) and a double transfer \((DMA2)\).

### A.5.3 Signal-based Implementation

In this solution data movement is accomplished using a \(DMA\) operation, while the \(set\_bit\) primitive is implemented by sending a signal to the remote \(SPE\). The signal must be sent using a \(fence\) instruction to ensure that the presence bit is set to one only after the first \(DMA\) completes. Thanks to the \(MFC\) unit, the \(SPU\) can simply queue the \(DMA\) transfer, which continues asynchronously, and than check for the empty queue event. The overall latency of this implementation is composed by the \(DMA\) transfer and signal latencies. In fact, since the queue empty test is performed locally, no overhead is added over the physical communication.

### A.5.4 DMA1 Implementation

Analyzing the signal based implementation it is clear that adding one \(EIB\) transfer only for synchronization purposes adds the overhead of establish a new connection over the \(EIB\) ring for moving the minimum message size (16 byte). To avoid this overhead we implemented differently the \(set\_bit\): this solution merges both the message and the synchronization in the same \(EIB\) connection.

We consider the message as a structure, named “enlarged” message: it contains the original messages plus a single byte that acts as a bit of \(r\_syn\_bit\_vect\), thus all buffer entries on the receiver side are “enlarged messages”. The \(wait\_bit\), instead of polling on a signal register, polls into the local storage the synchronization byte of the message that has to be received. The main requirement for this solution to work correctly is that the synchronization byte must be written after the message data: we must know the details of the \(EIB\) transfer policy.

In the current Cell architecture there is no guarantee about the order in which a set of \(DMA\) operations are executed except when inserting a fence or barrier instruction; moreover when a single transfer is larger than 128 bytes it is splitted in more transfers which are executed out of order. Unfortunately there is no single mechanism to guarantee that the synchronization data byte is written last.
A.5. CONCRETE IMPLEMENTATION ON THE CELL

Our \textit{DMA1} implementation works only for special messages where the enlarged message location is 128 byte aligned and its size is up to 128 bytes. When this holds the \textit{MFC} transfers the data in order, starting from the lower address. Inserting the synchronization byte after the message data will ensure the correctness of the operation.

A.5.5 DMA2 Implementation

The last implementation strategy we tested is still based only on \textit{DMA} operations but it is able to transfer messages larger than 128 bytes. The basic idea is still to exploit the compile time knowledge of type size and structure the physical data layout to transfer the information efficiently over the EIB.

The weakness of the signal implementation is that delivering a signal requires establishing a connection over the EIB to transfer the 32 bit signal word. For example if we have a message containing 184 bytes, the signal based implementation will create 3 separate connections over the \textit{EIB}: the first moves the initial 128 bytes, the second moves the next 56 bytes and the last delivers the 4 byte signal word. The \textit{DMA2} solution is still based on the idea of merging message and synchronization data and achieves this by attaching the synchronization byte with the last “not full” \textit{DMA} block. This requires messages to be 128 byte aligned and the implementation of the \textit{set\_bit} consists in an initial \textit{DMA} transfer of the largest message chunk multiple of 128 byte followed by a second \textit{DMA} operation to transfers the remaining data plus a char for the synchronization. Between the two transfer operation a fence instruction is inserted to ensure that the synchronization data is written after the message content. The double \textit{DMA} implementation exhibits a communication latency that correspond to the physical latency of the two fenced \textit{DMA} transfer.
Figure A.9: Comparison of latency and bandwidth performance of different LC channel implementation and of DMA transfer.
A.6 The Cost Model

We evaluated our LC channel implementations on an IBM QS21 IBM Cell BE blade; the results are detailed in A.9(a) and A.9(b). All tests were run in unload EIB scenario, therefore avoiding any kind of conflict on the interconnection structure. Since the DMA1 and DMA2 implementations are respectively developed for short and long messages, they are presented by the same curve tagged DMA1/DMA2.

The chart in Figure A.9(a) represents, for different message size, the latency of the channels implementations (DMA1/DMA2 and DAMA) and the one of the pure DMA (DMA).

The DMA1 implementation reaches the optimal latency for short messages: equal to the latency of the actual DMA transfers. This means that for messages smaller than 128 bytes, the overhead of sending an “enlarged” message instead of the pure one is negligible.

For longer messages DMA2 and DAMA are quite similar; they do not express relevant performance differences. Both of them feature an overhead, respect the pure DMA, lower that 200 clock cycles.

We can assert that it is not possible to develop a IBM Cell BE communication mechanisms that reaches better performance. In each implementation we exploit, according to the IBM Cell BE hardware and firmware architectural constraints, the minimum number of transfer to copy the message and advise about the operation completion.

All the three protocols follow a lock free algorithm to manage channel shared queues. This means that we can completely avoid two negative characteristics of IBM Cell BE atomic operation: they present a high latency and they can not be completely delegated to the MFC. With lock free mechanisms, our implementations present a service time of about 140 clock cycles; this is the time a send call spends before returning. The measured value is lower than the lowest latency of LC channel: in any configurations the protocol overhead is completely overlapped to the DMA communication required.

The chart in Figure A.9(b) shows how the performance bandwidth of the LC channels saturate up to the 85% of the maximum bandwidth feature by DMA transfer.

Comparing these results with MPI performance reported in [27, 32, 48] we can see that our solutions achieves more than one order of magnitude better latency: for a 128 bytes message our DMA1 implementation uses just 250 clocks while MPI implementation uses 6500 clocks. Our solutions provides also a much higher bandwidth: for example on 64 kbyte messages, DMA2 transfers data at almost 20 Gbps and Cell MPI run time provides just 6 Gbps. The main impact on the performance difference is the fact that the strict semantic of LC allows us to pre-allocate buffers in the receiver local storage and never copy incoming data.

As previously mentioned one of the most important aspect for developing a framework for structured parallel programming is the availability of a detailed cost
model of the communication; this is a fundamental element to predict program performance and to compare possible different optimization and tunings. The cost model of our implementation is the following:

\[
L_{\text{com}}(msg) = \begin{cases} 
(0.1363 \times \text{size}_{msg} + 365) \text{ \textit{clocks cycles}} & \text{if } \text{size}_{msg} > 128 \text{ \textit{bytes}} \\
250 \text{ \textit{clocks cycles}} & \text{if } \text{size}_{msg} < 128 \text{ \textit{bytes}} 
\end{cases}
\]  

(A.1)

The current limitation of this solution is the requirement of pre allocated buffers on receiver side which, given the size of SPE local store, can be problematic. We are currently investigating several solutions to this issue.
A.7 Conclusion and Feature Works

We approached the issue of building an high performance and predictable support for handling data communication for a structured parallel programming framework for chip multi-core systems. We review the available communication libraries and conclude that they are too heavy weight to be used in chip multi-core systems, therefore we design the \( \mathcal{L} \mathcal{C} \) a communication language, which is a sub set of the CSP model. We then detail the abstract communication protocol and some abstract optimization strategies for the \( \mathcal{L} \mathcal{C} \) semantic.

We describe our concrete implementation of MammuT, a library for \( \mathcal{L} \mathcal{C} \), on the IBM Cell BE, where we exploit all the available features of the architecture to focus on achieving highest possible performance. Experimental results, obtained on an IBM QS21 Cell blade, show that both MammuT latency and bandwidth are very similar to the actual hardware and firmware performance. More over, our implementation adds a minimal overhead and provides 1 order of magnitude improvement over IBM Cell BE MPI implementation. The results obtained derive from static information extractable from \( \mathcal{L} \mathcal{C} \) that allows the development of more performed and optimized communication protocols.
Bibliography


Index

\[\Delta^\text{in},\ 81\]
\[\Delta^\text{in}_{\text{com}},\ 85\]
\[\Delta^\text{out},\ 82\]
\[\Delta^\text{out}_{\text{com}},\ 85\]

\[Q^-\text{transformations}\]
  Generic \(Q^-\text{transformations}\), 105
  Negative \(Q^-M^-\text{transformations}\), 191
  Negative \(Q^-\text{transformations}\), 134
  Positive \(Q^-M^-\text{transformations}\), 187
  Positive \(Q^-\text{transformations}\), 110

Equivalence Operations
  Complete Step Equivalence, 41
  Shape Equivalence, 42
  Structural Step Equivalence, 41

Homogeneous Uniform Affine Model, 64

IDR, 79
IIR, 78
Incoming Communication Set, 85
Incoming Dependent Region, 79
Incoming Independent Region, 78

Movement Vector, 78

Neighbour Index Set, 78
Neighbour Set, 78

ODR, 81
OIR, 82
Outgoing Communication Set, 85
Outgoing Dependent Region, 81
Outgoing Independent Region, 82

Oversending, 143
Owner-Computes Rule, 11, 27
  in the Structured Model, 27

Partition Space, 76
Partition Space Length, 76

Relaxed Computational Equivalence, 68

Space Invariant Stencil, 22

Spatial Structure, 24
  General Spatial Structure, 24
  Regular Spatial Structure, 24
  Toroidal and Regular Spatial Structure, 24

Stencil Classes
  Affine Space Invariant, 45
  Homogeneous, 43
  Semi Uniform, 48
  Uniform Affine, 46

Step Fusion Transformation, 154
  Correctness, 155
  Support Transformation, 154

Step Set, 22

The Incoming Partition Dependency Set, 81
The Outgoing Partition Dependency Set, 82

Working Domain, 25