University of Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking
Corso di Laurea Magistrale in Informatica e Networking

Master Thesis

# Optimization Techniques for Stencil Data Parallel Programs: Methodologies and Applications

Candidate
Andrea Lottarini

Supervisor
Prof. Marco Vanneschi

Academic Year 2011/2012

# Contents


**1 Introduction** **1**

1.1 Structured Grid Computations . . . . . . . . . . . . . . . . . . . . 1

1.2 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Data Parallel with Stencil Computations** **7**

2.1 Map Computations . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Data Parallel with Stencil Computations . . . . . . . . . . . . 9

2.2.1 Classification of Stencil Computations . . . . . . . . . 9

2.3 Structured Grids in the Real World . . . . . . . . . . . . . . . 10

2.3.1 Finite Difference Method . . . . . . . . . . . . . . . . . 11

2.3.2 Discretization of Partial Differential Equations . . . . . 12

2.3.3 Iterative Methods to Solve PDE . . . . . . . . . . . . . 13

2.3.4 Heat Equation . . . . . . . . . . . . . . . . . . . . . . . 15

2.4 Properties of Structured Grid Computations . . . . . . . . . . 19

2.4.1 Convergence . . . . . . . . . . . . . . . . . . . . . . . . 20

2.4.2 Boundary Conditions . . . . . . . . . . . . . . . . . . . 21

2.4.3 Stencil Coefficients . . . . . . . . . . . . . . . . . . . . 22

2.4.4 Notable Examples and Conclusions . . . . . . . . . . . 22

**3 A Hierarchical Model for Optimization Techniques** **25**

3.1 Optimization in a Hierarchical Model . . . . . . . . . . . . . . 26

3.2 The Reference Architecture . . . . . . . . . . . . . . . . . . . 29

3.2.1 Functional Dependencies . . . . . . . . . . . . . . . . . 31

3.2.2 Partition Dependencies . . . . . . . . . . . . . . . . . . 31

3.2.3 Concurrent Level . . . . . . . . . . . . . . . . . . . . . 35

3.2.4 Firmware Level . . . . . . . . . . . . . . . . . . . . . . 37

3.3 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**4 Optimizations for Structured Grids on Distributed Memory Architectures** **39**

4.1 $\mathcal{Q}$ transformations . . . . . . . . . . . . . . . . . . . . . . . 39

4.1.1 Positive and Negative $\mathcal{Q}$ transformations . . . . . . . . 41
4.1.2 $\mathcal{QM}$ and $\mathcal{QW}$ transformations . . . . . . . . . . . . . . 45
4.2 Shift Method . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.3 Step Fusion . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3.1 Ghost Cell Expansion . . . . . . . . . . . . . . . . . . . . 53
4.3.2 Collapsing Time Steps . . . . . . . . . . . . . . . . . . . 57
4.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**5 Optimizing for Locality on Structured Grids Computations 61**
5.1 Extended Model . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2 Analysis of the Impact of the Step Fusion Method on Memory Hierarchies . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.3 Classic Optimization Methods . . . . . . . . . . . . . . . . . 68
5.4 Loop Reordering Optimizations . . . . . . . . . . . . . . . . 70
5.5 Loop Reordering for Structured Grids Computations . . . . . 75
5.5.1 Loop Tiling . . . . . . . . . . . . . . . . . . . . . . . . 76
5.5.2 Time Skewing . . . . . . . . . . . . . . . . . . . . . . . 78
5.5.3 Cache Oblivious Algorithms for Structured Grids Computations . . . . . . . . . . . . . . . . . . . . . . . . . 83

**6 Combining Locality and Distributed Memory Optimizations 87**
6.1 Combining $\mathcal{SF}$ and Loop Reordering . . . . . . . . . . . . . . 87
6.1.1 $\mathcal{SF}$ and Tiling . . . . . . . . . . . . . . . . . . . . . . 87
6.1.2 Comparison of $\mathcal{SF}$ and Time Skewing . . . . . . . . . 90
6.2 Combining $\mathcal{Q}$ trasformations and Loop Reordering . . . . . . . 90
6.2.1 $\mathcal{Q}$ trasformations and Tiling . . . . . . . . . . . . . . . 91
6.2.2 $\mathcal{Q}$ trasformations and Skewing . . . . . . . . . . . . . . 94
6.3 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . 97

**A Experimental Results 99**

# List of Figures


1.1 Graphical representations of the functional dependencies of respectively *Jacobi* and *Jacobi* with applied $\mathcal{Q}$ trasformation stencils . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 The process of finding an approximate solution to the PDE via the Finite Difference Method. . . . . . . . . . . . . . . . . . 11
2.2 Three-point stencil of centered difference approximation to the second order derivative. This stencil represents Equation 2.3. 12
2.3 Discretization of temperature over a flat surface. The domain is divided into rectangles of width $h$ and height $k$. . . . . . . . 16
2.4 Five-point stencil for the centered difference approximation to Laplacean. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.5 Graphical depiction of periodic boundary conditions( Figure 2.5a ) and constant boundary conditions ( Figure 2.5b ) of a $3x3$ mesh. . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1 Framework for stencil computations as presented in [31]. . . . 30
3.2 Graphical depiction of the functional dependencies of Laplace stencil computation ( Figure 3.2a ) and its partition dependencies ( Figure 3.2b ). . . . . . . . . . . . . . . . . . . . . . 31
3.3 Perimeter of a partition with respect to partition area for row and block partitioning in a 2 dimensional working domain. . . 34
3.4 Dependencies between partitions with respect to parallelism degree for row or block partitioning in a two dimensional working domain. . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.5 Pseudocode representation of the Laplace stencil at the concurrent level. Partitions (or processes) are labelled by means of the coordinates with respect to the actual partition. . . . . 36
3.6 Pseudocode representation of the Laplace stencil at the concurrent level. With respect to concurrent code shown in Figure 3.5, we have that communication and computation overlaps. . 37

4.1 Graphical depiction of a time step of a mono dimensional jacobi stencil on a toroidal grid (Figure 4.1a) and a positive $\mathcal{Q}$ trasformation of the same stencil ( Figure 4.1b). . . . . . . . . 40
4.2 Graphical depiction of the incoming functional dependencies of Laplace stencil computation with positive $\mathcal{Q}$ Transformation applied. . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Graphical depiction of Partition dependencies in the case of the Laplace stencil (Figure 4.3a) and Laplace stencil with applied $\mathcal{Q}$ trasformation (Figure 4.3b). Incoming (Figure 4.3d) and outgoing (Figura 4.3c) communications at the concurrent level with respective regions. . . . . . . . . . . . . . . . . . . . . 43
4.4 Graphical depiction of two time steps of a mono dimensional jacobi stencil on a toroidal grid (Figure 4.4a) and the same computations with a positive $\mathcal{Q}$ trasformation applied in the first time step and a negative $\mathcal{Q}$ trasformation applied at the second step ( Figure 4.4b). The stencil kernel is the sum of the two elements of the shape. Notice that the final displacement of output values is correct. . . . . . . . . . . . . . . . . . . . 44
4.5 Graphical depiction of functional dependencies in the case of naive jacobi stencil (Figure 4.5a) and the Jacobi stencil with applied positive $\mathcal{QM}$ trasformation(Figure 4.5b). . . . . . . . 46
4.6 $\mathcal{QM}$ transformation of the one dimensional jacobi stencil computation presented in Figure 4.5. . . . . . . . . . . . . . . . . 46
4.7 Graphical depiction of the incoming communications of Laplace stencil computation (Figure 4.7a). Graphical depiction of the outgoing communications ( Figure 4.7b ) with $\mathcal{QM}$ trasformation applied. . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.8 Execution of transformed Laplace Stencil with $\mathcal{QM}$ trasformation(Figure 4.8a). Execution of transformed Laplace Stencil with $\mathcal{QW}$ trasformation(Figure 4.8b) . . . . . . . . . . . . . . 49
4.9 Graphical representation of incoming communications (fig. 4.9a) and outgoing communications (fig. 4.9b) of the nine point stencil. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.10 Graphical representation of the incoming communication pattern (fig. 4.10a) and outgoing communication pattern (fig. 4.10b) of the nine point stencil with shift method applied. Numbering reflects the order of the communications needed to preserve correctness. . . . . . . . . . . . . . . . . . . . . . . 52
4.11 Communication patterns of a generic stencil computation (left) compared with the same computation with oversending method applied (center) and with a $\mathcal{Q}$ trasformation applied (right). . 56

4.12 Step Fusion method applied to the two dimensional Jacobi stencil computation. . . . . . . . . . . . . . . . . . . . . . . . 57

5.1 Extended reference model. . . . . . . . . . . . . . . . . . . . . 63
5.2 Memory accesses with increasing step fusion factor in Jacobi computation. . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.3 Number of cache miss (normalized with respect of the number of time steps) for different size of the data matrix and a fixed number of cache locations (1k). . . . . . . . . . . . . . . . . . . 66
5.4 Elements accessed by the naive and the $\mathcal{SF}$ version of the jacobi 2d stencil. Red elements are updated by the naive version. Blue elements are updated by the $\mathcal{SF}$ version. . . . . . . . . . 67
5.5 Pattern of accesses to matrix A of Listing 5.9 . . . . . . . . . 72
5.6 Convex polyhedron which represent the iteration space of Listing 5.10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.7 Reuse of a generic element of the mesh in a jacobi 2d stencil. . 76
5.8 Graphical depiction of the optimized schedule of the two dimensional jacobi stencil . . . . . . . . . . . . . . . . . . . . . . 77
5.9 Figure 5.9b shows a graphical depiction of the pattern of updates in a optimized schedule for the 3d Laplace stencil shown in Figure 5.9a. . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.10 Graphical depiction of time skewing applied to a monodimensional laplace stencil. . . . . . . . . . . . . . . . . . . . . . . 79
5.11 Area of a trapezoid (cache block) for a generic mono dimensional stencil . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.12 Projection of a 3d space time lattice for a skewed 2 dimensional stencil with order equal to one. . . . . . . . . . . . . . . . . . 82
5.13 Time skewing that has to be reduced because the number of time steps is not sufficient. The optimal trapezoid (dotted) would have been substantially bigger than the actual trapezoids (shaded) . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.14 A generic trapezoid. As usual, the two slopes are derived from the data dependencies of the computation. . . . . . . . . . . . 84
5.15 Figure 5.15a shows a graphical depiction of a space cut. Figure 5.15b shows a graphical depiction of a time cut. . . . . . . . . 84

6.1 Number of cache miss with respect to step fusion level. The optimal value corresponds to $M^2/\mathbf{B}$. Matrix has a size of $2048x2048$ and the cache has a size of 1024 elements. . . . . . 91

6.2 Graphical depiction of time skewing on a mono-dimensional stencil. Different trapezoids should be executed from left to right. . . . . . . . . . 94
6.3 Graphical depiction of parallel time skewing. Different trapezoids belongs to different workers. The space-time portions labelled as “clean up work” should be updated after the adjacent trapezoids. . . . . . . . . . 95
6.4 Circular Queue optimization of a time skewed mono dimensional stencil. . . . . . . . . . 95
6.5 Execution of three steps of the proposed modified version of time skewing. . . . . . . . . . 96

A.1 Mean completion time of a single time step of the naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil executed on Andromeda. . . . . . . . . . 100
A.2 Communication time of a single time step of the of naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil executed on Andromeda. . . . . . . . . . 101
A.3 Computation time of a single time step the of naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil (without MPI communications) executed on Andromeda. . . . . . . . . . 102
A.4 Mean completion time of naive of the naive $\mathcal{Q}$ and $\mathcal{Q} + \mathcal{SH}$ version of the Jacobi stencil executed on Titanic. . . . . . . . . . 102
A.5 Total, computation and communication time of the two dimensional jacobi stencil (naive version) executed on Titanic. . 103
A.6 Mean completion time of naive and $\mathcal{Q} + \mathcal{SH}$ version of the 3d Jacobi stencil executed on Andromeda. . . . . . . . . . 104
A.7 Mean completion time of naive,$\mathcal{SF}^2$ and $\mathcal{SF}^2 + \mathcal{SH}$ version of the Jacobi stencil executed on Titanic. Values are normalized w.r.t the number of time steps. . . . . . . . . . 105
A.8 Comparison of Computation cost for both naive and $\mathcal{SF}^2$ version. Communication cost of the naive version is shown for reference. Notice that for sufficiently big partitions the computation cost dominates. . . . . . . . . . 105
A.9 Comparison of naive, loop tiling and time skewing version of the two dimensional laplace stencil. . . . . . . . . . 106
A.10 Speedup ($T_{naive}/T_{opt}$) of loop tiled and time skewed version with respect to the naive version of the two dimensional laplace stencil. . . . . . . . . . 107

# Listings

1.1 Pseudocode of a Jacobi update for the resolution of a two dimensional poisson equation. . . . . . . . . . . . . . . . . . . . 2
1.2 Pseudocode of reordered Jacobi update for the resolution of a two dimensional poisson equation. . . . . . . . . . . . . . . . . . 3
2.1 Pseudocode of a generic map computation. The computation is defined over 4 time steps. . . . . . . . . . . . . . . . . . . . 7
2.2 Pseudocode of the implementation of the map computation presented in Listing 2.1. Two real data structures are used and elements are updated sequentially. . . . . . . . . . . . . 8
2.3 Pseudocode of Laplacian Operator. . . . . . . . . . . . . . . . 9
2.4 Pseudocode of a Jacobi Iteration for matrix equation 2.7. . . . 13
2.5 Pseudocode of a Gauss Seidel Iteration for matrix equation 2.7. 14
2.6 Pseudocode of a Red Black Gauss Seidel Iteration for matrix equation 2.7. . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.7 Pseudocode of Jacobi method for the linear system 2.18. . . . 18
4.1 Pseudocode representation of a **two dimensional** nine point stencil, which has dependencies with every surrounding partition. Partitions (or processes) are labeled by means of coordinates with respect to the examined partition. . . . . . . . . . . 51
4.2 Pseudocode representation of a two dimensional laplace stencil with oversending method applied. . . . . . . . . . . . . . . . . 55
5.1 Pseudocode of naive 2d Jacobi stencil. . . . . . . . . . . . . . 62
5.2 Pseudocode of 2d Jacobi stencil with applied $\mathcal{SF}$ method. . . 62
5.3 Code of a mono dimensional laplace operator . . . . . . . . . . 70
5.4 Loop unrolling of Listing 5.3 . . . . . . . . . . . . . . . . . . . 70
5.5 Pseudocode that computes the sum of the columns in a matrix. We are assuming that the matrix A is stored in memory in row order. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.6 Pseudocode of the Loop Interchanged version of the algorithm shown in Listing 5.5 . . . . . . . . . . . . . . . . . . . . . . . 70
5.7 Pseudocode of the matrix vector multiplication . . . . . . . . 71

5.8 Pseudocode of the matrix vector multiplication with innermost loop tiled. b is a multiple of block size **B**. . . . . . . . . . . . . . 71
5.9 Pseudocode of the matrix vector multiplication with both loops "tiled". . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.10 Generic for loop with loop conditional as affine functions. . . . 73
6.1 Pseudocode representation of a generic structured grid computation defined at the concurrent level. The working domain is defined in two dimension and has size $MxM$. Depending on the stencil shape some of the SEND and RECV operations might not be necessary. . . . . . . . . . . . . . . . . . . . . . . . 88
6.2 Nested loops for the update of the incoming independent region of a two dimensional Jacobi stencil. . . . . . . . . . . . . . 89
6.3 Nested loops for the update of the incoming independent region of a two dimensional Jacobi stencil with $\mathcal{SF}$ applied. . . 89
6.4 Pseudocode representation of a generic two dimensional stencil ($\eta = 1$) with shift method applied. Not all SEND and RECV operation might be necessary depending on the stencil's shape. 92
6.5 Pseudocode representation of a generic two dimensional stencil ($\eta = 1$) with oversending method applied. Not all SEND and RECV operation might be necessary depending on the stencil's shape. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

# Chapter 1

# Introduction

Data Parallelism is a well known form of parallelization, which is based on partitioning the data across different parallel computing nodes. Its relevance will increase in the near future because of two joint factors. The first factor is the industry switch to parallel microprocessors. The second factor is the subtle and constant transition from control intensive computations to data intensive computations driven by the massive amount of information generated today[4]. One of the most challenging open problems is the optimization of such computations in order to ensure the desired performance and hopefully, portability of performance, among the diverse set of future computer architectures. In order to investigate the difficulties of optimization, we selected a subset of all data parallel computations known as structured grid computations[4]. As we will see in the next section, structured grids are central for many simulation codes; therefore, they were studied extensively in the high performance computing community. Although they are relatively simple computations in terms of structure, their optimization is a difficult process and many different solutions were proposed in literature. Because of these complications, no standard libraries or frameworks exist for structured grid computations while almost standard solutions exist for dense linear algebra (ATLAS[43]), sparse linear algebra (OSKI[42]) and spectral (FFTW[19]) computations.

## 1.1 Structured Grid Computations

Partial differential equation (PDE) solvers constitute a large portion of all scientific applications. PDE solvers are at the heart of simulation codes for many areas, from physical phenomena[27] to financial market stock pricing[22]. In order to solve partial differential equations, one possible approach is the

finite difference method[28]. In this method, the differential operators are approximated by truncated Taylor expansions of their derivatives and the continuous domain where PDEs are defined is discretized. This results in a very sparse matrix equation with predictable entries that can be solved efficiently by using iterative methods. Iterative methods find the solution by repeatedly updating an initial guess until numerical convergence is achieved. Each point is updated with a weighted contributions of its neighbours. Consider as an example a two dimensional poisson equation: $\Delta u = f$ where $f$ is a known function. After applying the finite difference method its solution can be found by updating an initial guess with the Jacobi iterative method until the error is below a given threshold. The pseudocode of a single jacobi update is shown in Listing 1.1.

```
for i = 1 .. n
    for j = 1 .. n
        u_new[i][j] = (f[i][j] - u_old[i−1][j] - u_old[i+1][j] - u_old[i][j+1] -
            u_old[i][j+1])/4
```

**Listing 1.1:** Pseudocode of a Jacobi update for the resolution of a two dimensional poisson equation.

These solvers can be easily implemented in a data parallel fashion where different workers update different portions of the result for the next time step.

Given the regularity of these computations, they are called **structured grid** computations. Computations in this class range from very simple Jacobi iterations (Listing 1.1) to multigrid[9] or adaptive mesh refinement methods[7]. We chose structured grids computations as a benchmark for optimization methods for data parallel computations because of two factors:

1. They are fundamental for a wide set of simulation codes in multiple disciplines.

2. Although they are very simple, these computations usually achieve a fraction of their theoretical peak performance on modern architectures[25].

Therefore, the optimization of these computations has been the subject of much investigation. In particular, previous research has shown that memory transfers constitute the main bottleneck for this class of computations. Therefore, research focused primarily on tiling optimizations that attempt to reduce the memory traffic by increasing the temporal locality (reuse) of the computation. Tiling optimizations constitute a subset of the class of **loop reordering** transformations. Loop reordering transformations modify the

order in which updates are executed. Assuming a row order storage of data, we could reorder the computation in Listing 1.1 in the following way:

```
for jj = 1,B,n
    for i = 1 .. n
        for j = jj .. jj + B − 1
        u_new[i][j] = (f[i][j] − u_old[i−1][j] − u_old[i+1][j] − u_old[i][j+1] −
            u_old[i][j+1])/4
```

**Listing 1.2:** Pseudocode of reordered Jacobi update for the resolution of a two dimensional poisson equation.

Since the updates of the Jacobi method can be performed in any order, the transformed program will produce the same result. Moreover, by selecting a good parameter $B$ we can reduce the number of cache miss. A formalization of this method, together with a rigorous analysis, is presented in Chapter 5.

Initial research focused on tiling only the spatial dimension[34] (Listing 1.2). However, reuse in structured grids computations is relatively limited compared to classic methods for dense linear algebra. In fact, further research has shown that tiling also in the time dimension (time skewing) gives the best results[24]. A comprehensive analysis of different tiling techniques was presented in [14].

The effectiveness of tiling is strictly related to an optimal selection of the tile sizes. Sophisticated models were introduced in order to solve this problem[25]. However, peculiarities of modern architectures makes formal modelling of this problem difficult. In fact, research has focused on auto-tuning techniques in order to select an optimal set of parameters without having to derive a formal model[15, 13]. Another possible approach is to use cache oblivious algorithms. A cache oblivious time skewed algorithm for d-dimensional structured grids was presented in [20, 21]. The idea behind cache oblivious algorithms is to recursively divide the working domain so that almost optimal tile sizes are selected for a given level of recursion thus, making the algorithm oblivious to the problem size or the specifications of the cache hierarchy. Recently, a compiler for stencil computations using a cache oblivious approach was presented[38].

Some optimizations methods, specific to structured grid computations were recently developed in our HPC lab[29, 31]. Unlike other methods present in literature, these methods attempt to reduce the overhead of communications between processors by slightly modifying the computation. As a example consider the code in Listing 1.1 which can be graphically represented as in Figure 1.1a. The $\mathcal{Q}$ trasformation presented in [31] modifies the functional dependencies of the computation and produces the shape in Figure 1.1b.

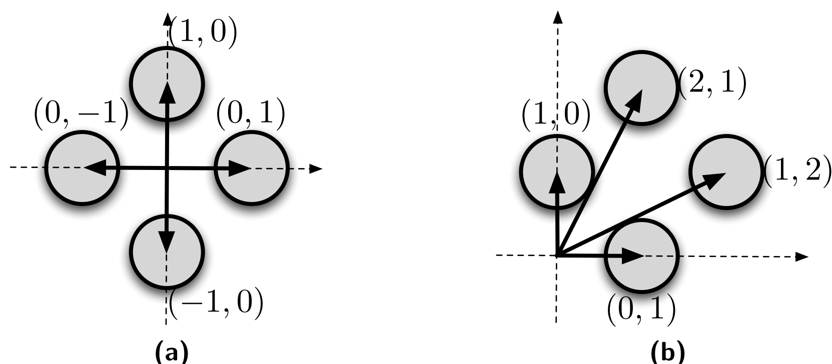


**Figure 1.1:** Graphical representations of the functional dependencies of respectively *Jacobi* and *Jacobi* with applied $\mathcal{Q}$ trasformation stencils

These optimizations are presented in Chapter 4. Unfortunately, research on specific methods for parallel stencil codes on distributed memory machines is very limited[35]. Some methods were presented in order to reduce the number of communications among parallel executors[17, 33]. Overall, the work of Meneghin[31] is the most comprehensive in this specific context. In particular, it presents transformations for structured grids that ensure a minimal number of communications and occupation of memory among all techniques found in literature. We present them, together with standard methods found in literature, in a structured model extended from [31]. A cost model is used to derive theoretically the performance gain of different optimizations. In particular, for tiling optimizations, we show that it is possible to reduce the number of memory accesses to the order of the theoretical lower bound. Moreover, their interaction with transformations presented in [31] is analyzed in detail and solutions are presented in order to maintain the benefits of both classes of optimizations.

**Methodology** We claim that a structured model is necessary to study the interaction of different optimizations. Optimizations are often presented in a non-structured way where benefits are elucidated in an intuitive way and then experimental results are used to prove the optimization's suitability. Instead, we selected a simple cost model derived from [40]. The cost for a program is the maximum between the cost of communication and computation $T_C = max(T_{calc}, T_{comm})$. We are assuming that we can overlap completely computation and communication. $T_{calc}$ is the maximum of the number of memory access among different processes in an external memory model. $T_{comm}$ is derived by associating a fixed cost to every communication $T_{setup}$ and a variable cost that depends on the message size, $T_{trasm}$. So a com-

munication of $m$ elements between two workers will have an associated cost of $T_{setup} + m * T_{trasm}$. By using such model, it is possible to show the performance gain of optimizations without meddling with peculiarities of modern architectures. Moreover, we are able to study the interaction of the different methods, which were analyzed individually.

## 1.2 Thesis Outline

Data parallel computations are presented in Chapter 2. Structured grids are examined in detail in the same chapter. Their relevance is demonstrated with real world examples and typical properties of a structured grid computations are described. The structural model that we used to present different optimizations is introduced in Chapter 3. This model is general in order to represent any framework for a class of computations. Moreover, a specific instance of the model for structured grids computations, along with a concretion and optimization examples, are illustrated. Optimizations for a parallel implementation of structured grids are introduced in Chapter 4. Most of the methods presented were developed by Meneghin in his doctoral thesis[31]. Chapter 5 briefly explains optimizations implemented by modern compilers. Among all possible optimizations, the class of loop reordering transformations is analyzed in great detail because most optimizations relevant to structured grids computations found in literature belongs to this class ( tiling optimizations belongs to this class). Specific optimization examples are presented for structured grids and their impact on performance is validated using the model introduced in Chapter 3. Our analysis concludes in Chapter 6 where we study the combination of optimization methods for parallel execution (Chapter 4) and loop reordering (Chapter 5).

# Chapter 2

# Data Parallel with Stencil Computations

In this chapter, we introduce data parallel with stencil computations. Data parallel is a well known parallel programming paradigm which is based on the replication of functions and partitioning of data [40, 41]. Data parallel with stencil computations (we will from now on refer to them as stencil computations for simplicity) constitutes a subset of data parallel computations, which have functional dependencies between executors. In fact, the term stencil indicates the communication pattern among workers, which is necessary to perform the computation.

## 2.1 Map Computations

Firstly, we analyze the simplest class of data parallel computations known as *map*. In such paradigm, data is partitioned among workers (or executors) which all perform the same function $\mathcal{F}$ on the partitioned data. This is usually repeated for a series of **time steps** ( four time steps for the map computation presented in Listing 2.1). As an example, we consider an image processing application which reduces luminosity of every pixel of a grayscale image represented as a $M$x$M$ matrix. Pseudo-code of a generic *map* computation is presented in Figure 2.1.

**for** $s = 0 .. 4$
    **forall** i,j = 0 .. M−1 **do** $\mathcal{A}_{i,j}^{s+1} = \mathcal{F}(\mathcal{A}_{i,j}^{s})$

**Listing 2.1:** Pseudocode of a generic map computation. The computation is defined over 4 time steps.

At the highest level of abstraction, there is no reference either to partitions or to data structures. Conceptually $s+1$ matrices exist and the update of every element is performed in parallel[1]. At an underlying (concrete) level, there is a fixed number of real processors and concrete data structures. Therefore, multiple elements will be stored in real data structures and assigned to a single real processor (executor), which will sequentially perform the given function on every point of its partition[2].

```
for s = 0 .. 4
    for i = 0 .. M−1
        for j = 0 .. M−1
            B[i,j] = F(A[i,j])
    for i = 0 .. M−1
        for j = 0 .. M−1
            A[i,j] = B[i,j]
```

**Listing 2.2:** Pseudocode of the implementation of the map computation presented in Listing 2.1. Two real data structures are used and elements are updated sequentially.

We need to formally define the relationship between ownership and update of an element:

**Definition 2.1** (Owner Computes Rule). *The processor that owns the left-hand side element will perform the calculation.*

In other words, this rule states that the owner of the $\mathcal{A}_{i,j}$ element is the only one allowed to modify it. After the data distribution phase, each process takes ownership of the distributed data that it is storing; this means that it is the only one that can modify that data. Consequences of the owner computes rule are that the elements in the right hand side have to be sent to the worker performing the update. Notice that this is not the only possibility, because the computation may take place on a different worker and the final result could be sent to the owner of the left hand side for assignment. However, since every element $\mathcal{A}^{s+1}_{i,j}$ needs only its predecessor $\mathcal{A}^{s}_{i,j}$, no interaction is necessary between executors. Therefore, map computations can be easily translated from a high level representation to a concrete implementation; elements can be grouped together either statically (at compile time) or dynamically (at runtime) in any possible combination without affecting correctness.

[1] Conceptually, this is equivalent to the Virtual Processors presented in [39]

[2] We will use the notation $\mathcal{A}_{i,j}$ to indicate an element at the highest level of abstraction and A[i,j] to indicate a concrete element stored in a data structure.

## 2.2 Data Parallel with Stencil Computations

As anticipated at the beginning of this chapter, stencil computations require a pattern of communications between workers. We need to formally define this concept.

**Definition 2.2** (Stencil computation). *A stencil computation is a data parallel computation where functional dependencies exist between different elements.*

**Definition 2.3** (Functional Dependency). *A functional dependency is a relationship between two elements: $i \rightarrow j$ meaning that element $j$ need elements $i$ in order to be updated.*

Notice that functional dependencies for map computations are always from an element to itself (Listing 2.1). To introduce stencil computations, we consider the Laplace equation solver (Figure 2.3).

```
for s = 0.. N
    forall i,j = 0 .. M−1 do
        A^{s+1}_{i,j} = F(A^s_{i,j}, A^s_{i−1,j}, A^s_{i+1,j}, A^s_{i,j−1}, A^s_{i,j+1})
```

**Listing 2.3:** Pseudocode of Laplacian Operator.

From the pseudo-code is evident that $A^{s+1}_{i,j}$ has the following functional dependencies: $A^s_{i,j+1}$, $A^s_{i+1,j}$, $A^s_{i-1,j}$ ,$A^s_{i,j-1}$. The set of all functional dependencies of an element is called the **shape** of the stencil. The element that is updated is called the **application point**. The function performed on the input data is the stencil **kernel**. When the $\mathcal{A}_{i,j}$ elements are assigned to a real partition the owner computes rule is applied; therefore, a pattern of interaction between workers is derived.

### 2.2.1 Classification of Stencil Computations

At this point, a taxonomy of data parallel computations is at this point necessary. Firstly, we classify these computations depending on the presence or absence of functional dependencies. If no functional dependencies are present between different elements, then we turn to map computations (Section 2.1). If there are functional dependencies, we define the following four classes based on the properties of functional dependencies of the computation[3]:

[3]This classification was introduced in a more mathematical rigorous way in [31] as the $\mathcal{HUA}$ model.

- Fixed vs Variable: In a Fixed stencil, the functional dependencies do not change over different time steps. Otherwise the stencil computation is Variable.

- Dynamic vs Static: In a Static stencil, the functional dependencies over different time steps can be derived at compile time. Otherwise, if the dependencies are based on the value of the elements of the domain, the stencil is dynamic.

The class of fixed static computations is the simplest to analyze. Many problems in this class happen to have regular functional dependencies[4], that are fixed not only with respect to the time steps, but also with respect to the spatial position (Notice that the Laplace equation solver presented in Figure 2.3 belongs to this class).

Static fixed stencil computations can be found at the heart of Partial Differential Equations (PDE) solvers. PDE solvers are fundamental for almost any simulation codes, from the heat equation [28] to stock market pricing[22] and computer vision [23]. Stencil codes are also used in BioInformatics algorithms for RNA prediction[3, 11] usually on monodimensional arrays while PDE solvers are usually performed on multidimensional grids [7, 28, 9, 30, 44, 6, 18]. All these computations have a pattern of interaction which is limited to neighbouring elements. Moreover, they are fixed with respect to time and space; therefore, they are usually cited as **Structured Grids** computations.

## 2.3 Structured Grids in the Real World

In this section, we present how structured grids computations arise from the finite difference method for partial differential equations. Although this is outside the actual scope of this thesis, we wanted to explain why such class is so important and how it arises from real world problems. Otherwise, it would seem that we are studying a synthetic benchmark.
The process of finding an approximate solution to PDE via the Finite Difference Method can be summarized as follows.

1. For a given PDE, we firstly approximate differential operators using a **finite difference approximation** of the partial derivatives at some point x.

[4]With the exception of the borders of the domain where boundary conditions arise.

2. PDEs are defined over continuous domains. Therefore, we discretize the domain of the PDE by dividing it into small subintervals. In each subinterval $i$, we apply the finite difference approximation of Step 1 and thus, arrive at a linear system of difference equations.

3. We solve the linear systems with iterative methods such as Jacobi or Gauss Seidel.

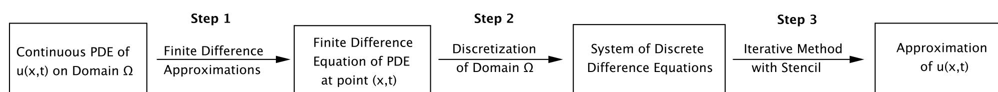


**Figure 2.1:** The process of finding an approximate solution to the PDE via the Finite Difference Method.

### 2.3.1 Finite Difference Method

The finite difference method is based on local approximations of the partial derivatives in a Partial Differential Equation, which are derived from low order Taylor series expansion [36]. By the Taylor series expansion of a function $u$ in the neighborhood of $x$, we have that

$$u(x+h) = u(x) + h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} + \frac{h^3}{6}\frac{d^3u}{dx^3} + \frac{h^4}{24}\frac{d^4u}{dx^4} + O(h^5) \tag{2.1}$$

$$u(x-h) = u(x) - h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} - \frac{h^3}{6}\frac{d^3u}{dx^3} + \frac{h^4}{24}\frac{d^4u}{dx^4} + O(h^5) \tag{2.2}$$

where $h$ is a value close to zero such that $x+h$ and $x-h$ are in the neighborhood of $x$. If we add Equation 2.1 and 2.2 and divide by $h^2$, we arrive at the following approximation of the second order derivative.

$$\frac{d^2u(x)}{dx^2} = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + O(h^2) \tag{2.3}$$

The above expression is called the centered difference approximation of the second order derivative. The dependence of this derivative on the values of $u$ at the points involved in the approximation is represented by a **stencil** [36]. The figure below shows the three-point stencil of the centered difference approximation to the second order derivative.

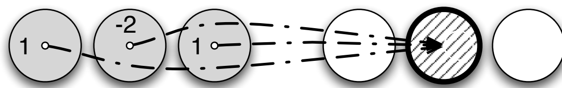


**Figure 2.2:** Three-point stencil of centered difference approximation to the second order derivative. This stencil represents Equation 2.3.

### 2.3.2 Discretization of Partial Differential Equations

Consider now a very simple differential equation:

$$-u^{''}(x) = f(x) \text{ for } x \in (0,1) \qquad (2.4)$$
$$u(0) = u(1) = 0$$

The value of $u^{''}(x)$ is known and we want to compute an approximation of $u(x)$. Boundaries conditions are: $u(0) = u(1) = 0$. If we want to approximate the solution of a PDE over its domain of definition, we discretize the domain by dividing it into smaller regions.
The interval [0,1] of equation 2.4 is divided into $n+1$ subintervals of uniform spacing $h = 1/(n+1)$. The discrete set of points that divide the interval are:

$$x_i = i \cdot h \text{ where } i = 0, ..., n+1 \qquad (2.5)$$

This set of points, derived by the discretization of the real continous domain is called the mesh[36].

By applying equation 2.3 to equation 2.4, we have that

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i \qquad (2.6)$$

where $u_i$ is the numerical approximation of $u(x_i)$ and $f_i \equiv f(x_i)$. Note that for $i = 1$ and $i = n$, the equation will involve $u_0$ and $u_{n+1}$ which are known quantities, both equal to zero in this case. Thus, we have a set of $n$ linear equations which we represent by the following matrix equation $A\mathbf{u} = \mathbf{f}$ .

$$\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n + u_{n+1} \end{pmatrix} \qquad (2.7)$$

This linear system can be solved by direct methods such as LU or QR factorization. However, these methods require to explicitly modify the matrix

2.7 which is bigger than the discretization domain. Moreover, this matrix is very sparse and very regular; storing it explicitly is inefficient and standard factorization methods would destroy its structure. Therefore, iterative methods are the best solution for these kind of problems.

### 2.3.3 Iterative Methods to Solve PDE

Now, we elucidate three iterative methods in order to solve the linear system, $A\mathbf{u} = \mathbf{f}$. These three iterative methods, applied to a sparse and regular matrix, such as the one in Equation 2.7 will produce stencil computations.

#### Jacobi Method

The Jacobi method is the simplest approach. The $i$th equation of a system of $n$ linear equations is:

$$\Sigma_{j=1}^{n} a_{i,j} u_j = f_i \tag{2.8}$$

where $u_j$ is the $j$th entry of the vector $\mathbf{u}$. The idea (beyond the Jacobi method) is to solve independently for every vector component $u_i$ while assuming the other entries of $\mathbf{u}$ remain fixed. This results in the iteration:

$$u_i^{(k)} = \frac{f_i - \Sigma_{j \neq i} a_{i,j} u_j^{(k-1)}}{a_{ii}} \tag{2.9}$$

Since this method treats each equation independently, all the $u_i^{(k)}$ components can be computed in parallel. Moreover, notice how it is not necessary to explicitly store the matrix A (Equation 2.7). The implementation of the Jacobi Method is presented in Listing 2.4.

```
for t = 1..T
    forall i = 1 .. n
        u_new[i] = (f[i] − u_old[i−1] − u_old[i+1])/2
    forall i = 1 .. n
        u_old[i] = u_new[i]
```

**Listing 2.4:** Pseudocode of a Jacobi Iteration for matrix equation 2.7.

Notice that the element f[i] does not belong to the shape of the stencil since it is a constant. Since the array f is never updated its elements are conceptually shared among all the workers.

### Gauss Seidel method

A faster version of the Jacobi method is the Gauss-Seidel method. The idea is to reuse values updated in the current timestep. More precisely,

$$u_i^{(k)} = \frac{f_i - \Sigma_{j<i} u_j^{(k)} - \Sigma_{j>i} u_j^{(k-1)}}{a_{ii}} \tag{2.10}$$

Since each component of the new iterate depends upon **all** previously computed components, the updates cannot be done simultaneously as in the Jacobi method. On the other hand, the Gauss-Seidel Iterations uses less memory and it is faster to converge. Its implementation for linear system (2.7) is presented in Figure 2.5:

```
for t = 1 .. T
    for i = 1 .. n
        u[i]=(f[i] - u[i+1] - u[i-1] )/2
```

**Listing 2.5:** Pseudocode of a Gauss Seidel Iteration for matrix equation 2.7.

The Gauss Seidel iteration depends upon the order in which the equations (2.8) are examined. In particular, the Gauss Seidel iteration updates the elements of array $u$ in a linear scan. If this ordering is changed, the components of the new iterates will also change. Different orderings of the Gauss Seidel iteration are cited in literature as Multicolor Orderings and are usually employed to find a **compromise** between the Jacobi and the Gauss Seidel iterative scheme.

### Red and Black Gauss Seidel

The Red and Black Gauss-Seidel method changes the ordering of the standard Gauss-Seidel method. The standard Gauss-Seidel method follows the natural ordering while the Red and Black Gauss Seidel method follows an ordering that can be represented by a checker board pattern made of red and black dots. More precisely, in a monodimensional mesh, a gridpoint $i$ is colored red if it is even and is colored black otherwise. The method updates the solution in two passes: first the red dots are calculated from the black dots and then the black dots are calculated from the new red dots. This method not only has faster convergence than Gauss-Seidel, but also allows parallel updates since there is no interdependences within a single sweep.

The implementation of the Red and Black method for the matrix (2.7) is presented in Figure 2.6.

```
for t = 1 .. T
    forall i = 1 .. n
        // red sweep
        if mod(i,2) == 0
            b[i]=  (f[i] - a[i-1] - a[i+1])/2
    forall i = 1 .. n
        // black sweep
        if mod(i,2) == 1
            a[i]=  (f[i] - b[i-1] - b[i+1])/2
```

**Listing 2.6:** Pseudocode of a Red Black Gauss Seidel Iteration for matrix equation 2.7. Notice that the $u$ vector is replicated in two vector a and b which are updated in a alternated fashion.

### 2.3.4 Heat Equation

We are now going to present the two dimensional heat equation as an example of a more complex, real world application of structured grids computations. The heat equation describes the distribution of heat (variation of temperature) in a given region over time. Consider a flat surface with a given distribution of temperature. It can be discretized as a two dimensional array $u$ where $u[i][j]$ contains the discretized value of temperature in the spatial domain (Figure 2.3).

The heat equation states:

$$\frac{\partial u}{\partial t} = \alpha \Delta u \tag{2.11}$$

where $\Delta$ is the laplacian operator and $\alpha$ is a positive constant related to the physical properties of the surface material (thermal diffusivity). In our two dimensional mesh the laplacian operator corresponds to the following:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2.12}$$

We can utilize the two variable version of the finite difference approximation shown in Equation (2.3) in order to obtain the discrete laplacian operator:

$$\Delta u \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2} + \frac{u(x,y+k) - 2u(x,y) + u(x,y-k)}{k^2} \tag{2.13}$$

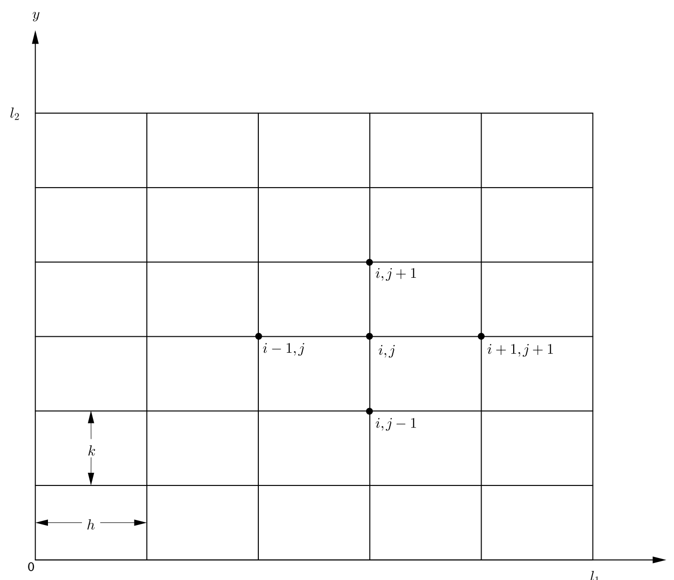


**Figure 2.3:** Discretization of temperature over a flat surface. The domain is divided into rectangles of width $h$ and height $k$.

If we let $h = k$, i.e., discretization has the same precision over the two axis, we have the following simplification:

$$\Delta u(x) \approx \frac{1}{h^2}[u(x+h,y)+u(x-h,y)+u(x,y+h)+u(x,y-h)-4u(x,y)] \quad (2.14)$$

The above equation is called the five-point centered approximation to the Laplacian[36] and its corresponding stencil is shown in Figure 2.4.

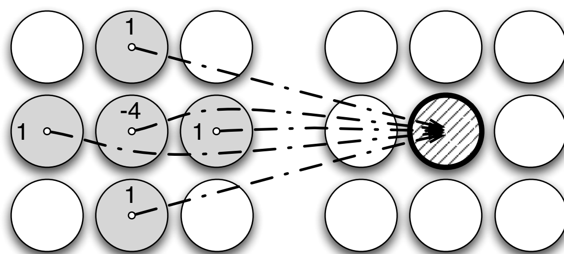


**Figure 2.4:** Five-point stencil for the centered difference approximation to Laplacean.

Consider now the case that $\frac{\partial u}{\partial t}$ is known over domain $\Omega$ and boundary values are known on the domain boundary $\tau$. We want to compute the

distribution of temperature $u$ at a given time instant $t_0$. By substituting equation (2.13) in (2.11) we obtain the following Discrete Poisson Equation:

$$\begin{gathered} u(x+h,y)+u(x-h,y)+u(x,y+h) \\ +u(x,y-h)-4u(x,y)=h^2 f(x,y) \text{ on } \Omega \\ u=0 \text{ on } \tau \end{gathered} \tag{2.15}$$

where function $f(x,y)$ represents the known values of $\frac{\partial u}{\partial t}$ over domain $\Omega$. Similarly to Section 2.3.2, we can derive a matrix equation from this system of equations. In order to do so, we take the lexicographical column ordering of $\mathbf{u}$, meaning

$$\mathbf{u}=((u_{1,1},u_{2,1}...,u_{n,1}),(u_{1,2},u_{2,2},...u_{n,2}),...,(u_{1,n}u_{2,n}...u_{n,n})). \tag{2.16}$$

and we will indicate the various column as:

$$\begin{aligned} u_1 =&(u_{1,1},u_{2,1}...,u_{n,1}) \\ u_2 =&(u_{1,2},u_{2,2}...,u_{n,2}) \\ &\vdots \\ u_n =&(u_{1,n},u_{2,n}...,u_{n,n}) \end{aligned} \tag{2.17}$$

We obtain the follow matrix equation:

$$\begin{pmatrix} T & -I & 0 & \dots & 0 \\ -I & T & -I & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -I & T & -I \\ 0 & \dots & 0 & -I & T \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} \tag{2.18}$$

where I is the $n$x$n$ identity matrix and T is the $n$x$n$ tridiagonal matrix:

$$\begin{pmatrix} 4 & -1 & 0 & \dots & 0 \\ -1 & 4 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 4 & -1 \\ 0 & \dots & 0 & -1 & 4 \end{pmatrix} \tag{2.19}$$

Applying the Jacobi method to the matrix equation (2.18) we obtain the following code:

```
for t = 1..T
    forall i = 1 .. n
        forall j = 1 .. n
            u_new[i][j] = (f[i][j] - u_old[i-1][j] - u_old[i+1][j] - u_old[i][j+1] -
                u_old[i][j+1])/4
    forall i = 1 .. n
        u_old[i][j] = u_new[i][j]
```

**Listing 2.7:** Pseudocode of Jacobi method for the linear system 2.18.

### Simulation

The previous example assumed that the derivative of temperature at a given time was known. The final value computed is the distribution of temperature at a given time instant (steady state). Consider the more interesting case where we know the distribution of temperature at a given time $t_0$ and by using the heat equation (2.11) we want to simulate how the system evolves in time. In order to do so we introduce another approximation by discretizing $\frac{\partial u}{\partial t}$ using the fist order forward time difference:

$$\frac{\partial u}{\partial t} \approx \frac{u(t+k) - u(t)}{k} \tag{2.20}$$

We then apply this approximation (2.20) (discretized with respect of time) to the heat equation (2.11) obtaining:

$$\frac{u(t+1) - u(t)}{k} = \alpha \Delta u \tag{2.21}$$

then we discretize over the spatial domain using the discrete laplacian operator (2.13) obtaining.

$$\begin{aligned} \frac{u(t+1,x,y) - u(t,x,y)}{k} = \alpha \frac{1}{h^2}[u(t,x+h,y) + u(t,x-h,y) \\ +u(t,x,y+h) + u(t,x,y-h) - 4u(t,x,y)] \end{aligned} \tag{2.22}$$

We let $r = \alpha k/h^2$ and rewrite equation 2.22:

$$\begin{aligned} u(t+1,x,y) = u(t,x,y) + r * (u(t,x+1,y), u(t,x-1,y), \\ u(t,x,y+1), u(t,x,y-1) - 4u(t,x,y)) \end{aligned} \tag{2.23}$$

which is the five point stencil presented in Figure 2.4, only with different coefficients.

The importance of Structured Grid computations should be clear now. Partial differential equations are fundamental in order to model almost any physical phenomena. Moreover, PDE solvers based on the finite difference method require the execution of a structured grid computation. The two dimensional heat equation presented here is probably one of the simpler problems to solve. Other models, e.g., hydrodynamic models involve multiple discretized domains of higher dimensionality[13]. Moreover, it should be clear that the size of a stencil shape and the size of the discretized domain are both a function of the required precision. Therefore, structured grids, although a fairly narrow class of data parallel computations, are fundamental for many simulation codes and exhibit remarkable differences depending on the modelled phenomena.

## 2.4 Properties of Structured Grid Computations

Now that we have introduced some simple real world examples of structured grids computations, we can further explore their properties. We have seen that they perform sweeps[5] over multidimensional data structures. The size and number of spatial dimensions of these data structures are related to the type of problems and the required precision. For a "typical" stencil computation, the following usually applies:

1. The size of accessed (and updated) data structures exceed the capacity of available data caches.

2. The number of shape points is small, e.g. five points for the laplacian operator.

A consequence of 1 is that elements have to be fetched from the memory multiple times during a time step. Moreover, because of 2, the number of floating point operations per point is relatively low, which suggests that transfers of data from the memory are the limiting factor for performances. Therefore, most of the research on optimizations for stencil computations has focused on the full exploitation of the memory hierarchy in order to avoid stalling caused by memory transfers; most notably by using tiling optimizations[34]. Tiling

[5] With the term sweeps we denote the update of a data structure, touching every point. The ordering is not relevant.

optimizations modify the order in which updates are performed in order to reduce the distance between accesses to the same location in memory. This effectively reduces the number of cache miss and increases the computation performance. Tiling is not only utilized for structured grids computation, but also in scientific computing applications[6]. Its efficiency is strictly dependent on:

1. The computation type.
2. The data size.
3. The underlying concrete machine where the program will be executed.

We will now concentrate on the first point: the peculiarities of structured grids computations that may affect optimizations techniques. We will expand on the latter two topics in the following chapters.

Now we can analyze the three methods introduced in the previous section. The Jacobi, Gauss-Seidel and Red & Black examples presented in Section 2.3 are all possible solutions to the same computational problem. The Jacobi method is the most promising in terms of performances because:

- There are no functional dependencies between elements inside the loop. Therefore the update of different elements can be performed in **parallel**. On the other hand, the Gauss-Seidel method has dependencies between elements during the same time step, e.g., $\mathcal{A}^{s+1}_{i,j}$ has the following functional dependencies $\mathcal{A}^{s}_{i,j+1}$ , $\mathcal{A}^{s}_{i+1,j}$ , $\mathcal{A}^{s+1}_{i-1,j}$ , $\mathcal{A}^{s+1}_{i,j-1}$ .
- During a time step, Jacobi performs a single sweep over all elements while the Red & Black version requires two sweeps.

However, it is clear that the result obtained by these three algorithms is different after a timestep. Therefore, we have to consider their numerical properties in order to determine which one performs best.

### 2.4.1 Convergence

While numerical convergence properties of iterative methods are outside the scope of this thesis, they are fundamental for achieving the best performance in real world applications. The Jacobi method is without doubt the most

[6] Tiling can be beneficial for any computation accessing data structures (bigger than the size of available data caches) multiple times. These conditions usually arise in scientific computing applications.

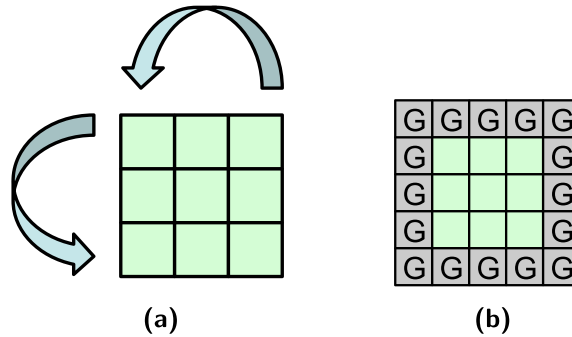


**Figure 2.5:** Graphical depiction of periodic boundary conditions( Figure 2.5a ) and constant boundary conditions ( Figure 2.5b ) of a $3x3$ mesh.

efficient, but it is very often the slowest to converge to a solution [13]. Moreover, it is not possible to determine a priori which of the three methods will converge the fastest although heuristics exist. As an example, for problems arising from finite difference approximations, the Red & Black algorithm gives the best results [16].

Consider now a more subtle problem related to numerical convergence. Numerical convergence has to be tested periodically, i.e., at the end of each time step ( or a sequence of time steps ). These tests obviously take some time therefore increasing the total running time of the computations. Even if their contribution can be negligible, they modify the computation structure, reducing its regularity and therefore excluding some optimization techniques. Intuitively, we can understand that if the computation during different time steps is always the same, then we could take advantage of this property, e.g., by collapsing the execution of different time steps.

### 2.4.2 Boundary Conditions

In the examples in Section 2.3, we selected very simple boundary conditions. Since finite difference methods are used to discretize real world problems, complex boundary conditions may arise. Boundary conditions can be classified in a similar way as stencil computations:

- Constant Boundaries: Boundary values do not change over time. In this case, we have a halo around the boundaries of the grid where values are never updated. These elements are often called **ghost cells**.
- Variable Boundaries: Boundary values change over time. A common case is periodic boundary conditions where the domain is toroidal; therefore, functional dependencies that exceed the domain boundaries

are wrapped around the computation domain. As an example, consider a two dimensional domain, where the right neighbour of an element in the right border is the left most element lying in the same horizontal axis.

The case of ghost cells is the easiest to implement and does not affect performance. It will be necessary to have slightly bigger data structures in order to contain the ghost cells. Ghost cells will be only read and never updated; therefore, the computation performed for every element is the same. In the case of periodic boundary conditions, additional modulo and conditional operations should be introduced in order to distinguish boundary and external elements and to wrap around spatial coordinates on the toroidal domain. In some cases, the handling of boundary conditions can dominate the runtime of a stencil computation[38].

### 2.4.3 Stencil Coefficients

The matrix presented in equation (2.7) has a very regular structure. However, there are matrices where the non zero elements are not predictable. Thus, it will be necessary to store additional grids of coefficients, increasing the memory transfer per element update. This is true for lattice boltzmann methods (LBM) for computational hydrodynamics[13].

### 2.4.4 Notable Examples and Conclusions

Other notable examples of widely used stencil computation that are hard to classify are:

- Multigrid methods: Used as PDE solvers, they utilize a hierarchy of discretizations [9]. This approach is based on the fact that the convergence of the finite elements methods can be accelerated by varying the domain discretization over time.

- Adaptive Mesh Refinement (AMR) methods: In this class of computations, the discretization is varied over time depending on the actual phenomena that is modelled. As an example consider a physical model with collisions, where we want to dynamically increase the precision of the simulation in the part of the domain where the collision take place. These methods are used when a higher precision on all the domain at all time would make the simulation infeasible[7].

Notice that AMR methods, although very similar to structured grid computations, are dynamic variable stencil computations.

In conclusion, structured grids are a subset of data parallel computations which exhibit a pattern of interaction only between neighbour elements. They usually stem from numerical algorithms for the resolution of partial differential equations, which are in turn at the heart of most simulation codes.

# Chapter 3

# A Hierarchical Model for Optimization Techniques

In this chapter, we introduce a formal model for optimization methods. The process of **concretion** from a high level abstraction to an executable program for stencil computations was presented by Meneghin in his doctoral thesis[31]. Specifically, at the highest level, the stencil computation is expressed in a domain specific language while at the lowest level, an executable file is produced. This process of **concretion** is performed by a framework for a class of computations.

We claim that optimizations **conceptually** work at different levels of abstraction; therefore, they can be added to the concretion hierarchical model. Moreover, their effect can be estimated by defining a cost model at every different level of abstraction, thus providing a tool to analyze program transformations. The two biggest difficulties of optimization are:

1. Determining if a transformation is safe. Enforcing a semantic equivalence that is too strict will reduce the number of applicable transformations. "Unfortunately, compiler without high level knowledge about the application, can only preserve the semantics of the original algorithm" [2]. Therefore, it is necessary to gather domain specific knowledge about the application that has to be optimized and render this information usable by a compiler.

2. Determining if a transformation is beneficial for performances.

3. Determining the optimal configuration, i.e. the best parameters, for a large set of different optimizations which have complex interaction patterns.

The benefits of a framework are obvious for a programmer since it provides high level mechanisms to express a computation. These high level mechanisms are implemented in a hierarchical way; therefore, the framework is also extensible. On the other hand, the benefits of optimizations are determined with certainty only by executing the computation. However, it is fundamental to have a structured model to analyze optimizations and estimate their impact using a **cost model** of the real architecture. Therefore, we claim that by using a single hierarchical model we can represent and analyze both the implementation and the optimization of a class of computations.

## 3.1 Optimization in a Hierarchical Model

In this section, we introduce the notion of a framework for a generic class of computations[31]. We will assume that we have a domain specific language for this class of computations. By domain specific language, we refer to a language tailored explicitly to express a subset of all possible computations. Domain specific languages trade generality for expressiveness and for a high level of abstraction, thus resulting in higher productivity[32]. A computation expressed with such language features a high degree of abstraction with respect to an executable program described at firmware level. In fact, a high level language has the primary objective of preventing programmers from managing low level mechanisms. However, since the computation is executed on a real architecture, there is a gap between the level of abstraction desired by a programmer and the execution of the computation. A framework fills this gap between the high level representation and the firmware level executable by implementing the **concretion** process. This process consists in the transformation of a program described at the highest level into its equivalent version at the lowest level. The process is structured in a hierarchical way, meaning that it is defined on multiple steps. Every level $i$ in this process has a set of instruction (or mechanisms) $I_i$ and a specific language $L_i$. Moreover, every mechanism of a level $i$ is implemented at a lower level $j$ using mechanisms defined in the language $L_j$.

Now, we have to define a concretion step between two levels.

**Definition 3.1** (Concretion Function)**.** *Given two adjacent levels of abstraction, the concretion function $\mathbb{C}^i_j$ is a mapping from the higher level $i$ to the lower level $j$ which is always defined for every semantically correct program.*

We are implicitly stating that there always exists a **naive** way to translate a program from a high level of abstraction down to an executable one. If we only consider the definition of the concretion function and define the language

of every level, we have a model for the concretion of a class of computations. However, we enrich this model in order to account for optimizations. Intuitively, an optimization transforms an input program into an equivalent one, which performs better on a given architecture. Aho et al.[2] introduce the concept of optimization as "Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing". Although it is very clear, this is not a formal definition.

**Definition 3.2** (Program Transformation). *Let the space of well formed programs at level i be $L_i^*$. A program transformation is a function of $L_i^*$ in itself.*

Therefore, given a program $a \in L_i^*$, a transformed program $b \in L_i^*$ is produced. Notice that a transformation is defined on a specific level. It is obvious that a program transformation is relevant only if it preserves the *meaning* of the program; in that case, the transformation is **legal** (or safe). The explicit definition of legal transformation was debated extensively in the compiler research field. Bacon et al.[5] give many possible definitions of a legal transformation and conclude that the following is the most reasonable: "A transformation is legal if the original and the transformed programs produce exactly the same output for all identical executions". However, we will provide a more abstract definition that better suits our needs. More precisely, in order to formally define a legal transformation, we need to define the concept of equivalence of two programs.

**Definition 3.3** (Equivalence). *For every level i, there exists a formal semantic associated to every well formed program expressed in $L_i$. Two programs a and b will be equivalent at level i if their semantic is equal: $[\![a]\!]_i = [\![b]\!]_i$. Therefore, the space of well formed programs at level i will be partitioned into disjoint groups of equivalent programs.*

Now we can define:

**Definition 3.4** (Legal Transformation). *A transformation f defined at level i is legal if $\forall a \in L_i^* \;\; f(a) = b \Longrightarrow [\![a]\!]_i = [\![b]\!]_i$*

In other words, a transformation is legal if for every input program, it produces an equivalent one. There could be an infinite number of legal transformation which can be trivially obtained from a given input program $a$. As an example, consider a sample piece of code defined in a general purpose language:

```
B[i,j] = A[i,j] + 1
```

An equivalent program is:

```
B[i,j] = A[i,j] +3 -2
```

The semantic of the right hand side of both statements is obviously the same and countless other legal transformation can be made in a similar trivial way. Until now, we have only defined a program transformation and a legal transformation. An optimization is a legal transformation which introduces some benefits in terms of performance. Performance is evident when executing the application on a real architecture. Therefore, we need to formally introduce the concept of performance in our model. In order to do so, we assign a cost model to every level.

**Definition 3.5** (Cost Model)**.** *Given the space $L_i^*$ of well formed programs at level $i$, cost model $\mathcal{C}_i$ for level $i$ is a function of $L_i^*$ to $\mathbb{R}^+$.*

This is a very general definition because it states that for every level, there exists a function that associates every program to its cost, which cannot be negative. Since our model is hierarchical, the cost model should also be hierarchical. Therefore, the cost model at the highest level features a high level of abstraction and it does not consider peculiarities of specific architectures. On the other hand, the cost model of lower levels should be closer to the actual execution of the application on the given architecture. Next, we formally define an optimization.

**Definition 3.6** (Optimization)**.** *Given a hierarchical model which corresponds to the execution of a computation on a target architecture, a legal transformation $f$, defined at level $i$, is an optimization if $\exists P_1 \in L_i^*.\ f(P_1) = P_2$ and at an underlying level $j \leq i$, the cost of the transformed program $C_j(P_2)$ is less than the cost of the original program $C_j(P_1)$*[1]*.*

In other words, an optimization for a program corresponds to the application of a legal transformation such that the cost of the transformed program is lower at an underlying (or at the same) level of abstraction. We are not imposing that the cost should be lower at the same level of abstraction where the transformation is applied. This is because the cost model at an higher level might be too abstract to capture the advantage of the optimization ( this is the case of many optimizations described in Chapter 4). Notice that we are also not implying that the cost of the transformed will be lower at the firmware level, where it corresponds to the actual completion time of the computation. Otherwise, we would have that an optimization, in order

[1] When considering the cost of a program $C_j(P_1)$ where $P_1$ is defined at a higher level of abstraction $i > j$ we are indeed considering $C_j(\mathbb{C}_j^i(P_1))$

to be labelled so, has to increase performances on any possible real architecture which is generally infeasible. The extreme consequence is that we could only define optimizations at the firmware level (therefore only for a specific architectures). Instead, we want to define and analyze optimizations using abstractions that are oblivious to architecture peculiarities. It would be compiler's responsibility to decide when to apply a given optimization after knowing the real architecture where the program has to be executed.

This definition of optimization in a hierarchical model, although fairly abstract, expounds many real practices in compilation techniques. The equivalence classes of a program at a given level correspond to the space of possible optimizations (optimization space). A heuristic serves to prune this space and eliminate choices that are known to be inefficient. After space pruning, the compiler will make a choice among all candidate programs. Notice that this type of optimization problems are usually NP, e.g., selecting the best partitioning[15]. Defining the process on multiple levels reflects the fact that optimizations works at different levels of abstraction. Therefore, an optimization working at higher level of abstraction, where the semantics of the language is as close as possible to the intended semantics of the program, will manage different mechanisms with respect to a peephole optimization[2] working on the object code of the application.

## 3.2 The Reference Architecture

In this section, we introduce the notion of a framework for the class of stencil computations developed in[31]. Similarly to Section 3.1, we will assume that we have a domain specific language[32] for the class of stencil computations. We will use these computations as an example throughout all this thesis, but note that the formalization presented in the previous section is valid for any generic computation. A stencil representation, defined with a domain specific language, features a high degree of abstraction with respect to an executable program described at firmware level. The model presented in this section will give an idea of the whole process of concretion. In [31], the concretion process was defined on four separate levels:

1. Functional Dependencies.

2. Partition Level.

[2]In compiler theory, peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code, e.g., removing duplicate instructions.

3. Concurrent Level.

4. Firmware Level.

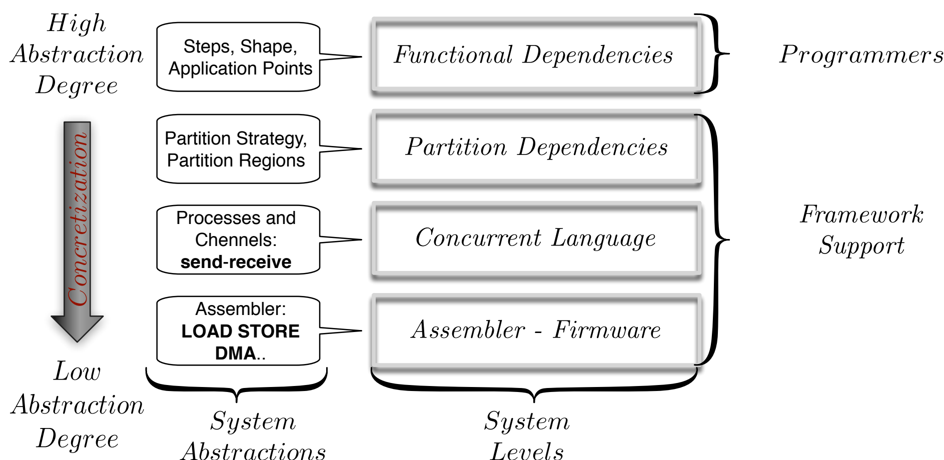


**Figure 3.1:** Framework for stencil computations as presented in [31].

I The stencil computation is defined at the highest level of **Functional Dependencies**.

II The stencil computation, defined at the functional dependencies level is then translated in the underlying **Partition Level** where functional dependencies become dependencies among partitions.

III At the **Concurrent Level**, the partitions are associated to different processes and the dependencies between partition become communications among processes. Therefore, we assume that a message passage paradigm is used.

IV Finally, we compile an executable program by means of a standard compiler.

We briefly explain these steps showing the concretion process for a structured grid computation: the Laplace operator of Figure 2.3. At every step, we will indicate language, equivalence class, and cost model of every level.

### 3.2.1 Functional Dependencies

The visual representation of functional dependencies of the laplace stencil is presented in Figure 3.2a. This is a very intuitive representation as it graphically depicts the application point and the shape of the stencil. Since structured grids computations are static fixed stencils, every element has the same shape relative to its application point. Therefore, such a simple representation is sufficient to characterize the whole computation. The cost model is very simple at this level. The cost is proportional to the numbers of application points that have to be updated. The semantic of the computation is given by the **shape** of the stencil.

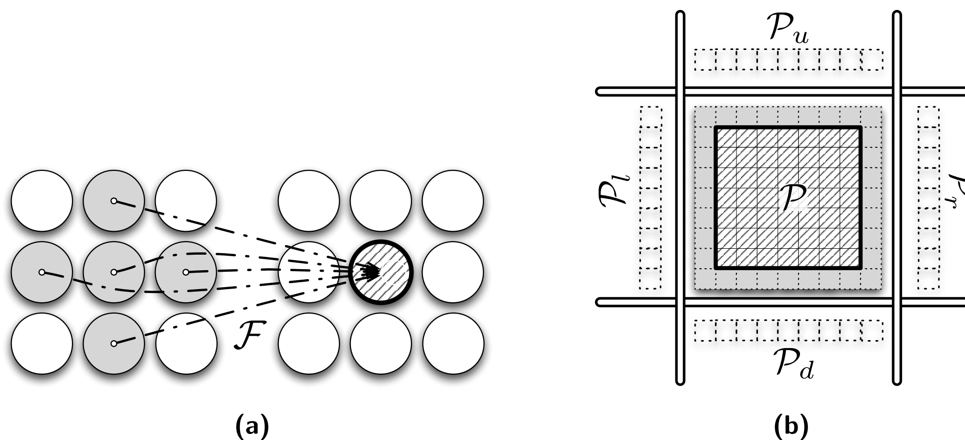


**Figure 3.2:** Graphical depiction of the functional dependencies of Laplace stencil computation ( Figure 3.2a ) and its partition dependencies ( Figure 3.2b ). White external cells represent the Incoming Dependency Set of $\mathcal{P}$. Grey cells represent elements in the Incoming Dependent Region while cells in the central zone, delineated by the black box, contain elements of the Incoming Independent Region. Since this particular shape is symmetric, outgoing and incoming sets overlap. However, this is not always the case because some stencil's shapes might not be symmetric.

### 3.2.2 Partition Dependencies

When passing from the functional dependencies level to the partition dependencies level, the elements will be divided into different partitions. A partition $\mathcal{P}$ will have a set of neighbour partitions and will interact with some of them. Functional dependencies between elements that cross the partition boundaries will be translated as dependencies between partitions. We can divide the partition in different **regions** depending on the functional

dependencies with the neighbouring partitions. Consider the elements whose dependencies extend outside the partition borders; we can define:

- **Incoming Dependent Region**: composed by elements which shape extend outside the partition borders.
- **Incoming Independent Region**: composed by all elements of partition $\mathcal{P}$ not included in the incoming dependent region.
- **Incoming Dependency Set**: all elements outside the partition $\mathcal{P}$ which are needed to perform the computation.

Then, we consider all elements which belong to the partition $\mathcal{P}$ and are needed by a neighbor partition.

- **Outgoing Dependent Region**: contains all elements which belong to the shape of some external element.
- **Outgoing Independent Region**: composed by elements which are not contained in any external element shape.
- **Outgoing Dependency Set**: all elements outside the partition $\mathcal{P}$ which need elements from $\mathcal{P}$.

Figure 3.2 shows the concretion process from the functional dependencies level to the partition dependencies level for the Laplace stencil. These dependencies contain the set of all functional dependencies among elements of the given partition and the elements in the neighbour partitions.

Notice that in order to represent computations at this level we can use a graphical representation. Moreover, equivalent computations at this level will produce the same output partition given the same input.

As anticipated in Section 3.1, the cost model at this level should be less abstract with respect to the model at the highest level. We define the cost at the partition dependencies level proportional to the number of application point to be updated and the size of the outgoing dependent region.

**Heuristics at the level of Partition Dependencies** After, having defined the first two levels of our hierarchy, we can introduce the first optimization by selecting an optimal partition strategy. Conceptually, we can group together elements in any possible way. However, we use the cost model of the partition dependencies level in order to define an heuristic to select a good partitioning strategy. Since there is cost associated to every element that has to be updated, we would want to have partition with the same

size. Therefore, we can rule out any non regular partitioning that would produce an unbalanced amount of work between partitions. Three suitable candidates in order to divide the working domain in equal parts for a two dimensional computations are: row, column and block partitioning. Since we are considering structured grids computations, we know that partition dependencies can only arise on the surface of a partition. Therefore, the surface of a partition is an upper bound on the size of partition dependencies. We can conclude that block partitioning is obviously the most effective as the surface to volume ratio is lower with respect to the other two partitioning strategies (Figure 3.3). Notice also how the number of partition dependencies are invariant with respect of the parallelism degree for row (or column) partitioning (Figure 3.4).

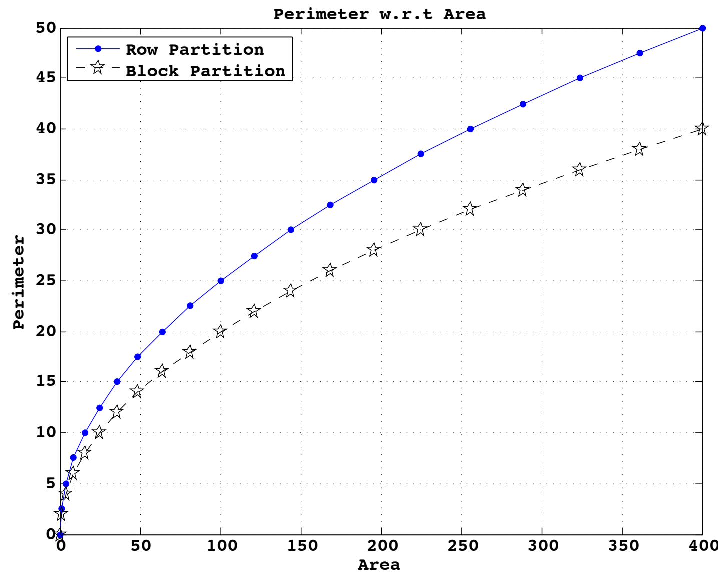


**Figure 3.3:** Perimeter of a partition with respect to partition area for row and block partitioning in a 2 dimensional working domain. If we consider a square working domain, row and column partitioning will produce the same results.

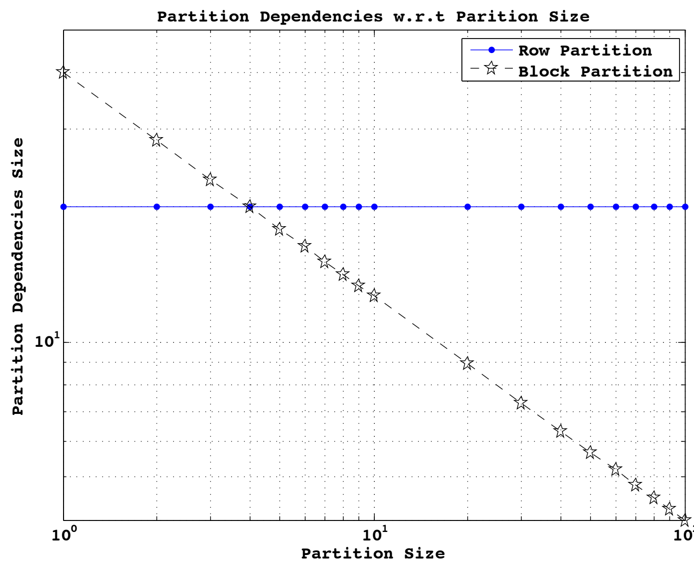


**Figure 3.4:** Dependencies between partitions with respect to parallelism degree for row or block partitioning in a two dimensional working domain.

By selecting block partitioning among all possible partition techniques, we have defined a heuristic for the class of structured grids computations.

Notice that heuristics, in general, are guided by a deep knowledge of the underlying concrete levels while the heuristic that we introduced is based on the codification of the domain specific knowledge associated with the application domain. Similarly to domain specific language, space pruning guided by high level knowledge is very effective, but it lacks generality since it applies only to a narrow class of computations.

### 3.2.3 Concurrent Level

When translating from the partition dependencies level to the concurrent level, the output is a program defined in a generic purpose programming language with additional mechanism to handle inter-process communications. We assume that a message passing paradigm is used. Therefore, the compiler implementing this phase of the concretion process has to determine the pattern of communications from the partition dependencies and generate code. Outgoing regions will be translated as send buffers while incoming regions will be translated as receive buffers. Communication primitives will be used to exchange this data. All data structures are instantiated at the concurrent level. More precisely, every partition is divided into *sections*. *Sections* are buffers which implement the *regions* defined at the partition level. Every *section*, which implements a portion of the dependent and independent regions, handles the communication with a neighbouring process.

At this level, we have a more detailed cost model. We assume that communication $T_{com}$ and computation $T_{calc}$ can overlap and define a cost model for both. Moreover, we define the cost of a communication of a message as $T_{setup} + m * T_{trasm}$ where the $T_{setup}$ and $T_{trasm}$ parameters depend on the target architecture[40]. $T_{calc}$ is instead proportional to the number of the instructions executed.

Applying the concretion function to the Laplace stencil, defined at the partition dependencies level, will produce the concurrent level code shown in Figure 3.5.

Notice that we are not taking advantage of overlapping communications and computation. All communications are performed at the beginning of a time step and subsequently all the elements of the partition are updated. We can optimize the concretion function in order to produce the concurrent code shown in Figure 3.6. By applying definition 3.6, this optimization is defined at the concurrent level and the reduction in cost can also be demonstrated at the concurrent level.

As we can see, the first phase consists of sending data to all adjacent partitions while computing the *Independent Incoming Region*. Sent elements belong to the Outgoing Dependent Region and are stored in buffers ( sections

```
double J_in[512][512], J_out[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step; i_step ++){

  SEND(P_(-1,0)); SEND(P_(0,-1));
  SEND(P_(0,1)); SEND(P_(+1,0));
  RECV(P_(-1,0)); RECV(P_(0,-1));
  RECV(P_(0,1)); RECV(P_(+1,0));

  COMPUTE( partition );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Figure 3.5:** Pseudocode representation of the Laplace stencil at the concurrent level. Partitions (or processes) are labelled by means of the coordinates with respect to the actual partition.

) specific to every neighbour. Then, elements of the *Incoming Dependency Set* are received from adjacent partitions and the *Incoming Dependent Region* is computed. In both cases, two matrices are used in order to preserve correctness of the algorithm. This algorithm will constitute the baseline for the optimizations presented in next chapters. Notice that a similar algorithm is used in [12] as a baseline MPI implementation for structured grid computations.

```
double J_in[512][512], J_out[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step; i_step ++){

  SEND(P_(-1,0)); SEND(P_(0,-1));
  SEND(P_(0,1)); SEND(P_(+1,0));

  COMPUTE( incoming independent region );

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

  COMPUTE( incoming dependent region );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Figure 3.6:** Pseudocode representation of the Laplace stencil at the concurrent level. With respect to concurrent code shown in Figure 3.5, we have that communication and computation overlaps.

### 3.2.4 Firmware Level

The language at the firmware level is object code (assembler). Therefore, it is strictly dependent on the architecture where the program is executed.

Since the language employed at the concurrent level is a normal programming language, translation from the upper level to this level is performed by a standard compiler. Therefore, we will not analyze in detail this last step of concretion. There are many optimization techniques which can be used when translating from the concurrent level to the firmware level. These are compiler optimizations in a classical sense[2] since their objective is to produce more efficient object code by:

- Exploiting the memory hierarchy on modern architectures.
- Modifying program structure in order to substitute inefficient pieces of code with more efficient ones.

A classification of these standard techniques is presented in Section 5.3.

## 3.3 Conclusions

We formally defined an hierarchical model to analyze the implementation and optimization of a generic class of computations (Section 3.1). We presented an instantiation of this model for the class of stencil computations and presented two simple examples of a heuristic (Section 3.2.2) and of an optimization (Section 3.2.3). In the following chapters, we will present more interesting examples of optimizations and their effects on structured grid computations.

# Chapter 4

# Optimizations for Structured Grids on Distributed Memory Architectures

In this chapter, we present a comprehensive review of optimizations techniques for structured grids computations on distributed memory architectures. In section 3.2, we have shown the process of concretion for a stencil computation. We described how elements are grouped together and consequently how functional dependencies become communications between workers. The final output of the concretion process is an executable file. Fortunately, in order to describe the class of optimizations presented in this chapter, it is sufficient to only consider the concretion process up to the concurrent level. In other words, the benefits from the optimizations here presented are evident at the concurrent level so there is no need to consider the whole concretion process.

## 4.1 $\mathcal{Q}$ transformations

$\mathcal{Q}$ transformations are a family of transformations, which performs a rigid translation of the stencil's shape. In section 2.2, we defined the shape of a stencil as the set of all functional dependencies of an element. Given the regularity of structured grids computations, the shape is the same for every element. Therefore, the shape of a d-dimensional stencil can be represented as a set of d-dimensional points which are displacements of the functional dependencies with respect to the application point. Consider as an example the 2 dimensional Laplace stencil in Figure 3.2a; its shape can be represented as $\{(0,-1),(0,0),(+1,0),(+1,0),(0,+1)\}$.

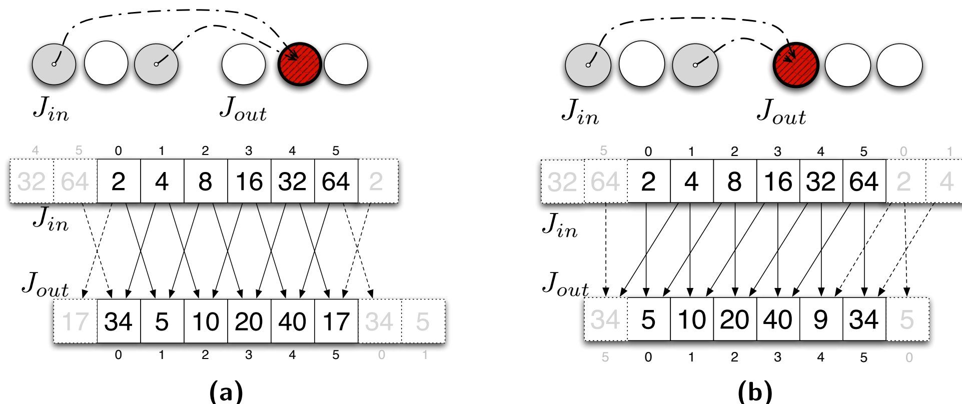


**Figure 4.1:** Graphical depiction of a time step of a mono dimensional jacobi stencil on a toroidal grid (Figure 4.1a) and a positive $\mathcal{Q}$ trasformation of the same stencil ( Figure 4.1b). The stencil kernel is the weighted average of the two elements of the shape.

**Definition 4.1** ($\mathcal{Q}$ transformations)**.** *A $\mathcal{Q}$ transformation is a rigid translation of a stencil shape. Given a d-dimensional shape $S = \{p_1, p_2, p_3, \cdots, p_i\}$ and a d-dimensional vector $q$, the $\mathcal{Q}$ transformation of $S$ given $q$ is $\mathcal{Q}_q(S) = \{p_1 + q, p_2 + q, p_3 + q, \cdots, p_i + q\}$.*

Firstly, consider that $\mathcal{Q}$ transformations are defined over Structured Grids. Since the shape of every element is modified in the exact same way, the resulting computation also belongs to the structured grids class.

The original and the modified programs are not equivalent in a classic sense. Many definitions of computational equivalence exist. By Allen and Kennedy[26], a transformed program is computationally equivalent to the original program if the original and transformed programs produce exactly the same output in the same order. This is obviously not the case for the family of $\mathcal{Q}$ transformations. However, since the transformations correspond to a rigid translation of the application point, the output values are indeed correct, but their position is altered with respect to the original version (Figure 4.1).

Therefore, it is still possible to recover the correct output grid by simply translating elements in the output data structures with vector $-q$ by the number of time steps of the computation. In order to formally define this property, it is necessary to introduce a new concept of computational equivalence[31].

**Definition 4.2** (Relaxed Computational Equivalence)**.** *A transformed program is relaxed equivalent to the original one if it is legal in the classical sense*

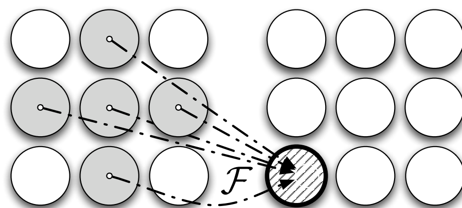


**Figure 4.2:** Graphical depiction of the incoming functional dependencies of Laplace stencil computation with positive $\mathcal{Q}$ Transformation applied.

*except for a spatial rearrangement of the values in the output data structures.*

Now we will claim, without proof (which can be found in [31]) that $\mathcal{Q}$ transformations are **relaxed safe** for Structured Grids computations which do not have functional dependencies between the same time step. The definition of relaxed safe follows from the previous definition of relaxed equivalence:

**Definition 4.3** (Relaxed Safe Transformation)**.** *A transformation $T$ is relaxed safe if for every transformed program $P_b$ such that $T(P_a) = P_b$, $P_b$ and the original program $P_a$ are relaxed equivalent.*

### 4.1.1 Positive and Negative $\mathcal{Q}$ transformations

We have defined $\mathcal{Q}$ transformations and we have shown that they produce the same results of the naive version. It is still unclear how they can be used to increase performances of a stencil computation, especially because in order to retrieve the final result, additional steps must be performed. Therefore, we introduce a particular class of $\mathcal{Q}$ transformations.

**Definition 4.4** (Positive $\mathcal{Q}$ trasformation )**.** *A $\mathcal{Q}$ transformation is a positive $\mathcal{Q}$ trasformation if the resulting shape contains only non negative coordinates.*

In the same way, we can define:

**Definition 4.5** (Negative $\mathcal{Q}$ trasformation )**.** *A $\mathcal{Q}$ transformation is a negative $\mathcal{Q}$ trasformation if the resulting shape contains only non positive coordinates.*

The positive $\mathcal{Q}$ trasformation for the Laplace stencil presented in Figure 3.2a is shown in Figure 4.2. Notice how every coordinate of the shape is positive with respect to the new application point. We can analyze the effect of this transformation on the Laplace stencil with the hierarchical model presented in Section 3.2.

1. Functional Dependencies level: It is clear that $\mathcal{Q}$ transformations work at the functional dependencies level. In this case, the naive shape of Figure 3.2a is translated into the shape of Figure 4.2.

2. Partition Dependencies level: We can see from Figure 4.3a and 4.3b that since the points of the transformed shape are all positive, we have that outgoing and incoming dependent regions do not overlap anymore. More precisely, by using either positive or negative $\mathcal{Q}$ transformations, the resulting outgoing and incoming dependent regions are the reflection of one another with respect to the center of the partition.

3. Concurrent Level: At this level, the advantage of positive (or negative) $\mathcal{Q}$ trasformations is evident because the **number** of communications between workers is reduced. By comparing Figure 4.3c and 4.3d, we can see that the number of send operation goes from 4 to 3. The same holds for receive operations. Generally, the maximum number of communications (incoming or outgoing) in a structured grid computation in $n$ dimension is reduced from $3^n - 1$ to $2^n - 1$[31] by using positive (or negative) $\mathcal{Q}$ transformations.

Notice, that the size of transferred data is unaffected by the transformation and only the number of communications is reduced.

The family of $\mathcal{Q}$ trasformations works by manipulating the stencil shape defined at the functional dependencies level but the effects of this transformations are evident only at the underlying concurrent level. At this point, we can introduce an additional optimization. By alternating positive and negative $\mathcal{Q}$ transformations, there is no need to translate values in the output data structures if the number of time steps is even (Figure 4.4b).

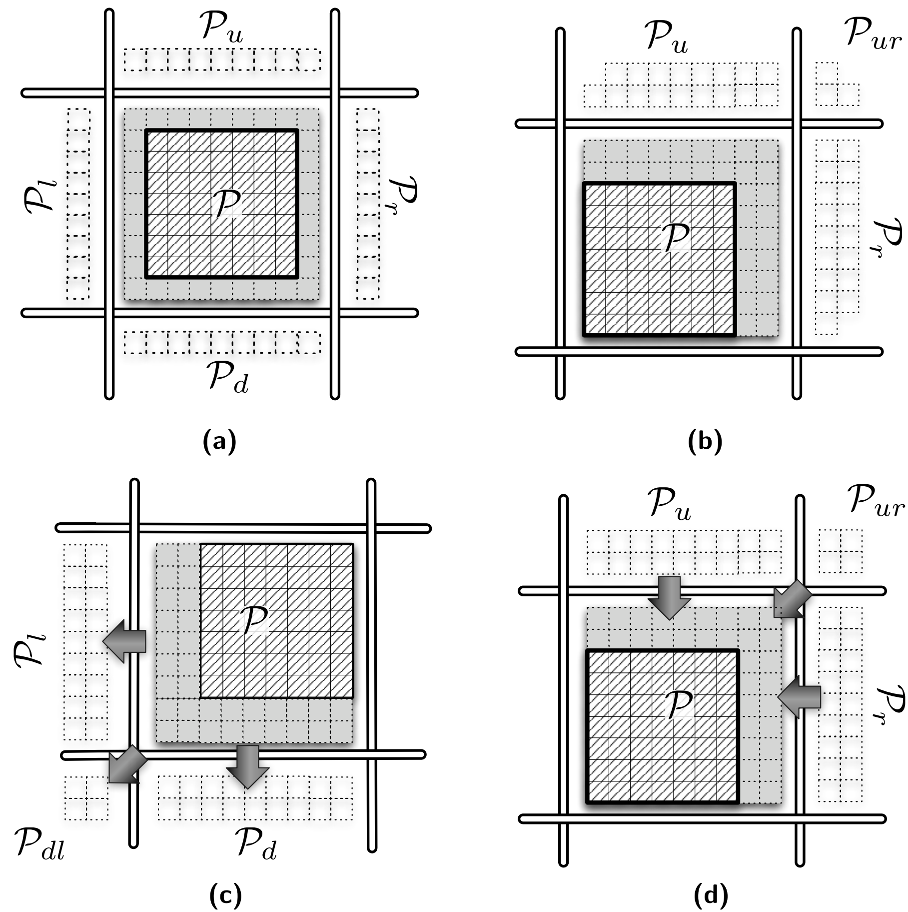


**Figure 4.3:** Graphical depiction of Partition dependencies in the case of the Laplace stencil (Figure 4.3a) and Laplace stencil with applied $\mathcal{Q}$ trasformation (Figure 4.3b). The region at the center of $\mathcal{P}$, surrounded by the thicker black box is the incoming independent region. The grey elements belong to the incoming dependent region, while the white elements belong to the incoming dependency set. Incoming (Figure 4.3d) and outgoing (Figura 4.3c) communications at the concurrent level with respective regions are also shown. Notice how outgoing and incoming regions overlap completely in the original version, but only partially in the transformed one.

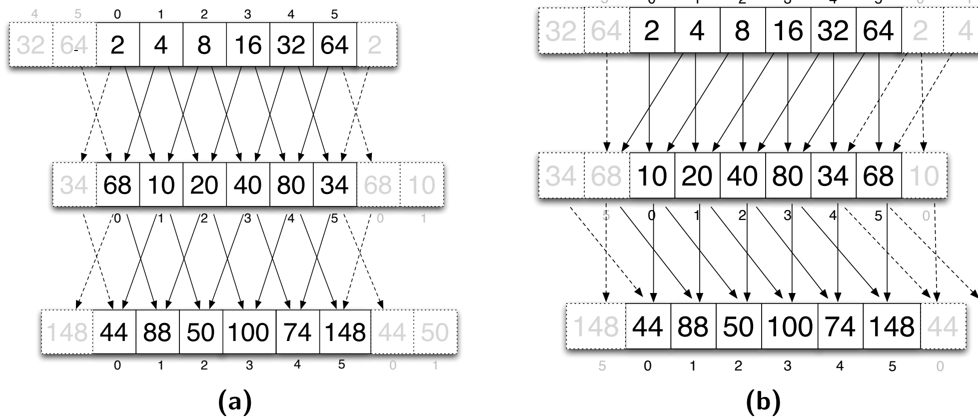


**Figure 4.4:** Graphical depiction of two time steps of a mono dimensional jacobi stencil on a toroidal grid (Figure 4.4a) and the same computations with a positive $\mathcal{Q}$ trasformation applied in the first time step and a negative $\mathcal{Q}$ trasformation applied at the second step ( Figure 4.4b). The stencil kernel is the sum of the two elements of the shape. Notice that the final displacement of output values is correct.

### 4.1.2 $\mathcal{QM}$ and $\mathcal{QW}$ transformations

We now introduce two optimizations that belongs to the Family of $\mathcal{Q}$ transformations and are used to reduce memory occupation in stencil applications. If we consider a general stencil program, with no dependencies inside a time step, implemented at the concurrent level (Figure 3.6), it requires two data structures[1] to store the working domain. One data structure will contain elements at time step $t$ while the other will be used to store element of time step $t+1$ when they are computed. In other words, one data structure will store elements at even time steps while the other will store elements at odd time steps. It is indeed possible to map these two data structures to a single one. This requires to modify both the stencil's shape and the access pattern. Both modifications can be computed by analyzing the shape of the computation.

We can define both positive and negative $\mathcal{QM}$ transformations.

**Definition 4.6** (Positive $\mathcal{QM}$ trasformation)**.** *A positive $\mathcal{QM}$ trasformationis a rigid translation of a stencil shape which is performed along a given axis. Given stencil shape in a d-dimensional space: $S = \{p_1, p_2, p_3, \cdots, p_i\}$, there exists a minimum integer $i$ such that $S_1 = \{p_1 + [i, 0, \cdots, 0], p_2 + [i, 0, \cdots, 0], p_3 + [i, 0, \cdots, 0], \cdots, p_i + [i, 0, \cdots, 0]\}$ has only positive values in the first coordinate of its shape points.*

We can also define a negative $\mathcal{QM}$ trasformationby enforcing that the first coordinate of resulting shape points are negative. Notice the difference between positive $\mathcal{Q}$ trasformation s, which enforce that every coordinate of every point of the shape should be non negative and positive $\mathcal{QM}$ trasformations which requires only one coordinate of every point to be strictly positive. Similarly to what we did in Section 4.1 we will analyze this optimization in the hierarchical model (Section 3.2). For simplicity, we will consider the sequential case (no partition dependencies level) and periodic boundary conditions.

1. Functional Dependencies level: It is clear that $\mathcal{Q}$ transformations work at the functional dependencies level. Consider as an example the mono dimensional Jacobi update that we presented in Section 2.3.3. Figure 4.5 shows the original (4.5a) and modified shape (4.5b).

2. Concurrent Level: At this level, the advantage of positive (or negative) $\mathcal{QM}$ trasformations is evident. Consider the modified shape in Figure

[1] We are using the generic term data structure instead other more specific terms, e.g. matrix, because the working domain of a stencil is generally defined on a d-dimensional space.

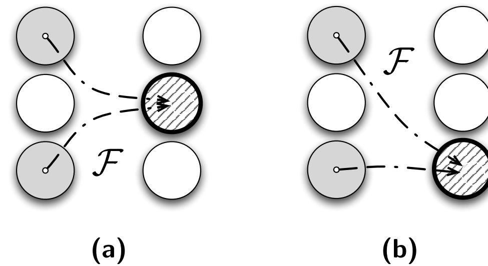


**Figure 4.5:** Graphical depiction of functional dependencies in the case of naive jacobi stencil (Figure 4.5a) and the Jacobi stencil with applied positive $\mathcal{QM}$ trasformation(Figure 4.5b).

4.5b. It is possible to update the domain elements using a single data structure, by storing results from the bottom up (Figure 4.6). However, because of periodic boundary conditions, it is necessary to use an additional buffer to store elements at the boundary of the domain that are updated first. It has been proven[31] that the size of the additional space for buffering is negligible with respect of the size of the working domain.

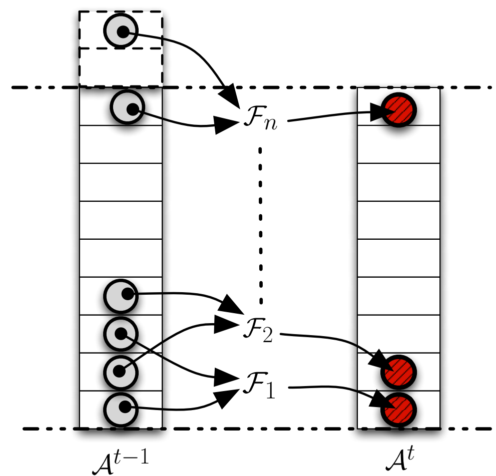


**Figure 4.6:** $\mathcal{QM}$ transformation of the one dimensional jacobi stencil computation presented in Figure 4.5. Operation are labeled as $\mathcal{F}_1 \cdots \mathcal{F}_n$ in the order in which they **must** be executed. The two array in the figure are logical since the implementation will utilize a single data structure. The ghost cells in the upper left corner has to be updated at the beginning of every time step in order to preserve correctness.

Consider now a distributed architecture. We cannot overlap computation and communication anymore since the modifications made at the concurrent level by $\mathcal{QM}$ trasformations enforces the update of elements in a given order. At the concurrent level, the sections of data structures that correspond to the outgoing dependency set are sent before the computation starts. Similarly, elements of the incoming dependency set have to be received before starting the computation. A complete example of a positive $\mathcal{QM}$ trasformationis presented for the Laplace Stencil (Figure 4.7).

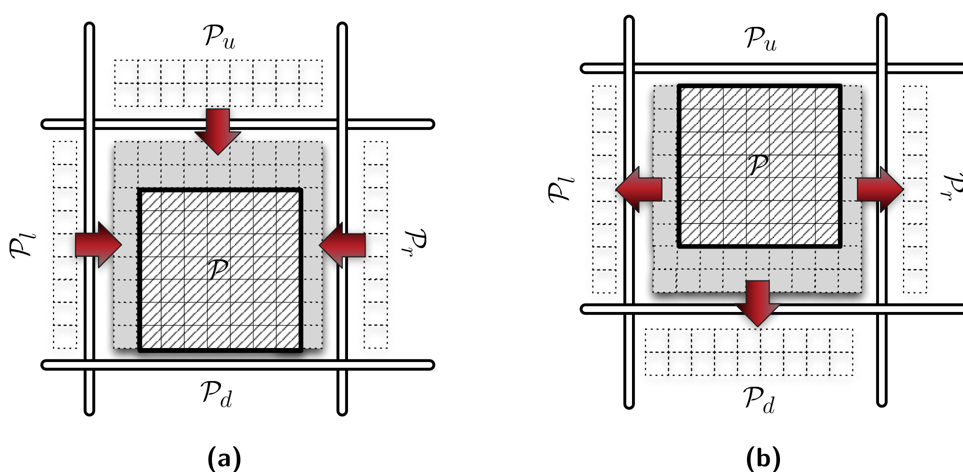


**Figure 4.7:** Figure 4.7a shows a graphical depiction of the incoming communications of Laplace stencil computation. The grey cells represent elements in the Incoming Dependent Region while cells in the central zone, surrounded by the black box, contain elements of the Incoming Independent Region. Figure 4.7b shows a graphical depiction of the outgoing communications ( Figure 4.7b ) with $\mathcal{QM}$ trasformation applied. The grey cells represent elements in the Outgoing Dependent Region while cells in the central zone, surrounded by the black box, contains element of the Outgoing Independent Region.

### $\mathcal{QW}$ trasformations

Moreover, we realized that positive (and negative) $\mathcal{Q}$ trasformations also allows updates on a single data structure. Some modifications are necessary at the concurrent level in order to have $\mathcal{Q}$ transformed programs on a single data structure. This new class of optimizations is called $\mathcal{QW}$ trasformations.

1. Elements of the outgoing dependency set must be sent before update starts.

2. Elements of the incoming dependency set has to be received before starting the computation.

3. A wavefront update of elements is enforced.

Wavefront computations are well known in literature and the Gauss Seidel method presented in Section 2.3.3 belongs to this class. In wavefront computations, the dependency between the elements of the domain requires that the computation starts at a singular point at a corner of the plane and propagates its effect diagonally to other elements. In the two dimensional case, using a positive $\mathcal{Q}$ trasformation , we can see that the computation should start on the lower left corner and every element update allows the update of its right and upper neighbour (Figure 4.8b). At the concurrent level, it is clear that by using $\mathcal{QW}$ trasformations we must first synchronize with neighbouring processes and then update local elements. We have successfully combined the positive effects of positive $\mathcal{Q}$ trasformations and $\mathcal{QM}$ trasformations. Similarly to $\mathcal{Q}$ trasformation we can alternate positive and negative transformations in order to restore the correct position of every output element.

**Hybrid Programming Model** Until now, we only considered a distributed environment where message passing primitives are used to communicate between processes. We briefly analyze the effects of the $\mathcal{Q}$ trasformation when a hybrid programming model is used. By hybrid programming model we mean that message passing is used between processing nodes and shared memory mechanisms are used inside a node. Positive $\mathcal{Q}$ trasformation allows us to split the update of partition points in a time step in any possible combination without affecting the correctness. On the other hand, $\mathcal{QW}$ trasformations and $\mathcal{QM}$ trasformations introduce some data dependencies that have to be conserved. Notice that these are not functional dependencies (as in the case of Gauss Seidel) since they are not present at the functional dependencies level. At the functional dependencies level, two **logical** data structures exist. Data dependencies arise at the concurrent level as a consequence of the implementation of logical data structures with real data structures. From Figure 4.8, we can see that $\mathcal{QM}$ trasformations preserve some usable parallelism at a finer grain[2] with respect to $\mathcal{Q}$ trasformations . On the other hand, $\mathcal{QW}$ trasformations have no readily available parallelism. Only elements lying on the same diagonal can be updated in parallel, ( similarly to the Gauss Seidel computation ).

[2]By finer grain we mean that synchronizations are more frequent.

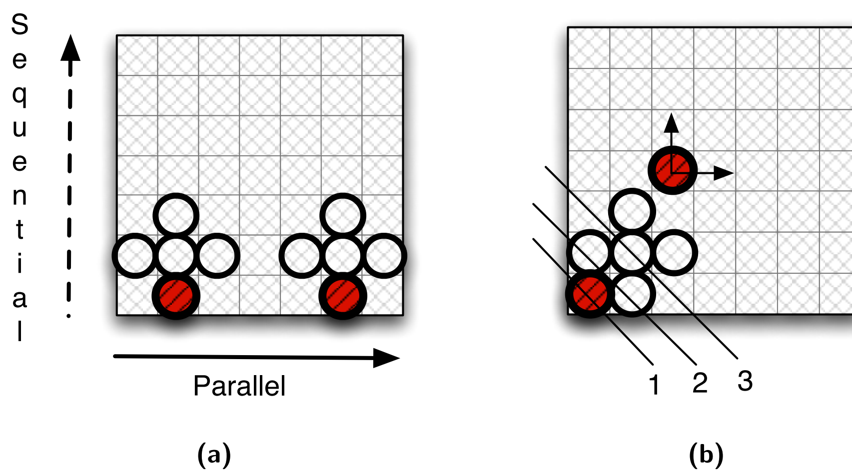


**Figure 4.8:** Execution of transformed Laplace Stencil with $\mathcal{QM}$ trasformation(Figure 4.8a). The application point has been shifted by one position along the vertical axis. Therefore points in the same horizontal plane can be updated in parallel. Execution of transformed Laplace Stencil with $\mathcal{QW}$ trasformation(Figure 4.8b). Notice that the shape is exactly the same of the $\mathcal{Q}$ trasformation version. The first element that has to be updated is shown in the lower left corner. A generic element is also shown; the two arrows points to the elements that cannot be updated **before** him. Therefore the updates must be executed in a waveform pattern. Different axis (1,2 and 3 in the figure) represents set of elements that can be updated in parallel if loop skewing is used.

## 4.2 Shift Method

The shift method ($\mathcal{SH}$ method ) is an optimization of the communication patterns and it was presented by Plimpton on [33]. It works on the concurrent level aggregating together communications. This is done by modifying the parallel program structure and grouping communications with diagonal neighbours along the main axis. Therefore, unlike transformations of the $\mathcal{Q}$ family, the Shift Method works entirely at the concurrent level. Consider that this method gives no performance gain if there are no diagonal dependencies, such as in the case of the Laplace stencil ( Figure 3.2a). Shift method becomes very useful if applied together with $\mathcal{Q}$ trasformation in order to further reduce the number of communications. More precisely, consider the case of the Laplace stencil where diagonal dependencies appear after having applied the $\mathcal{Q}$ trasformation (Figure 4.2). By applying a positive $\mathcal{Q}$ trasformation , the number of required communications goes from four to three. By further applying the shift method, we can reduce the number of communications

from three to two. It has been proved in [31] that the combination of positive $\mathcal{Q}$ trasformations and shift method gives the lowest theoretical bound on the number of communications which is equal to the number of spatial dimensions.

We present now the application of the shift method on the nine point stencil. The nine point stencil is an higher precision approximation of the Laplace stencil[28]. We can see from Figures 4.10a and 4.10b, that communications are first performed on the vertical axes (1). Then, some of the ghost cells elements received have to be copied on the send buffers (2). Matching send and receive operations are performed on the horizontal axis (3), (4).

Figure 4.1 shows the structure of the concurrent code for a shifted stencil computation. It can be compared with the baseline code presented in Figure 3.6.

It is evident that this method requires more complex concurrent code. A time step has to be divided into $d$ small steps where $d$ is the number of spatial dimensions of the stencil computation. Therefore, if we want to overlap communication and computation, the incoming independent region has to be updated in $d$ separate phases. We can summarize these additional difficulties as:

- Buffers size vary depending on their position on the working domain.
- More synchronization among processes is required.
- Superposition of computation and communication is more complex.

```
double J_in[512][512], J_out[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step; i_step ++){

  SEND(P_(-1,0)); SEND(P_(+1,0));

  COMPUTE( half independent incoming region );

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

  copy(border elements of receive buffer into send buffers)

  SEND(P_(0,-1)); SEND(P_(0,+1));

  COMPUTE( remaining half of independent incoming region );

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

  COMPUTE( dependent incoming region );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Listing 4.1:** Pseudocode representation of a **two dimensional** nine point stencil, which has dependencies with every surrounding partition. Partitions (or processes) are labeled by means of coordinates with respect to the examined partition.

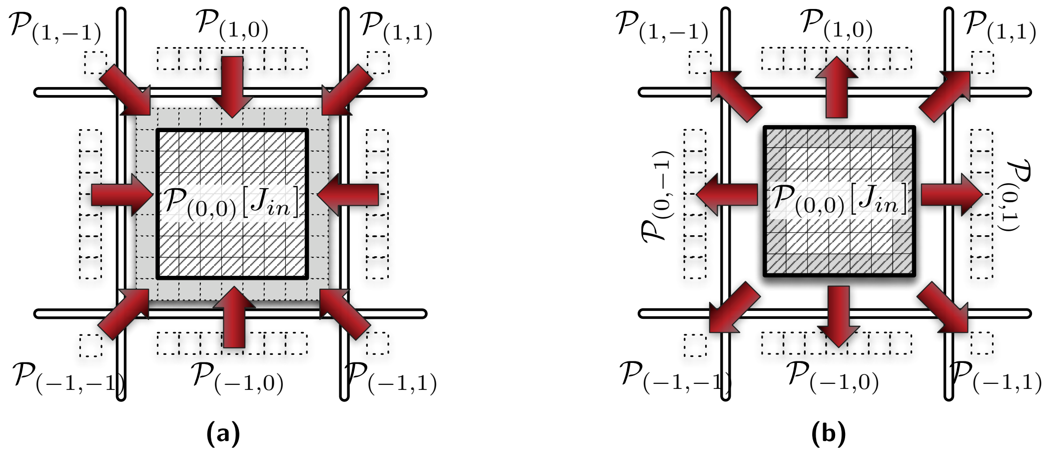


**Figure 4.9:** Graphical representation of incoming communications (fig. 4.9a) and outgoing communications (fig. 4.9b) of the nine point stencil.

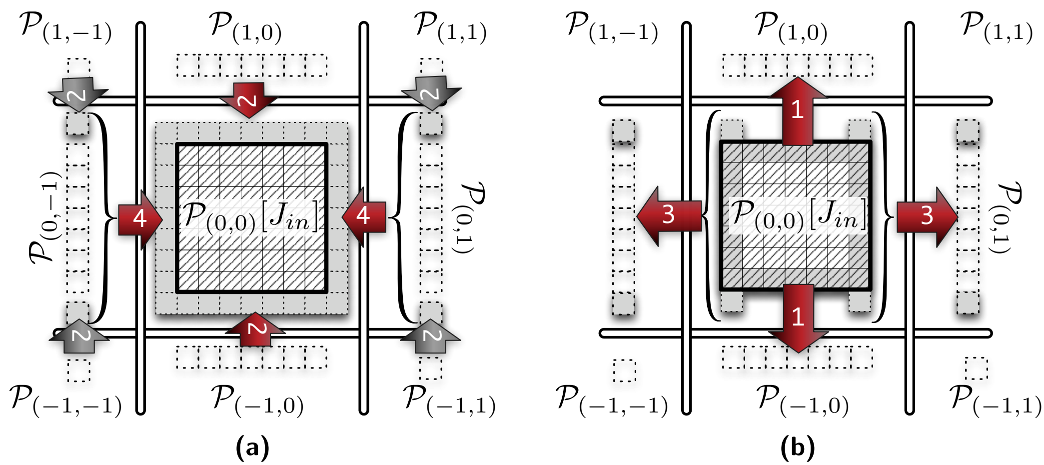


**Figure 4.10:** Graphical representation of the incoming communication pattern (fig. 4.10a) and outgoing communication pattern (fig. 4.10b) of the nine point stencil with shift method applied. Numbering reflects the order of the communications needed to preserve correctness.

## 4.3 Step Fusion

The step fusion method was developed by Meneghin[31] and it is strongly influenced from the *ghost cell expansion* method presented in [17]. The *ghost cell expansion* method (also called **oversending** in literature) is based on sending more data less frequently between partitions. Therefore, we have more loosely coupled processes and a reduction of the number of communications necessary to perform the computation.

### 4.3.1 Ghost Cell Expansion

In [17], it was proposed to group together elements needed by a neighbour processes in more than one step, or in other words to expand the ghost cells of every partition. This produces an interesting balance among computation and communication time spent by each process to perform the computation. The communication pattern obtained, using this method to group two consecutive steps, is presented in Figure 4.11. For simplicity, a two dimensional stencil featuring functional dependencies only on the vertical axis is used. We can compare the naive and oversend version of this simplified stencil:

1. On timestep 0, the oversend version transfers twice the necessary data size and so, the receive buffer (ghost cells) has to be doubled in size.

2. While communications are performed, both the naive and oversend version compute the incoming independent region.

3. When communications terminate, both versions update the incoming dependent region. The oversend version also updates a portion of its ghost cells.

4. The oversending version can complete timestep 1 without communicating with neighbouring processes.

It is now clear that this method work at the concurrent level. Similarly for the shift method, computation and communication patterns are altered, but the overlying representation of the stencil remains unchanged. The ghost cell expansion can be applied multiple times. Its effects on the number of communications are dramatic since they are reduced linearly with respect of the level of expansion. By expanding ghost cells by a factor of two (Figure 4.11), we have half the communications of the naive version. A factor of three of expansion would produce one third of the communications, and so on. However, this reduction in the number of communications comes at a cost:

1. Every level of ghost cell expansion requires that an additional set of ghost cells have to be added, thus increasing the memory footprint of the computation. The size of the ghost cells is proportional to the surface of the partition. Depending on the surface/volume ratio of the partition, the increase in memory requirements can be substantial.

2. Additional ghost cells have to be updated. The previous consideration about the surface/volume ratio holds also for the amount of additional updates to perform. Moreover, the relationship between the level of expansion and number of updates is quadratic. Given the original size of ghost cells is $g$, the number of updates for $n$ level of expansions are $\Sigma_{i=1}^{n} i * g = g * \Sigma_{i=1}^{n} i = \Theta(n^2)$

3. Communications can only overlap with the update of the incoming independent region at the first timestep.

From this analysis, we say, without loss of generality, that this method is beneficial only when communications are the bottleneck of the computation.

The structure of the concurrent code for a single ghost cell expansion is shown in Figure 4.2.

**Interaction with $\mathcal{Q}$ trasformations** Notice that we can combine $\mathcal{Q}$ trasformations and the ghost cell expansion method since they work at different levels. Firstly, the shape is modified by a $\mathcal{Q}$ trasformation at the functional dependencies level. Then, the implementation at the concurrent level is modified by the ghost cell expansion method. The positive effects of both class of optimizations are therefore combined and an example is presented in Figure 4.11.

```
double J_in[512][512], J_out[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step/2; i_step+ = 2){

  SEND(P_(−1,0));
  SEND(P_(0,−1));
  SEND(P_(0,1));
    SEND(P_(+1,0));

  COMPUTE( independent incoming region );

  RECV(P_(−1,0));
  RECV(P_(0,−1));
  RECV(P_(0,1));
  RECV(P_(+1,0));

  COMPUTE( incoming dependent region);

  COMPUTE( external elements ∈ dependency set );

  //Next Step

  swap(J_in, J_out);

  COMPUTE( partition );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Listing 4.2:** Pseudocode representation of a two dimensional laplace stencil with oversending method applied.

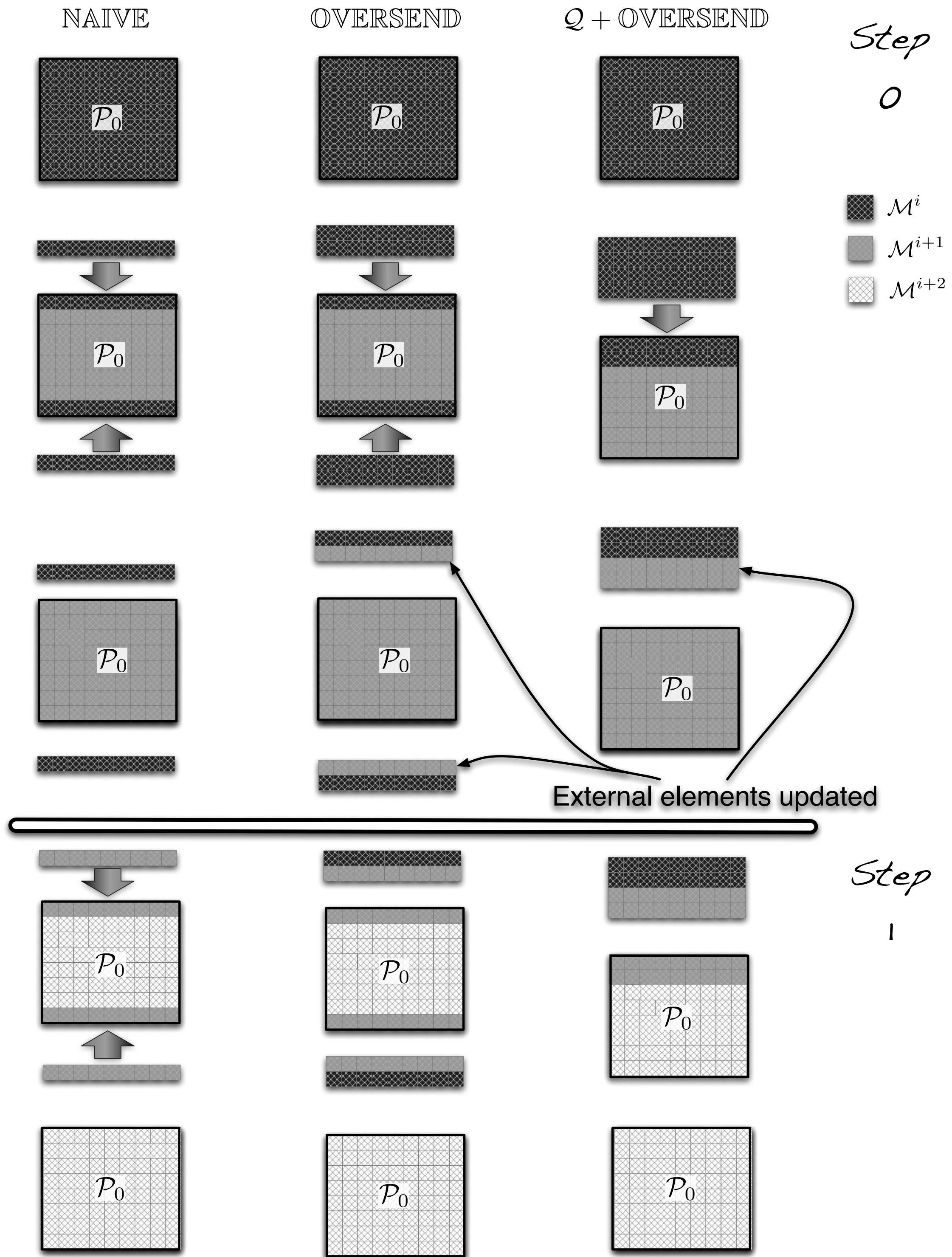


**Figure 4.11:** Communication patterns of a generic stencil computation (left) compared with the same computation with oversending method applied (center) and with a $\mathcal{Q}$ trasformation applied (right).

### 4.3.2 Collapsing Time Steps

In [31], the Step Fusion method ($\mathcal{SF}$ ) is proposed as an alternate version of the ghost cell expansion with additional positive effects on the memory hierarchy. The step fusion method is performed directly on the functional dependencies layer by grouping together two ( or more ) consecutive steps. Notice that this is possible only if the computation kernel of the stencil is a linear function. As an example consider the jacobi stencil:

```
forall i,j = 0 .. M−1 do B[i,j] = 0.25∗( A[i−1,j] + A[i,j−1] +
          A[i+1,j] + A[i,j+1] )
```

and its transformed version with two fused time steps:

```
forall i,j = 0 .. M−1 do B[i,j] = 0.0625∗( A[i−2,j] + 2∗A[i−1,j−1] +
          2∗A[i−1,j+1] + A[i,j+2] + 4∗A[i,j] + A[i,j−2] +
          2∗A[i+1,j+1] +2∗A[i+1,j−1] + A[i+1,j] )
```

It has been proven that the communications between processes for applications with $\mathcal{SF}$ applied are the same for applications with oversending. Similarly to the oversending method, it is possible to apply step fusion multiple times . Figure 4.12 shows multiple levels of step fusion and highlights the relationship between $\mathcal{SF}$ and functional dependencies. We can see how different levels of step fusion can be defined recursively. Every shape at level $i$ of step fusion is obtained by adding all functional dependencies of shape points at level $i-1$.

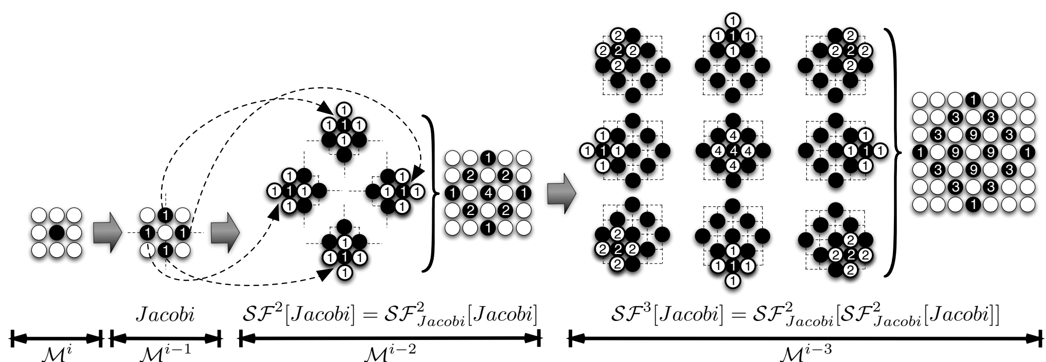


**Figure 4.12:** Step Fusion method applied to the two dimensional Jacobi stencil computation.

Since this transformation is performed at the level of functional dependencies,we do not have the overhead of updating the ghost cells. However, the two stencil performs a fairly different computation. When defining the cost

model of the concurrent level, we said that the $T_{calc}$ was proportional to the number of executed statements. However, both updates can be expressed as a single statement although it is evident that they perform a different number of floating point operations and access a different number of elements. Therefore, in order to asses the effect on performances of the $\mathcal{SF}$ trasformation , we could modify the cost model of $T_{calc}$ at the concurrent level. On the other hand, $T_{calc}$ was defined in such a naive way because, at the concurrent level, we are primarily considering the effects of interactions between processes. In other words, the optimization of the parallel implementations is at a higher level of abstraction ( and importance ) with respect to the optimization of the sequential version. We can not even analyze such modifications at the firmware level because its cost model is strictly dependent on the architecture and features no abstraction whatsoever. Therefore, in order to present optimizations of the sequential program, it is convenient to add an intermediate level of abstraction between the concurrent level and the firmware level. In particular, since sequential performances of stencil computations are primarily related to the efficiency of the memory system ( Section 2.4), we will present a new level tailored to consider a generic memory hierarchy in the next chapter.

## 4.4 Conclusions

In this section, we have shown a comprehensive list of optimizations for structured grids computations on distributed architectures. We omitted proofs of correctness and preferred to show the effects of the different techniques mostly in a graphical way which we believe is the best to explain how these methods work and why they are beneficial for performances. The reader can find all formal proofs in [31].

Consistently with the model presented in Section 3.2, we have shown how we can join optimizations at different levels of abstraction. We have shown that by combining positive (or negative) $\mathcal{Q}$ trasformation s with the $\mathcal{SH}$ method , we obtain the minimum number of communications which is equal to $d$ for a $d-$dimensional stencil (both for receive and send operations). By also applying the ghost cell expansion method we can further reduce the number of communications linearly with respect to the level of expansion. However, unlike $\mathcal{Q}$ trasformation s and the $\mathcal{SH}$ method , the ghost cell expansion method has negative effects on the $T_{calc}$ of the computation since more elements have to be updated.

We also presented an extension to the ghost cell expansion method called the $\mathcal{SF}$ trasformation [31]. In this case, we have all the positive effects of

ghost cell expansion, without the drawback of updating external elements. However, we could not perform a comprehensive analysis for this method. In fact, the cost model at the concurrent level is not suited for the analysis of the impact of this transformation on the memory hierarchy and the cost model at the firmware level is too concrete to perform a reasonable analysis.

# Chapter 5

# Optimizing for Locality on Structured Grids Computations

In this chapter, we present optimizations that exploit the memory hierarchy of modern architectures. Optimizations described here (except $\mathcal{SF}$ trasformation ) are used in general for many computations; however, we will consider primarily their effects on structured grids computations. In order to present these optimizations in a structured way, we have to extend the model explained in Section 3.2. The effects of $\mathcal{SF}$ trasformation on the memory hierarchy are presented as an introduction to the problems that stem from deriving an accurate cost model for memory optimizations. A brief overview of the classic optimizations methods used by compilers is presented in Section 5.3. The class of loop reordering techniques is analyzed in Section 5.4. Finally, the application of such techniques on structured grids computations is shown in Section 5.5.

## 5.1 Extended Model

Drawing from the analysis of the previous chapter, we will now extend the model described in Section 3.2 in order to represent a wider set of optimizations with respect to what was presented in [31]. The model has an abrupt jump between the concurrent level and the firmware level. Thus, to fill this gap, we are going to introduce an important addition to the model: memory hierarchies.

In section 4.3.2, we introduced the $\mathcal{SF}$ trasformation . We concluded that, when applicable, the $\mathcal{SF}$ trasformation can decrease the number of communications among workers linearly with respect to the fusion level, thus reducing the $T_{comm}$ of the computation. These effects derive from the fact

that the $\mathcal{SF}$ trasformation collapses together different time steps. However, we could not assess in a definitive way the overall impact of this optimization. In fact, the dimensionality of the stencil shape, i.e., the number of functional dependencies, is increased sensibly. Therefore, we have to consider that the cost for a single update is higher and we have to determine what is the impact on the final $T_{calc}$ of the computation. We analyze these concepts for the two dimensional Jacobi stencil:

```
for t = 1..T
    for i = 0 .. M−1 do
        for j = 0 .. M−1 do
            B[i,j] = 0.25*( A[i−1,j] + A[i,j−1] + A[i+1,j] + A[i,j+1] )
    swap(A,B)
```

**Listing 5.1:** Pseudocode of naive 2d Jacobi stencil.

and its transformed version with two fused time steps:

```
for t = 1..T/2
    for i = 0 .. M−1 do
        for j = 0 .. M−1 do
            B[i,j] = 0.0625*( A[i−2,j] + 2*A[i−1,j−1] +
            2*A[i−1,j+1] + A[i,j+2] + 4*A[i,j] + A[i,j−2] +
            2*A[i+1,j+1] +2*A[i+1,j−1] + A[i+1,j] )
    swap(A,B)
```

**Listing 5.2:** Pseudocode of 2d Jacobi stencil with applied $\mathcal{SF}$ method.

It is clear that the number of sweeps over the data structures is halved; therefore, the number of updates to perform is also halved. If a single update in the $\mathcal{SF}$ version costs at most two times the naive version update, then we have a clear performance benefit. Thus, we need to decide what cost should be associated to every statement.

We could consider the number of floating point operations performed. Since the naive version executes 4 flops per update while the $\mathcal{SF}$ version performs 13 flops per update, we can conclude that $T^{naive}_{calc} = T * i * j * 4$ flops while $T^{\mathcal{SF}}_{calc} = \frac{T}{2} * i * j * 13$. Therefore the $T_{calc}$ of the $\mathcal{SF}$ version is substantially higher.

On the other hand, we could consider the number of memory accesses $M$ using a very simple RAM machine model where every access has a unitary cost. Since the naive version needs 4 array elements per update while the $\mathcal{SF}$ version uses 9 array elements per update, we can conclude that $T^{naive}_{calc} = T * i * j * 4$ while $T^{\mathcal{SF}}_{calc} = \frac{T}{2} * i * j * 9$. Using this RAM model, the $T_{calc}$ of the $\mathcal{SF}$ version is still higher, but not as much as using the flop model.

None of these two models are suitable for our purposes since they are both too simple. As we anticipated in Section 2.4, arithmetic intensity[1] for structured grids is usually too low to be influential. Therefore, the first model is not representative for structured grids computations. The second model is probably more accurate, but it completely neglects the effect of the memory hierarchy.

An accurate cost model for our needs is the external memory model [1]. Every node has an unbounded memory $M$ and a cache of limited size $\mathbf{C}$. Data is transferred from the Memory to the Cache in Blocks of size $\mathbf{B}$. The cache is completely associative and an optimal replacement strategy is implemented. By varying the parameters $\mathbf{C}$ and $\mathbf{B}$, we can effectively model every level of the memory hierarchy and the $T_{calc}$ corresponds to the number of memory accesses. The equivalence classes at this level are the same as the Concurrent Level. We could change the language from a generic language to one where load and store operations are explicit (they are fundamental at this level). Nonetheless we believe that this would render the exposition too cumbersome since the pattern of load and store operations can be derived from pseudocode.

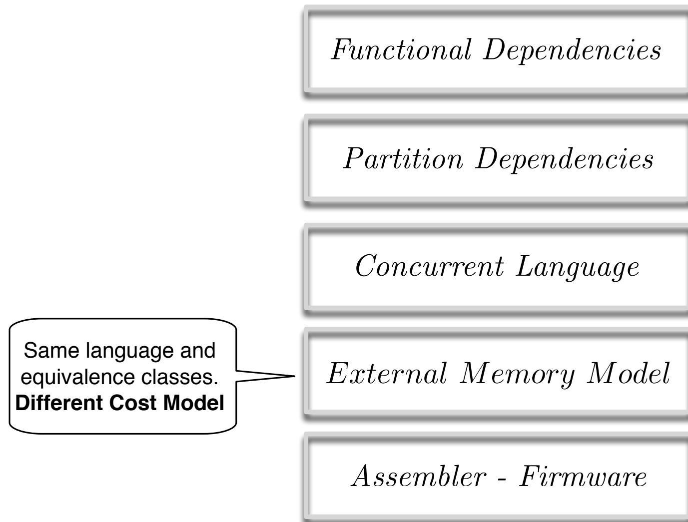


**Figure 5.1:** Extended reference model.

In conclusion, in this extended model (Figure 5.1), we can analyze at the concurrent level the effects of different parallelization schemes assuming the $T_{calc}$ is fixed for the given computation and using the cost model $T_{comm}$ for

[1]The number of floating point operations per update.

communications among workers. By utilizing such methodology, we analyzed different partitioning schemes (Section 3.2.2) and many optimizations that reduce of the number of communications (Chapther 4). At the level of the external memory model, we have a more complex definition of the $T_{calc}$ while the $T_{comm}$ is unchanged from the level above. We can analyze at the external memory model level, the impact of optimizations on the exploitation of the memory hierarchy and finally derive a more concrete estimate on the effects of the $\mathcal{SF}$ trasformation .

## 5.2 Analysis of the Impact of the Step Fusion Method on Memory Hierarchies

In this section, we derive the $T_{calc}$ of a program after applying $\mathcal{SF}$ with respect to the original one. Because of memory hierarchies, we cannot assign a cost to every statement per se, but for a given piece of code, we have to determine what is its working set and if it has temporal and/or spatial locality[41]. In both cases, the naive (Listing 5.1) and $\mathcal{SF}$ version (Listing 5.2), the working set corresponds to the matrices A and B accessed in a unit stride (maximal spatial locality). We have substantial reuse since matrix A is read T times in the naive case and T/2 times in the $\mathcal{SF}$ version of the algorithm.

The working set of the computation corresponds to the working set of the $i$ indexed loop. The first distinction is to determine if the entire working set ( of size $2 * M^2$ ) fits in memory. If this is the case, the two algorithms have equal performances on an external memory model because they produce the same number of faults $2 * M^2/\mathbf{B}$. In fact, matrices are fetched from memory and they are never evicted from the caches. Moreover, we should expect the first algorithm to be faster since it performs less floating point operations. However, typical problems' grids cannot be stored in cache, at least for the lower levels of the memory hierarchy.

Now, under the hypothesis that the entire working set of the computation does not fit in cache, we consider the working set of the innermost loop. The innermost loop updates a row of the matrix A, composed of $M$ elements in a unit stride fashion.

Since elements of the output matrix are updated without reading their values, they will not produce a cache miss when referenced. On the other hand, the number of elements to read depends on the shape of the stencil (Figure 5.2). Since stencil kernels access only a limited number of points close in space, the working set of the innermost loop corresponds to a horizontal

"strip" of the matrices (Figure 5.4).

**Definition 5.1** (Extent of a shape)**.** *Given a shape of a d-dimensional stencil* $\mathcal{S} = (s_0^0, \cdots s_d^0), \cdots, (s_0^n, \cdots s_d^n)$ *composed by n points, its extent* $\psi$ *is equal to* $max(s_0^0, \cdots, s_0^n) - min(s_0^0, \cdots, s_0^n)$.

The extent corresponds to the distance, along the outermost dimension, between the trailing and leading point of the stencil. As an example, the naive version (Listing 5.1) requires to read three rows of the matrix in order to update a single one (extent equal to three). The $\mathcal{SF}$ version (Listing 5.2) requires five rows of the matrix in order to update one row. Generalizing for a level of fusion $i$ ( where $i = 1$ indicates no step fusion ) the innermost loop accesses $(2*i+1)*M$ elements. In other words, the working set corresponds to $\psi$ which is the vertical extent of the shape. We face two possibilities:

I The working set can be held in cache; therefore, the whole matrix is loaded from memory only once during a single time step which implies $M^2/B$ faults occur at every time step. The step fusion version performs half the time steps of the naive version; therefore, the number of faults is also halved.

II The working set cannot be held in cache; therefore, during the update of different rows, the whole working set is loaded in cache yielding a final number of faults of $\frac{T}{i} * \psi * M^2$.

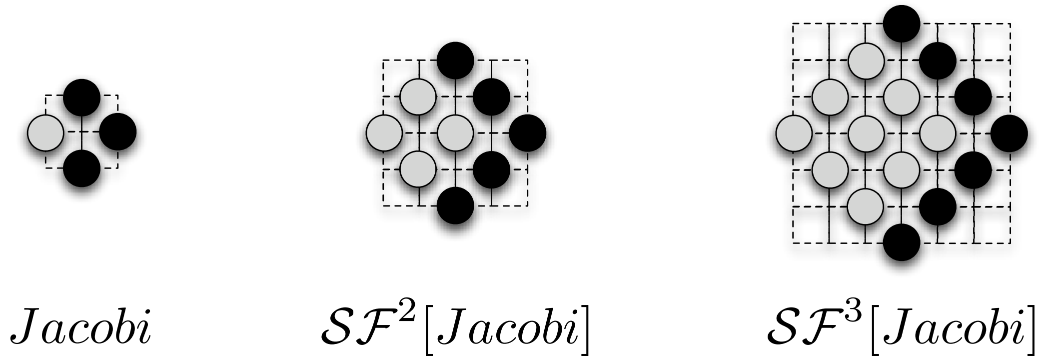


**Figure 5.2:** Memory accesses with increasing step fusion factor in Jacobi computation.

From the information that we have gathered, we derive an analytical model for the number of faults for an arbitrary level of step fusion. Results are presented in Figure 5.3

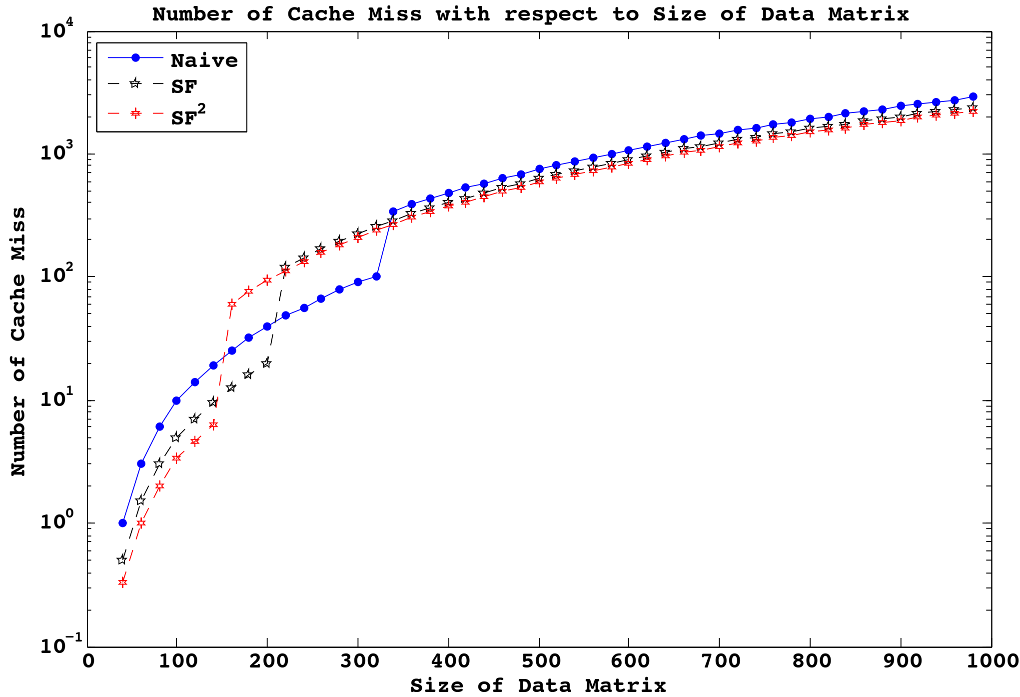


**Figure 5.3:** Number of cache miss (normalized with respect of the number of time steps) for different size of the data matrix and a fixed number of cache locations (1k).

We can see there is a significant reduction of cache miss for small sized matrices (case I: working set of innermost loop fit in cache) because the $\mathcal{SF}$ reduces the number of faults by a factor $i$ equal to the $\mathcal{SF}$ factor. For higher dimensionality of the matrix (case II: working set of innermost loop does not fit in cache), the gain is reduced because the number of cache miss decreases only by a factor $\frac{\psi}{3i} = \frac{2i+1}{3i}$ which can be easily derived from the previous analysis. Notice that for intermediate values of cache size, the step fusion method performs worse than the naive version. This is because the working set of the $\mathcal{SF}$ version is bigger; therefore, for some matrices, it does not fit in the cache while the working set of the naive version does (Figure 5.4).

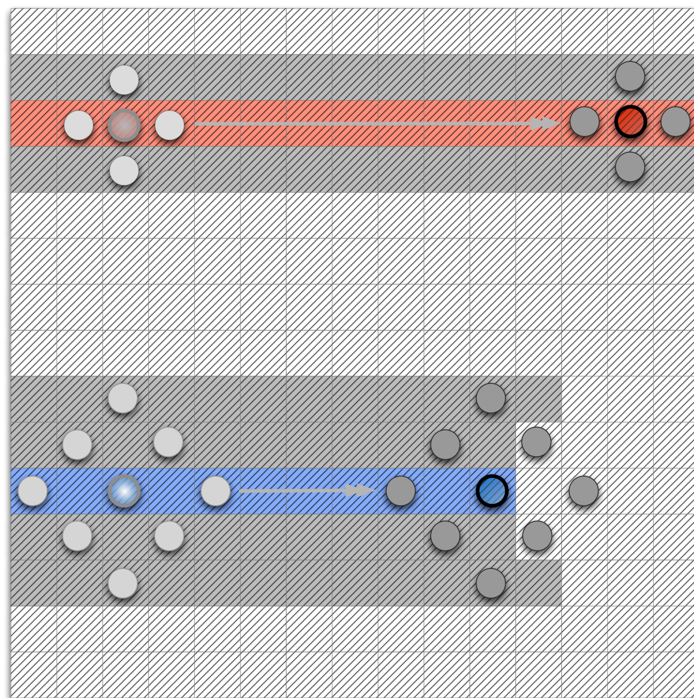

**Figure 5.4:** Elements accessed by the naive and the $\mathcal{SF}$ version of the jacobi 2d stencil. Red elements are updated by the naive version. Blue elements are updated by the $\mathcal{SF}$ version.

The $\mathcal{SF}$ trasformation is "representative" of the difficulties of the analysis of optimizations:

A. It is defined at the highest level of abstraction for a limited subset of computations; therefore, it cannot be automatically applied by a general purpose compiler.

B. Its effects are estimated only by considering multiple levels of the hierarchical model.

C. It affects both parallelism ( reducing the time spent in communications throughout the entire computations ) and sequential performances ( reducing, in most cases, the number of cache misses ).

D. The benefits for the sequential version depends on the problem size.

In conclusion, we have a complex trade off among different factors which concur to performance. By using the step fusion, we diminish communication among processes ( also with oversending ) and exploit more the memory hierarchy, but the number of operations performed for every single element is increased. For both oversending and $\mathcal{SF}$ methods, there exists an optimum number of time steps to be merged depending on the stencil properties and the model parameters (which in turn represent the real architecture).

## 5.3 Classic Optimization Methods

In this section, we analyze the optimizations methods usually exploited by compilers. We are excluding optimizations on the object code, which are strictly specific of the target architecture, and we concentrate on high level optimizations. More specifically, we will analyze optimizations that targets loop nests since they are the typical part of a computation where most of the time is spent (this is obviously the case for structured grid computations). Following the taxonomy presented in[5, 2], we identified three classes of optimizations:

1. Data Flow optimizations
2. Loop restructuring
3. Loop reordering

The class of Data Flow optimizations analyzes the "history" of the variables in order to substitute some piece of code with more efficient ones. Conceptually, the code is executed at compile time and the compiler eliminates unreachable code and substitutes pieces of code with less costly ones.

```
for i =1..n
    a[i] = a[i] + c*i
```

**Listing:** Original code

```
T = c
for i = 1..n
    a[i] = a[i] + T
    T = T +c
```

**Listing:** Code after data flow analysis

This particular optimization, for variables which are updated with functions of loop indices, is called loop-based strength reduction. Another very simple example is the following:

```
for i =1..n
    if (j>0)
    a[i] = a[i] + c
```

**Listing:** Original code

```
if (j>0)
for i = 1..n
    a[i] = a[i] + c
```

**Listing:** Code after data flow analysis

This class of optimizations may modify to a great extent the structure of the code and it is used widely by compilers. However, for the case of structured grids, it is probably ineffective. It might modify the assignment of temporal variables and the order of arithmetic instructions, but it cannot prevent cache misses. Given the peculiarities of structured grids computations, we will concentrate on optimizations intended to reduce the burden on the memory system.

The next class of optimization is loop restructuring, which is a generalization of loop unrolling. The loop structure is extensively modified in order to reduce loop overhead by decreasing the number of branch instructions. Moreover, by enforcing the reuse of the cpu registers inside a loop, it diminishes load and store overhead. Nonetheless, the order of instructions is unaffected. Even though the number of load and store instructions is reduced, the same number of elements has to be fetched from memory in the same order. The application of loop unrolling to a mono dimensional laplace operator is presented in Figure 5.4. Branch conditionals are executed half of the time in the unrolled loop. The number of load and store operations is 4 in the first loop and only 6 in the second one. If there is a sufficient number of cpu registers, it is possible to increase the level of expansion and

eventually unroll multiple nested loops (unroll and jam[2]). However, this does not reduce cache miss since the elements accessed by the two versions are the same and are in the same order.

```
for i =1..n
    b[i] = a[i] + a[i−1] + a[i+1]
```

**Listing 5.3:** Code of a mono dimensional laplace operator

```
for i = 1 .. n ,2
    b[i] = a[i] + a[i−1] + a[i+1]
    b[i+1] = a[i+1] + a[i] + a[i+2]
```

**Listing 5.4:** Code unrolled

Therefore, the class of loop reordering transformations is the only valid candidate for our needs. In fact, the contribution of other methods is not tangible at the abstraction level of the external memory model.

## 5.4 Loop Reordering Optimizations

Loop reordering techniques are an important class of optimizations that change the relative order of execution of the iterations of loop nests[5]. This class contains the optimizations that tries to exploit the temporal locality of the computation. In order to do so, statements that access the same memory locations are moved closer to one another so that the required values are almost certainly in the lower levels of the memory hierarchy. This reduces the number of cache miss throughout the execution of the whole computation and obviously increases performances.

In order to better explain these techniques, we will present some examples:

```
for j = 1 .. N
    for i = 1 .. N
        total[j] += A[i,j]
```

**Listing 5.5:** Pseudocode that computes the sum of the columns in a matrix. We are assuming that the matrix A is stored in memory in row order.

Notice that accesses to the array total do not cause cache misses. On the other hand, the matrix A is accessed in a stride $N$ pattern[2]. Thereby, if the cache cannot store $N$ different blocks at the same time, the whole computation will incur in $N^2$ faults. In this case, the following version is more efficient:

```
for i = 1 .. N
    for j = 1 .. N
        total[j] += A[i,j]
```

[2] There is a distance N between two elements accessed consecutively.

**Listing 5.6:** Pseudocode of the Loop Interchanged version of the algorithm shown in Listing 5.5

The number of faults caused by matrix A is reduced from $N^2$ to $N^2/\mathbf{B}$. Notice that the vector total has to be loaded from memory, leading to a maximum of $N^2/\mathbf{B}$ additional cache misses. The technique here presented is called Loop Interchange. When applicable, it is typically used to exploit spatial locality of an algorithm which is hidden by an unfavourable ordering of the loop nests. Similarly to the $\mathcal{SF}$ trasformation , its effectiveness depends on the size of the data structures.

Consider now the matrix vector multiplication:

```
int A[M][M], B[M],C[M];
for ( i=0; i<M; i++){
    for( j=0; j<M; j++){
        C[i] += A[i][j] + B[j]
```

**Listing 5.7:** Pseudocode of the matrix vector multiplication

By analyzing its memory access pattern, we realize that this algorithm has a high spatial locality that cannot be improved since A,B,C are accessed in a unit stride. This algorithm also has substantial reuse because every element of array B is accessed M times. If B fits in cache, then the number of memory accesses is minimal since every block of matrices A,B and C is accessed only once. Otherwise, at every iteration of the outermost loop, the array B has to be accessed from memory leading to $M^2/\mathbf{B}$ faults. Since the statements in the innermost loop can be performed in any order, we could try to rearrange them so accesses to the same location of B are less spread apart in time.

```
int A[M][M], B[M],C[M];
for ( jj = 0; jj < M/b; jj++) {
    for ( i=0; i<M; i++){
        for( j= b*jj; j<(jj+1)*b−1; jj++){
            C[i] = A[i][j] + B[j]
```

**Listing 5.8:** Pseudocode of the matrix vector multiplication with innermost loop tiled. b is a multiple of block size **B**.

Notice that B is now accessed one block at a time and completely used before moving to the next block. Faults from accesses to vector B are reduced from $M^2/\mathbf{B}$ to $M/\mathbf{B}$. However, now we have to explicitly load vector C from memory and incur in $M^2/(\mathbf{B} * b)$ faults. Applying again the same technique on the other loop we obtain:

```
int A[M][M], B[M],C[M];
for ( ii = 0; ii < M/b1; ii++) {
    for ( jj = 0; jj < M/b2; jj++) {
        for ( i=ii*b1; i<(ii+1)*b1-1; i++){
            for( j= jj*b2; j<(jj+1)*b2-1; j++){
                C[i] = A[i][j] + B[j];
```

**Listing 5.9:** Pseudocode of the matrix vector multiplication with both loops "tiled".

For simplicity we assume that $b1 = b2$. Both array B and C produce $M^2/B * b$ faults. Notice that the accesses to the A matrix over time are grouped in small squares, which is where the name tiling comes from.

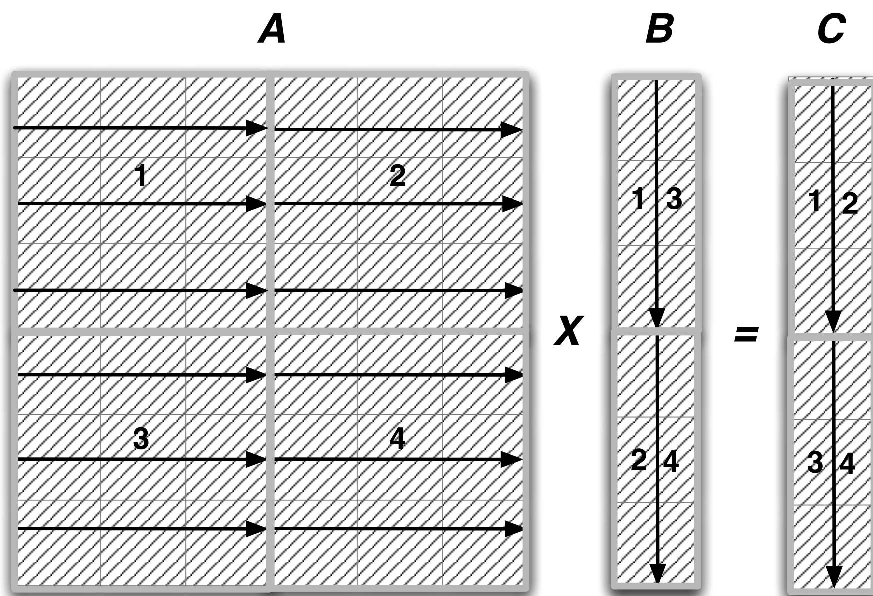


**Figure 5.5:** Pattern of accesses to matrix A of Listing 5.9

However, this is obviously not a formal definition. Moreover, the term tiling is misleading since a formal definition for tiling alone does not actually exist. Tiling is a particular pattern of access to data structures which is a consequence of a loop reordering transformation. In order to define loop reordering transformations, we need to define some concepts first:

**Definition 5.2** (Perfect Loop Nest)**.** *A perfect loop nest is a set of nested for loops where there are statements only in the innermost loop.*

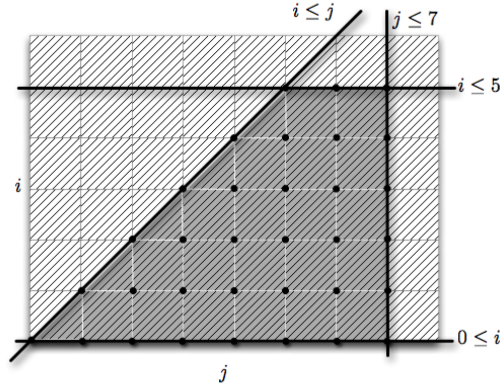


**Figure 5.6:** Convex polyhedron which represent the iteration space of Listing 5.10

The nested loops of the matrix vector multiplication in Listing 5.7 are a perfect loop nest.

**Definition 5.3** (Iteration Space). *Given a perfect loop nest composed by d nested loops, the iteration space of the loop nest is the set of all d-dimensional points* $(i_1, \cdots, i_d)$ *corresponding to all possible combinations of values of the loop indices during execution.*

In a perfect loop nest of $d$ loops, if all loop bounds are affine functions of outer loop indices, and if all loop variables are incremented by one at each iteration, then the iteration space can be represented by a convex polyhedron. Consider as an example the following code in Listing 5.10.

```
for ( i = 0; i <= 5; i++) {
    for ( j = i; j <= 7; j++) {
        A[i][j] = 0;
```

**Listing 5.10:** Generic for loop with loop conditional as affine functions.

Every loop bound is an affine function of outer loop indices (bounds based on constant, e.g., $i <= 5$, are obviously affine) and every loop index is incremented by one at every iteration. Therefore, the loop nest can be represented in a two dimensional space. Every loop bound divides the space in two half spaces of which one can be discarded. The intersection of valid half spaces produces the convex polyhedron shown in Figure 5.6. Such representation of perfect loop nests is called the **polyhedral model**.

By visiting the lattice of the polyhedron (Figure 5.6) in a lexicographic order we obtain the sequence of execution of points in the iterations spaces

of Listing 5.10. This sequence of points:

$$(0,0),(0,1),\cdots,(0,7),(1,1),(1,2),\cdots,(1,7),(2,2),\cdots,(5,5),(5,6),(5,7)$$

that represents the execution of the program is called the **schedule** of the program.

**Definition 5.4** (Loop Reordering Transformation)**.** *A loop reordering transformation of a program P is a permutation of its schedule.*

After having formally defined the concept of schedule and iteration space, it was easy to define a loop reordering transformation. Notice that loop interchange and loop tiling optimizations are a specific instance of loop reordering. Other optimizations in this class are: loop fission, loop fusion, loop reversal, loop coalescing and loop collapsing[10, 5]. They are all specific case of loop reordering.

Similarly to Section 3.1, we want to find transformations that preserve the semantics of the program and increase performance. Therefore, the compiler should acquire the following information in order to apply loop reordering transformations:

1. Data reuse: sets of iterations that access the same data.

2. Data dependencies: if two statements in the iteration space access the same memory location and one of the two is a store operation, then their relative order cannot be changed.

This information is stored by the compiler as a set of affine functions that associate points in the iteration space (the polyhedron lattice) to the coordinates of accessed data structures.

As an example consider the matrix vector multiplication in Listing 5.7. The affine array accesses of the inner most statement are:

$$\left\langle C, \begin{pmatrix} 1 & 0 \end{pmatrix}; A, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; B, \begin{pmatrix} 0 & 1 \end{pmatrix} \right\rangle \tag{5.1}$$

The first affine function indicates the element that has to be updated. More precisely, at iteration $\begin{pmatrix} i \\ j \end{pmatrix}$ the element $\begin{pmatrix} 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix}$ of matrix C has to be updated. In the 2d jacobi stencil, the affine array indexes (of vector A) are:

$$\left\langle \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\rangle \tag{5.2}$$

A polyhedral compiler[3] conceptually test all possible schedules, i.e., all possible permutations of the iteration space. Firstly, it checks that all data dependencies are preserved. If this fails, the schedule is discarded. If the schedule preserves the data dependencies, then a cost is associated to the schedule. The cost of every admissible schedule is computed as a function of the distance between memory accesses to the same location. Finally, the less costly schedule is chosen. In a real compiler, it is obviously impossible to test all the possible schedules so heuristics are used to derive admissible schedules and to choose the best one. We will not discuss further the difficulties of automating this process. The theory is well established and most of the research is related to produce good heuristics. However, it is interesting to note that no mainstream compiler utilizes the polyhedral model because of its numerous limitations:

I. Perfect loop nests.

II. All functions must be pure functions with no side effects.

III. All loop indexes must have unit stride.

IV. Loop bounds must be affine functions of outer loop indexes.

We will now concentrate on how loop reordering techniques can be applied to structured grids computations. Consistent with the concepts in Chapter 3, we will see that restricting the scope to a subclass of computations allows us to define very efficient transformations that could not have been derived by an automatic approach for generic computations.

## 5.5 Loop Reordering for Structured Grids Computations

In order to exploit loop reordering techniques, we study the data reuse and data dependencies of structured grid computations. Firstly, consider that there are no data dependencies inside a time step for a structured grid computation. Therefore, any schedule that does not alter the relative order of updates of different time steps is legal. In other words, the iteration space of a single time step can be permutated in any possible order. On the other hand, reuse inside a time step is quite limited since it is proportional to the number of shape points. Consider a generic element of a two dimensional mesh where the jacobi stencil is executed. This element is will be used only four times (the cardinality of the shape) in a single time step (Figure 5.7).

[3] A compiler able to reconstruct a polyhedral model out of loop nests.

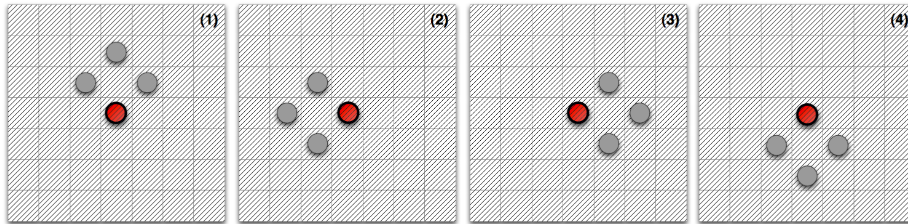


**Figure 5.7:** Reuse of a generic element of the mesh in a jacobi 2d stencil.

Compared to the matrix vector multiplication, where each element is accessed a number of times proportional to the size of the vector, there are limited opportunities for cache reuse in structured grid computations. Nonetheless, it is possible to exploit this reuse by reordering the loop and modifying the order in which updates inside a time step are performed (Section 5.5.1).

A more aggressive strategy is to reorder all the loops including the outermost loop of the time steps. Therefore, updates of different elements of the data matrix are **skewed** with respect to time steps. For example, it is possible to update elements at time step $s + 4$ when there are still elements that need to be updated at time step $s$. The rationale behind this method is to reuse updated elements as soon as possible[45] thus reducing sensibly the number of faults with respect to the non skewed reordering (Section 5.5.2). Notice that skewing is not always possible for real world stencil computations. Multigrid computations (Section 2.3) do not perform the exact same operation on all time steps; therefore, skewing cannot be applied[34].

## 5.5.1 Loop Tiling

In this section, we analyze the effects of loop reordering inside a time step. We already examined the two dimensional jacobi stencil by deriving the impact of the $\mathcal{SF}$ trasformation for a memory hierarchy. We have shown that in order to update a row of the matrix (Listing 5.1), multiple rows are loaded in cache; the number of rows is proportional to the extent of the shape. If a single row can not fit in cache, the entire working set (consisting of multiple rows) has to be loaded in cache at every iteration even though part of it was present in cache at the previous iteration. Therefore, we update only portions of one row so that the whole working set can fit in cache. Then we move to the next row and update it. This is repeated until we have completely updated a column-wise portion of the matrix(Figure 5.8).

Unlike matrix vector multiplication that featured reuse of the same ele-

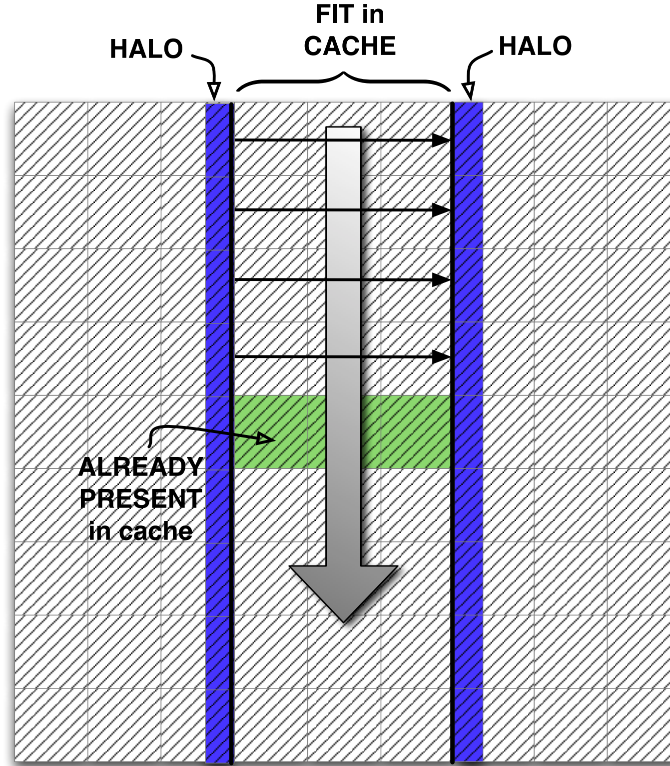


**Figure 5.8:** Graphical depiction of the optimized schedule of the two dimensional jacobi stencil. Notice how elements used to compute the next row are already present in cache. Notice also the "halo" of elements around the examined tile which are loaded from memory twice.

ments throughout the entire computations, here reuse is limited to a small portion of the matrix. Therefore, once a row is updated, we proceed column-wise since some elements of the working set of this iteration can be reused at the next iteration. By doing so, we also limit the number of "tiles" that has to be computed. If we analyze the performance of the algorithm on the external memory model for a generic stencil of extent $\psi$, we have that faults are reduced from $\psi * M^2/\mathbf{B}$ to approximately $M^2/\mathbf{B}$. In practice every element is loaded from memory only once (which is optimal) except for elements around tiles that are loaded twice. Notice that, although reuse is very limited, we are able to reduce cache miss up to a value really close to the theoretical lower bound. This technique can be generalized in $d$ dimensions and was presented in[34] for three dimensional stencils. In the case of three dimensional stencils, e.g. the three dimensional Laplace operator (Figure

5.9a), sub planes that fit in cache are updated and then we move to the sub plane above it until an entire column has been updated (Figure 5.9b).

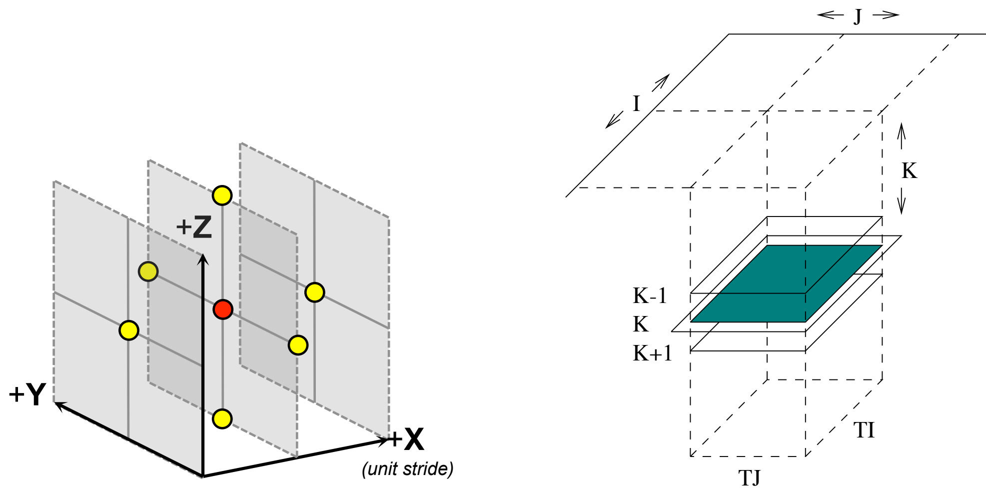


**Figure 5.9:** Figure 5.9b shows a graphical depiction of the pattern of updates in a optimized schedule for the 3d Laplace stencil shown in Figure 5.9a.

### 5.5.2 Time Skewing

In the previous section, we presented a loop reordering technique that reduces the number of cache misses by a factor proportional to the shape extent. This experimentally proved our claims that there are limited opportunities for cache reuse in stencil computations. Therefore, we now present an optimization technique that sensibly decreases the number of faults by reordering the schedule not only on the spatial dimension, but also along the temporal dimension. As we anticipated in Section 5.5, the idea beyond time skewing is to reuse values in cache as often as possible[24]. Therefore, consider a monodimensional laplace stencil. If the elements of the grid $A_{i-1}^{(k)}$,$A_i^{(k)}$,$A_{i+1}^{(k)}$ have been updated the next element updated will be $A_i^{(k+1)}$ instead of $A_{i+2}^{(k)}$. In order to have the maximum reduction of cache misses, portions of the time space (cache blocks) are isolated. No additional data structures are needed to store the values of the working domain at different time steps ( Figure 5.10).

The execution of different cache block has to be performed in the order specified in Figure 5.10. Notice that the steepness of the slope depends on the order of the shape.

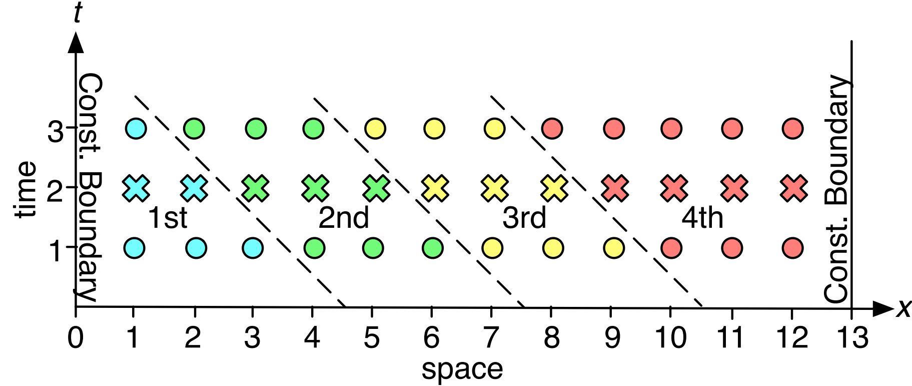


**Figure 5.10:** Graphical depiction of time skewing applied to a monodimensional laplace stencil. The circles and crosses indicate in which data structure the values are saved.

**Definition 5.5** (Order of a Stencil)**.** *Given a shape of a d-dimensional stencil* $\mathcal{S} = (s_0^0, \cdots s_d^0), \cdots, (s_0^n, \cdots s_d^n)$ *composed by n points,*

$$\eta = max(|s_0^0|, \cdots, |s_d^0|, \cdots, |s_0^n|, \cdots, |s_d^n|)$$

.

In Figure 5.4, the order of Jacobi is 1 while the order of the Jacobi $\mathcal{SF}$ version is 2. Therefore, the amount of time steps that can be executed inside a cache block depends both on the characteristics of the stencil shape and the space tiling. An analysis on the theoretical reduction of cache miss can be found in [20, 38]. The conclusions are that the number of faults, for a generic $d$-dimensional stencil are reduced by a factor proportional to $\mathbf{C}^{\frac{1}{d}}$ where $\mathbf{C}$ is the size of the cache. The best result obtained with Tiling, for an arbitrary number of time steps $T$, was $\Theta(TM^d/\mathbf{B})$ while time skewing can ensure $\Theta(TM^d/(\mathbf{B} * \mathbf{C}^{\frac{1}{d}}))$. The proof in [20] is elegant, but also overly complex and it provides asymptotic results. We will give an alternate (still rigorous) proof that explains how this reduction of cache misses is obtained and presents a more accurate (instead of asymptotic ) estimate of the number of faults.

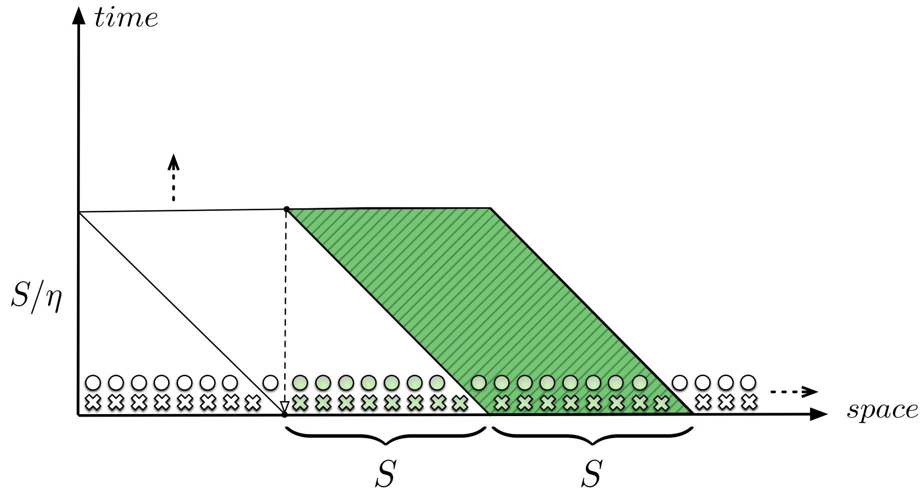


**Figure 5.11:** Area of a trapezoid (cache block) for a generic mono dimensional stencil (in green). The memory requirements of the stencil are the light green points under the projection of the trapezoid. $\eta$ equals one for this particular stencil.

**Theorem 5.1** (Reduction of cache misses in time skewing codes)**.** *Consider a d dimensional structural grid computation defined on a d dimensional domain of size $M^d$. Assume that both the number of time steps T and the subsection of the domain of size $M^{d-1}$ are greater than the size of the cache* $\mathbf{C}$ *($T = \Omega(\mathbf{C})$).*

*Then, the number of cache miss of the time skewed version is* $(2\eta)/(\frac{\mathbf{C}}{2})^{\frac{1}{d}}$ *times the number of cache miss of the naive version and so the gain is* $\Theta(\mathbf{C}^{\frac{1}{d}})$*.*

*Proof.* We first consider the mono dimensional case (see Figure 5.11). We want to reuse as much data as possible, so we divide the space-time in a series of trapezoids that fit precisely in cache and have maximum height. These"optimal" trapezoids have a base of length $S$ and a height of length $S/\eta$. We could erroneously conclude that the space required in cache by a trapezoid corresponds to its area; however, the number of cache locations equals to four times $S$ (its base). This is because the working domain (at different time steps) is stored using only two vectors; therefore, only the projection of trapezoid of size equal to two times $S$ is stored.

(1) A trapezoid repeatedly updates a portion of these two vectors of size equal to two times $S$ (Figure 5.11). By doing so, it loads in cache $S$ elements while $S$ elements are written back in memory.

(2) By imposing that the whole trapezoid must fit in cache, we obtain $\mathbf{C} =$

$4S$.

(3) The number of time steps $\mathcal{T}$ where the trapezoid is defined corresponds to $S/\eta$.

By combining (1),(2),(3) we have that a single sweep using trapezoids produces $M/B$ faults and writes back $M/B$ blocks, exactly as for the tiled version. However, $S/\eta$ time steps are fused together, thus linearly reducing the number of faults by a factor $S/\eta$ throughout the entire computation. Moreover, $S/\eta$ can be rewritten as $\mathbf{C}/(4 * \eta)$, giving the linear relation between reduction of faults and size of the cache.

We can generalize the above proof for the $d$ dimensional case. The area of a trapezoid in $d$ dimension, assuming the space is divided in equal blocks, is $S^d * S/\eta$ where $S/\eta$ is the height of the trapezoid and it corresponds to the performance gain. Now we have that $2 * (2S)^d = \mathbf{C}$ therefore $S = \frac{1}{2}(\frac{\mathbf{C}}{2})^{\frac{1}{d}}$. Moreover, $S/\eta = (\frac{\mathbf{C}}{2})^{\frac{1}{d}}/(2\eta)$ which obviously implies that the gain is $\Theta(\mathbf{C}^{\frac{1}{d}})$. □

Notice that assumption of $T = \Omega(\mathbf{C})$ and $M = \Omega(\mathbf{C})$ were made in order to ensure that the optimal trapezoid fits in cache. The upper bound on the memory requirements of a trapezoid is $2 * (2S)^d$ because we have two replicated data structures and the projection of a $d$ dimensional trapezoid is less than or equal to $(2S)^d$ ( Figure 5.12). Figure 5.12 shows that different trapezoids update different points in a 3d space time lattice for a generic two dimensional stencil with $\eta$ equal to one (projected on the two dimensional spatial grid). An example of a three dimensional stencil is presented in [24].

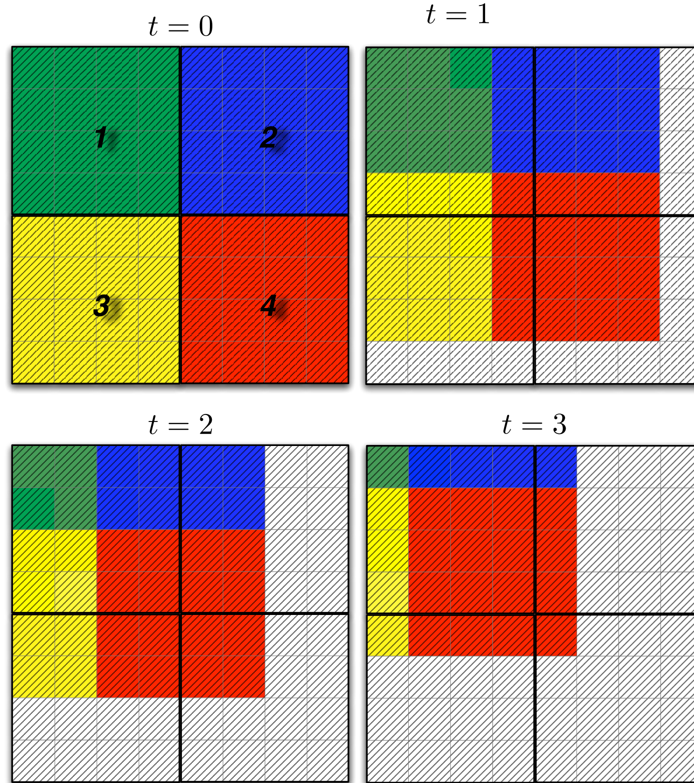


**Figure 5.12:** Projection of a 3d space time lattice for a skewed 2 dimensional stencil with order equal to one. Different colors corresponds to different trapezoids. Different frames corresponds to different planes at different time steps. The numbering shows the order in which trapezoids has to be executed while the grey area indicates portions of space time that has to updated by other trapezoids.

### Comparison of Tiling and Skewing

In conclusion, we analyzed both tiling and skewing under the assumption that the entire working set can not fit in cache. We have shown that tiling can reduce cache misses by a factor $\psi$ where $\psi$ is the extent of the shape thus, producing a number of cache misses inside a time step close to the optimal value of $M^d/\mathbf{B}$. Including the write back of elements in memory, we have a total of $2 * M^d/\mathbf{B}$.

Skewing is far more efficient theoretically since it reduce cache misses by an additional factor $\mathbf{C}^{\frac{1}{d}}$. We have shown that this gain is obtained by executing optimal portions of the space time domain. We assumed, not only that the number of grid points does not fit in cache, but also the number

of time steps is greater than the size of the caches. If this is not the case (Figure 5.13) a "time cut" is necessary. All time steps can be computed with

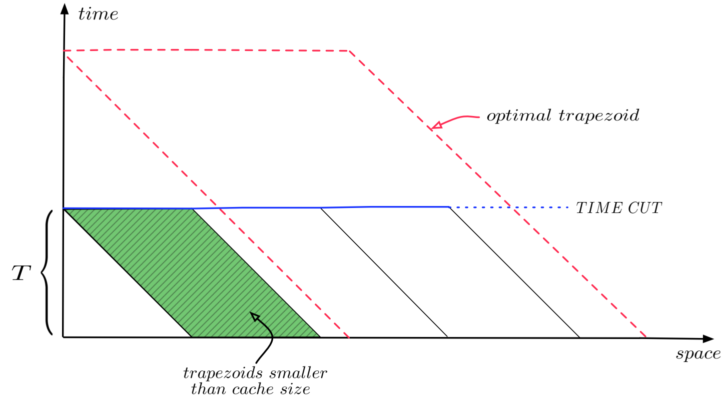


**Figure 5.13:** Time skewing that has to be reduced because the number of time steps is not sufficient. The optimal trapezoid (dotted) would have been substantially bigger than the actual trapezoids (shaded)

a single sweep of the data structures, yielding $2 * M^d/\mathbf{B}$ cache miss which is optimal. However, not all structured grids computations are amenable to skewing, e.g. multigrid computations[34].

### 5.5.3 Cache Oblivious Algorithms for Structured Grids Computations

It is evident the importance of the size of the cache **C** for both tiling and skewing optimizations. However, caches of a real machine are not as ideal as the cache of the model:

1. Caches might not be private.
2. An optimal replacement strategy is not used.
3. Caches are not completely associative.

Therefore, when tuning the algorithm with respect to the cache size **C**, it is better to be conservative. In fact, degradation of performance caused by an incorrectly guessed parameter might render optimizations counter productive. Also a real memory system is hierarchical so we should also apply these optimizations multiple times in order to optimize every level. Therefore, we need to choose a good parameter for the size of the available cache at every step.

Cache oblivious algorithms take advantage of cpu caches without having the size (or other details) of the cache as a parameter. Cache oblivious algorithms for stencil computations are presented in [20, 37]. These algorithms utilize time skewing so it is applicable only to stencil computations that allow time skewing.

Similarly to Section 5.5.2, When using the time skewing optimization, the space time is divided in trapezoids . The idea behind the cache oblivious algorithm is the following: Given a generic trapezoid $\mathcal{T}$ that has to be computed (Figure 5.14) we divide it in two trapezoids $\mathcal{T}_1$ and $\mathcal{T}_2$ to then recursively iterate the process over $\mathcal{T}_1$ and then $\mathcal{T}_2$. When the examined trapezoid has a height of only one time step, it is executed. A trapezoid can be divided in two ways:

1. A space cut if the width is at least twice the height (Figure 5.15a).

2. A time cut otherwise (Figure 5.15b).

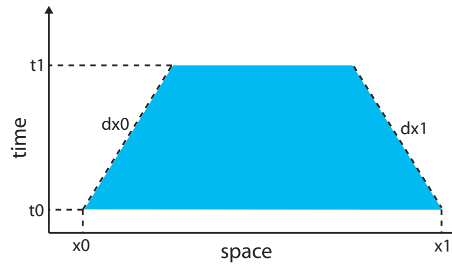


**Figure 5.14:** A generic trapezoid. As usual, the two slopes are derived from the data dependencies of the computation.

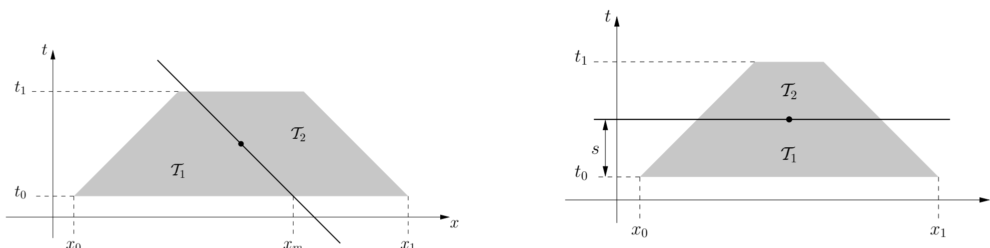


**Figure 5.15:** Figure 5.15a shows a graphical depiction of a space cut. Figure 5.15b shows a graphical depiction of a time cut.

The $n$ dimensional version is simply obtained by dividing first in any spatial dimension and then in the time dimension. The idea beyond this algorithm is quite simple: by recursively dividing the trapezoid, we will obtain (for a given level of recursion) trapezoids whose size is **almost** optimal. Moreover, we can mitigate the negative effects due to shared caches and optimize for all the levels of the hierarchy. It has been proved in [20, 38] that this algorithm asymptotically incur in $\Theta(M^d/\mathbf{C}^{\frac{1}{d}})$, which is the same result that we obtained for the time skewing algorithm. In practice, the number of cache miss of this algorithm is higher than the number of cache miss of a time skewed stencil code, finely tuned for a specific architecture and a specific data size. However, this algorithm should consistently deliver almost theoretical peak performances on any architecture and data size without any specific tuning.

# Chapter 6

# Combining Locality and Distributed Memory Optimizations

In this chapter, we compare the optimization methods presented in Chapter 4 and 3. Moreover, we explain how these methods can be applied together. We already discussed how optimization methods defined at the level of functional dependencies ($\mathcal{Q}$ trasformations and $\mathcal{SF}$ ) can be combined with methods defined at the concurrent level (Chapter 4). Now we can also add optimizations for locality presented in the previous Chapter. Since optimizations described in Chapter 4 are defined specifically for structured grids computations, we will limit our discussion to this class of computations.

## 6.1 Combining $\mathcal{SF}$ and Loop Reordering

In this section, we analyze the interaction of $\mathcal{SF}$ and loop reordering transformations. Similarly to Chapter 4, $\mathcal{SF}$ and loop reordering can be combined freely since they work at different levels of abstraction. Firstly, $\mathcal{SF}$ is applied producing a modified shape, then when the computation has been translated at the concurrent level, Loop Reordering is applied.

### 6.1.1 $\mathcal{SF}$ and Tiling

In order to discuss the effects of combining $\mathcal{SF}$ and loop reordering transformation, we firstly have to define the final output of the combination of the two optimization methods. Consider the structure of a generic structured grid code defined at the concurrent level (Figure 6.1).

```
double J_in[M][M], J_out[M][M];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step; i_step + +){

  SEND(P_(-1,0)); SEND(P_(0,-1));
  SEND(P_(0,1)); SEND(P_(+1,0));
  SEND(P_(-1,-1)); SEND(P_(+1,-1));
  SEND(P_(-1,+1)); SEND(P_(+1,+1));

  COMPUTE( incoming independent region );

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

  COMPUTE( incoming dependent region );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Listing 6.1:** Pseudocode representation of a generic structured grid computation defined at the concurrent level. The working domain is defined in two dimension and has size $MxM$. Depending on the stencil shape some of the SEND and RECV operations might not be necessary.

The part labelled as COMPUTE( incoming independent region ) is in fact a generic d-nested loop where $d$ is the dimensionality of the computation. For the Jacobi Stencil (or a generic two dimensional stencil of *order* 1), the following nested loops compute the Incoming Independent region of a working domain of size $N$:

```
for i = 1 .. N-2 do
    for j = 1 .. N-2 do
        B[i,j] = 0.25*( A[i-1,j] + A[i,j-1] + A[i+1,j] + A[i,j+1] )
```

**Listing 6.2:** Nested loops for the update of the incoming independent region of a two dimensional Jacobi stencil.

This is a perfectly nested loop therefore loop reordering can be applied. Since the **structure** of the concurrent level code is always the same (Listing 6.1), unregarding of the actual shape, we can apply loop reordering to any structured grid computation. In fact, consider the code for the incoming independent region of the $\mathcal{SF}$ version of Listing 6.2 (Listing 6.3). The structure is exactly the same. Notice that the only difference is the size of the incoming independent region which is reduced because the $\mathcal{SF}$ shape has a larger order.

```
for i = 2 .. N-3 do
    for j = 2 .. N-3 do
        B[i,j] = 0.0625*( A[i-2,j] + 2*A[i-1,j-1] +
        2*A[i-1,j+1] + A[i,j+2] + 4*A[i,j] + A[i,j-2] +
        2*A[i+1,j+1] +2*A[i+1,j-1] + A[i+1,j] )
```

**Listing 6.3:** Nested loops for the update of the incoming independent region of a two dimensional Jacobi stencil with $\mathcal{SF}$ applied.

This loop does not contain the time step so we are actually performing the tiling strategy. Moreover, there are no advantages in tiling the incoming dependent region because it has a limited size that will almost always fit in cache.

We will now analyze the impact of this additional modification on the memory hierarchy. We will consider the more general case where $M^{d-1}$ elements do not fit in cache. In Section 5.5.1, we have shown that the number of cache misses is reduced by a factor proportional to the extent $\psi$ of the stencil's shape. Therefore, tiling can be very beneficial in computations with applied $\mathcal{SF}$ since their $\psi$ is bigger than normal. However, a comparison should not be made between the tiled $\mathcal{SF}$ version and the original $\mathcal{SF}$ version because instead we should compare the naive and $\mathcal{SF}$ version when both are tiled. Since tiling ensures that the number of cache misses is reduced to

a number close to the theoretical optimum of $M^d/\mathbf{B}$ per time step, the $\mathcal{SF}^i$ version should perform $i$ times better because it reduces the number of time steps by a factor $i$.

Consider that, for high levels of $\mathcal{SF}$ , some factors concur to step fusion overhead:

I. The halo of a tile increases linearly with respect to the extent of the shape and therefore with respect to the level of step fusion (Figure 5.8). The assumption, made in Section 5.5.1, that the overhead introduced by halo elements is negligible for normal stencil's shape, might not apply anymore.

II. The size of a tile is reduced because the working set is increased, which in turn increases the number of tiles and exacerbates the problem in I.

III. The number of floating point operations also increases with respect to the level of step fusion.

If we consider I. and II. only (Figure 6.1), the number of cache miss is indeed higher than the optimal value but not substantially.

We can not address III. in a external memory model; however, experimental results[31] show that step fusion above a level of three is infeasible. In fact, the arithmetic intensity grows quadratically with respect to the step fusion level and thus dominates the computation time.

### 6.1.2 Comparison of $\mathcal{SF}$ and Time Skewing

Notice that $\mathcal{SF}$ and time skewing can be theoretically combined since they work on different levels; however, they both try to reduce the number of cache miss by collapsing the execution of different time steps. If we compare the two methods, we have that time skewing is far more efficient because it can collapse the execution of a number of time steps proportional to the cache size; therefore, reducing cache misses by the same factor. Step fusion can only merge effectively few time step but can also reduce the number of communications among workers. Therefore, the best method to apply depends on the peculiarities of the underlying architecture.

## 6.2 Combining $\mathcal{Q}$ trasformations and Loop Reordering

Similarly to Section 6.1, we discuss how we can combine $\mathcal{Q}$ trasformations and loop reordering techniques.

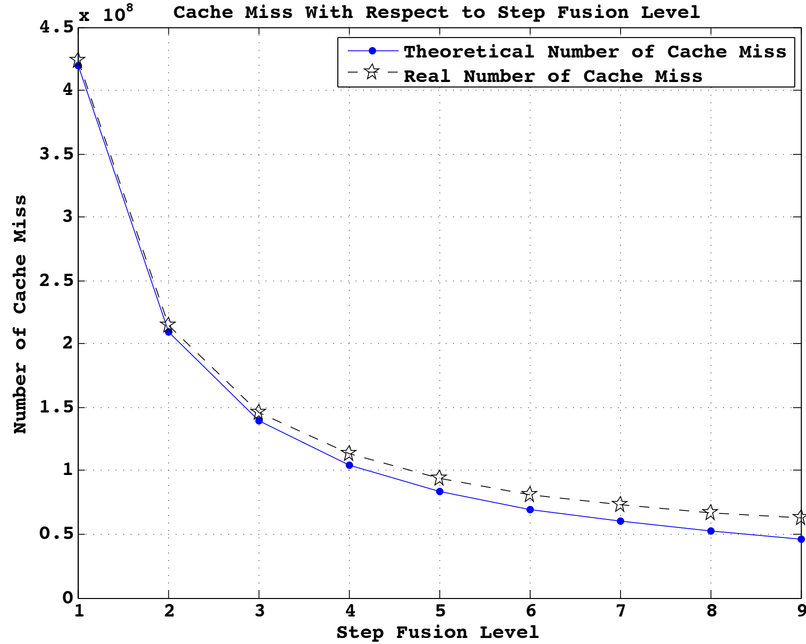


**Figure 6.1:** Number of cache miss with respect to step fusion level. The optimal value corresponds to $M^2/\mathbf{B}$. Matrix has a size of $2048x2048$ and the cache has a size of 1024 elements.

### 6.2.1 $\mathcal{Q}$ trasformations and Tiling

We will use the same argument of Section 6.1 and consider the structure of the code in Listing 6.1. The nested loop associated to the incoming independent region can be tiled as described in Section 5.5.1. Since $\mathcal{Q}$ trasformations have no impact on the memory hierarchy, the analysis of Section 5.5.1 is valid. Tiling will improve performance if the size of the cache is smaller than $\psi * M^{d-1}$. If this is the case, cache misses will be reduced by a factor equal to the extent of the shape $\psi$.

**$\mathcal{SH}$ method and Ghost Cell Expansion** Both methods operate at the concurrent level and modify the concurrent code produced. For convenience, we reproduce here the concurrent code of a generic structured grid computation with $\mathcal{SH}$ (Listing 6.4) or ghost cell expansion (Listing 6.5) that were introduced in Chapter 4.

For the shift method, we can easily apply tiling without causing any problems. Since the Incoming Independent region is divided in multiple parts equal to the number of spatial dimensions, it is sufficient to tile all the

```
double J_in[512][512], J_out[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step; i_step ++){

  SEND(P_(-1,0)); SEND(P_(+1,0));

  //update of first half of the Incoming Independent Region
  for i = 1 .. N-2 do
    for j = 1 .. N/2 - 1 do
      B[i,j] = 0.25*( A[i-1,j] + A[i,j-1] + A[i+1,j] + A[i,j+1] )

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

  copy(border elements of receive buffer into send buffers)

  SEND(P_(0,-1)); SEND(P_(0,+1));

  //update of second half of the Incoming Independent Region
  for i = 1 .. N-2 do
    for j = N/2 .. N - 2 do
      B[i,j] = 0.25*( A[i-1,j] + A[i,j-1] + A[i+1,j] + A[i,j+1] )

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

  COMPUTE( dependent incoming region );

  swap(J_in, J_out);
}
return_partition (J_out);
```

**Listing 6.4:** Pseudocode representation of a generic two dimensional stencil ($\eta = 1$) with shift method applied. Not all SEND and RECV operation might be necessary depending on the stencil's shape.

```
double A[512][512], B[512][512];
load_partition_values (J_in);
for(i_step = 0 ; i_step < num_step/2; i_step+ = 2){

  SEND(P_(−1,0)); SEND(P_(0,−1));
  SEND(P_(0,1)); SEND(P_(+1,0));
  SEND(P_(−1,−1)); SEND(P_(+1,−1));
  SEND(P_(−1,+1)); SEND(P_(+1,+1));

  for i = 1 .. N−2 do
    for j = 1 .. N − 2 do
      B[i,j] = 0.25∗( A[i−1,j] + A[i,j−1] + A[i+1,j] + A[i,j+1] )

  RECV(P_(−1,0)); RECV(P_(0,−1));
  RECV(P_(0,1)); RECV(P_(+1,0));
  RECV(P_(−1,−1)); RECV(P_(+1,−1));
  RECV(P_(−1,+1)); RECV(P_(+1,+1));

  COMPUTE( incoming dependent region);

  COMPUTE( external elements ∈ dependency set );

  //Second Step

  swap(A,B);

  for i = 0 .. N−1 do
    for j = 0 .. N − 1 do
      B[i,j] = 0.25∗( A[i−1,j] + A[i,j−1] + A[i+1,j] + A[i,j+1] )

  swap(A,B);
}
return_partition (A);
```

**Listing 6.5:** Pseudocode representation of a generic two dimensional stencil ($\eta = 1$) with oversending method applied. Not all SEND and RECV operation might be necessary depending on the stencil's shape.

resulting nested loops. However, in order to implement the tiling strategy presented in Section 5.5.1, we cannot divide the dimension of maximum stride of the Incoming Independent region. For the oversending method, we can also easily apply tiling. At the second step, the entire partition can be computed so the only difference is that the lower and upper bounds of the nested loops are bigger.

### 6.2.2 $\mathcal{Q}$ trasformations and Skewing

Using skewing in combination with the optimization methods presented in Chapter 4 is more complex. This is because of two factors:

1. The time skewing algorithm is strictly sequential; different trapezoids are updated in a wavefront pattern (Figure 5.12).
2. If we consider the loop over time steps in Listing 6.1, we do not have a perfect loop nest.

Therefore, we can not rely on automatic methods and we have to investigate how time skewing can be performed in parallel over a partitioned domain. Given a mono-dimensional domain, the execution of a time skewed algorithm can be represented as in Figure 6.2. Trapezoids must be executed in a given sequential order to preserve the correctness of the computation. It is possible to isolate trapezoids that can be executed in parallel as in Figure 6.3[21, 38]. We can force a trapezoid to have the base (spatial domain) equal to a partition in order to map it to a processing node. Then, the processing node can further divide the assigned trapezoid in order to exploit locality.

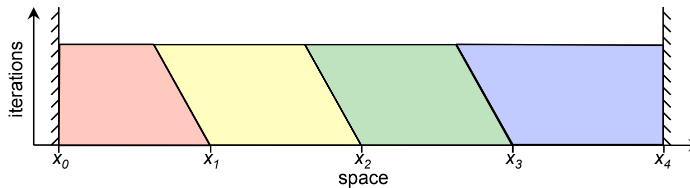


**Figure 6.2:** Graphical depiction of time skewing on a mono-dimensional stencil. Different trapezoids should be executed from left to right.

It is clear that different trapezoids can be updated in parallel: the base of a trapezoid will correspond to the Incoming Independent region and the number of time steps will be reduced by a constant factor which is tunable. However the portion of the Figure 6.3 labelled as clean up work should

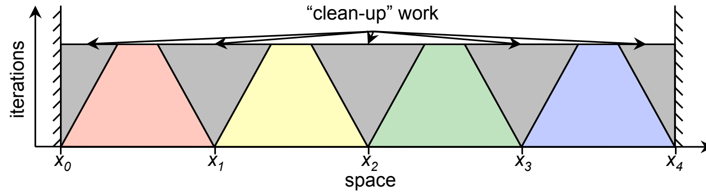


**Figure 6.3:** Graphical depiction of parallel time skewing. Different trapezoids belongs to different workers. The space-time portions labelled as "clean up work" should be updated after the adjacent trapezoids.

be computed concurrently by both processors who own the adjacent trapezoids. In order to do so, the borders (outgoing dependent region) should be sent for every collapsed time step (as in the naive version). Therefore, we have greatly reduced cache misses but the number of communications is unchanged. Obviously we can use the $\mathcal{Q}$ trasformation to reduce the number of communications and ameliorate the problem. Moreover, by using $\mathcal{QW}$ trasformation , we can use a single data structure to store all the data thus reducing the memory requirements of a trapezoid.

However, we should try to reduce the number of communications. Figure 6.4 shows a possible solution to this problem. In this case, partitions partially overlap in such a way that after performing communications two adjacent processing nodes can proceed without further synchronizations. This solution was proposed in [24, 14] as the circular queue method. However this is equivalent to the ghost cell expansion method[17]. Computation is divided in supersteps composed of previous time steps. The amount of data transferred among processes is unchanged but the number of communications is not proportional anymore to the number of collapsed time steps. Unfortunately, additional work has to be performed by every node (as in the oversending method).

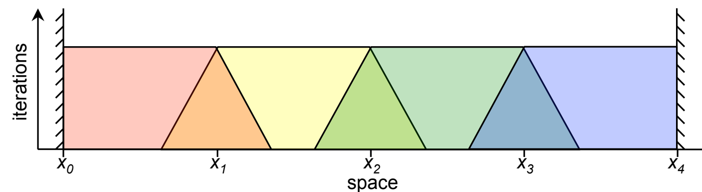


**Figure 6.4:** Circular Queue optimization of a time skewed mono dimensional stencil.

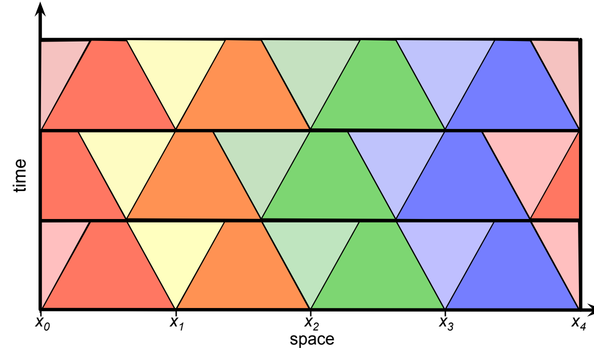


**Figure 6.5:** Execution of three steps of the proposed modified version of time skewing. Four different processors (red, orange, green and blue) update different portions of the space time domain. The update of the inverted triangle of space time points between different trapezoids is assigned in a circular pattern. By alternating the direction of communications the correct position of the partitions is restored.

Again, the number of time steps to collapse is tunable. The best number of steps to collapse using time skewing is difficult to obtain. However, in a real architecture, we should expect it to be a fraction of the theoretical upper bound derived in Section 6.2.2.

We now present an additional optimization of the skewing method shown in Figure 6.3 that has both a minimum number of communications and no computation overhead. The idea is to send the border of the trapezoid after it has been computed (not before as in the circular queue method). By using the $\mathcal{QW}$ trasformation in conjunction with time skewing, only one data structure is used. Therefore, it is easy to send the whole border to an adjacent node which will in turn update the entire inverted triangle of "clean up" work. Assuming periodic boundary conditions, a circular pattern of communications can be established and therefore every node performs the same number of operations. Obviously there is no computation overhead. At the end of a superstep, a node have in its buffers a different partition than its assigned one. However by alternating the direction of communications among processing nodes, it is possible to restore every element in its correct position ( Figure 6.5 ).

## 6.3 Conclusions

Throughout this thesis, we analyzed many optimizations techniques. Some of them are specific to structured grids computations (Chapter 4) while others are applicable to any scientific computation (Chapter 5). In Chapter 4, we presented the class of $\mathcal{Q}$ trasformations that together with the $\mathcal{SH}$ method ensure the lowest theoretical number of communications among processes for a generic $d$ dimensional stencil. Moreover, we analyzed the $\mathcal{QW}$ trasformation class of optimizations that reduce the memory requirements to an optimal value while maintaining a lower bound on the number of communications. We also presented the ghost cell expansion method and the $\mathcal{SF}$ trasformation . Both methods are capable of linearly reducing the number of cache miss, but they both introduce a computational overhead. The effect of the $\mathcal{SF}$ trasformation was thoroughly examined in Chapter 5 together with classical loop reordering transformations. We analyzed loop reordering transformations by taking into account that for some stencil computations, it is possible to collapse different time steps while for others, it is infeasible. In the former case, time skewing allows to dramatically reduce the number of cache miss. In the latter case, it is possible to reorder the computation inside a time step and by carefully selecting the schedule, achieve a number of cache faults close to the theoretical lower bound (for a single step). Finally in Chapter 6, we analyzed all the reasonable combinations of the aforementioned optimizations methods. We concluded that for stencil computations, where it is not possible to collapse different time steps, the combination of $\mathcal{Q}$ trasformation ,$\mathcal{SH}$ method and loop reordering limited to a time step will produce the best results. On the other hand, for computations where it is possible to collapse different time steps, we have compared $\mathcal{SF}$ and time skewing. We concluded that time skewing is probably more efficient in terms of performance, however a more accurate analysis on a real architectures is necessary. Finally, we examined how computations employing time skewing (which is strictly sequential) can be executed in parallel. We examined various solutions and derived, as a proof of concept, an alternative method that, unlike methods found in literature, has no computational overhead. All these optimizations were examined by following the methodology presented in Chapter 3, thus utilizing an abstract cost model in order to prove performance gain. In some cases, the limits of the model were evident (as in the case of the increase of flops in $\mathcal{SF}$ ) and we used experimental results to prove our claims. A meaningful testing of these methods should be performed on real architectures as a future work. Given the amount of methods presented, a comprehensive set of tests on a representative subset of all structured grid computations and a representative set of modern architectures will

be a very time consuming task although necessary. We performed a subset of all possible tests and we present some meaningful experimental results in the next chapter. Nonetheless, the structured model and the theoretical results presented throughout this thesis can be used as a reference in order to interpret experimental results and guide the optimization of structured grid computations.

# Appendix A

# Experimental Results

Tests were performed in order to validate theoretical results presented throughout this thesis. In order to perform these tests a specific compiler was developed in order to automatically produce optimized code with $\mathcal{Q}$ trasformations and/or $\mathcal{SF}$ trasformation and/or the $\mathcal{SH}$ method . In order to perform loop reordering optimizations we utilized the Pluto polyhedral compiler[8] in order to automatize the process of producing tiled nested loops. Tests were performed on the following machines:

**Andromeda**: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz multicore machine with two sockets. Every socket contains 4 cores with hyperthreading for a total of 16 hardware thread. OpenMPI 1.6 is used as the MPI library with gcc 4.6.2.

**Titanic** AMD Opteron(tm) Processor 6176 multicore machine with two sockets. Every socket contains 12 cores divided in two chips. No mechanisms for hardware multithreading is available therefore the number of hardware thread is 24. MPICH2 v1.4.1 is used as the MPI library with icc 12.1.0.

All reported results are the average of the completion time for a single time step (iteration) obtained as the average of 10 trials. All the tests do not consider scatter and gather operations since they are unaffected by the optimizations techniques presented. Moreover, all test are performed for a fixed parallelism degree and varying partition size, in order to test the effectiveness of different optimizations for different sizes of the working domain. In order to test the effectiveness of the $\mathcal{Q}$ trasformation and $\mathcal{SH}$ method we wanted to ensure that the following constrains were met:

I. Every partition has the same number of elements.

II. Every partition has the optimal surface to volume ratio (hypercubes), therefore the number of elements to transfer is minimal.

III. There are no communications from a partition to itself.

These constrains are necessary to ensure that communications are performed only when necessary and every partition has the same amount of elements to transfer. If the optimizations give a performance gain in this strict testing condition it is correct to assume that they will be even more effective when the amount of data to transfer is higher (or unbalanced) with respect to the partition size. If performance gain is proven under constrain II. it should be evident for other less desirable shapes where communications have even more impact on performance. Constrain III. is necessary to avoid communications that are useless (from a process to itself). In order to do so it is sufficient to have two partition per spatial dimension.

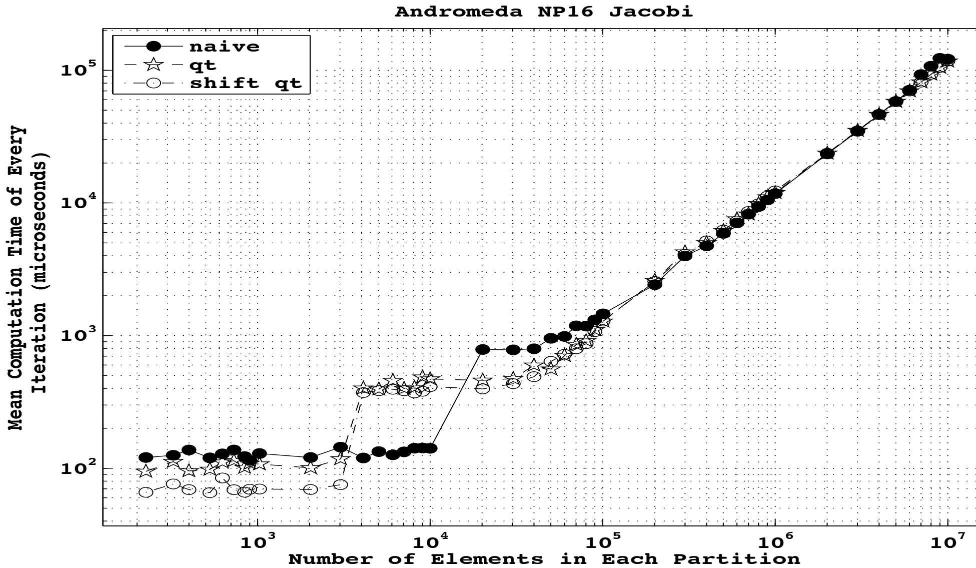


**Figure A.1:** Mean completion time of a single time step of the naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil executed on Andromeda.

The first set of tests is performed on Adromeda. From Figure A.1 we can see how the performance of the whole computation is increased by $\mathcal{Q}$ trasformations or $\mathcal{Q}$ trasformations and $\mathcal{SH}$ method . This effect is noticeable for partition's sizes up to $10^5$ elements. As expected, performance gain is noticeable for small sizes of the partition where communications' cost dominates. Figure A.2 shows that the cost of communications is generally reduced while the cost of computation is unaffected (Figure A.3). Notice that for some

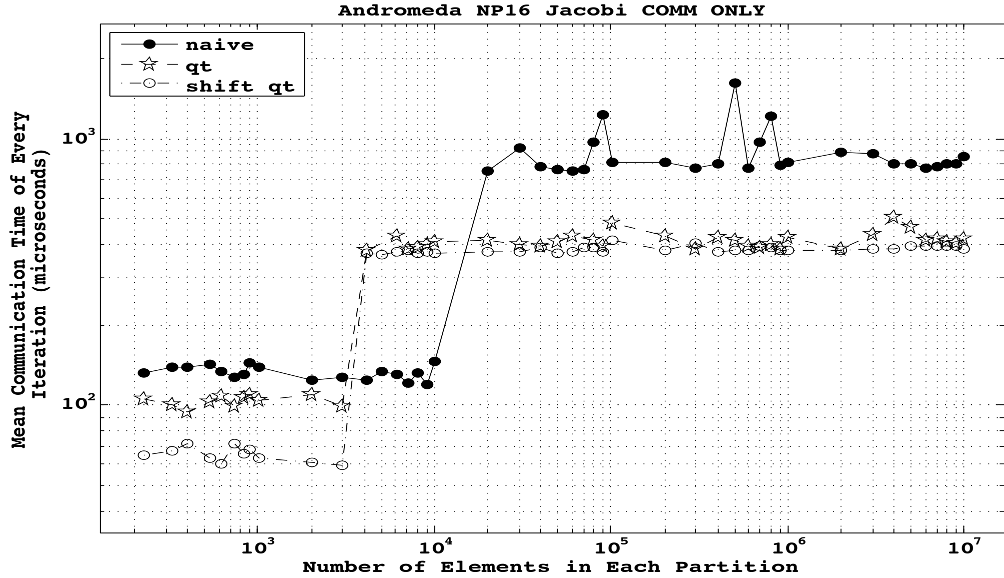


**Figure A.2:** Communication time of a single time step of the of naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil executed on Andromeda.

sizes of the partition there is a decrease in performance due to the fact that the cost of communications ( Figure A.2) increase abruptly with respect to the size of data transmitted. This is probably due to the library used for message passing. Since the family of $\mathcal{Q}$ transformations tends to group together communications (therefore performing bigger communications less frequently) they suffer of this deterioration of performance more than the naive version.

By performing the same test on another machine (Titanic) we obtain similar results (Figure A.4).

From Figure A.5 it is clear that communications are relevant only for a small size of the partition. Therefore, optimizations of the $\mathcal{Q}$ family can only be beneficial for small working domains in the two dimensional case.

Apart for some fluctuations on the experimental values we have a noticeable performance gain for partitions up to $10^5$ elements and almost negligible variation of performance for higher dimensionality.

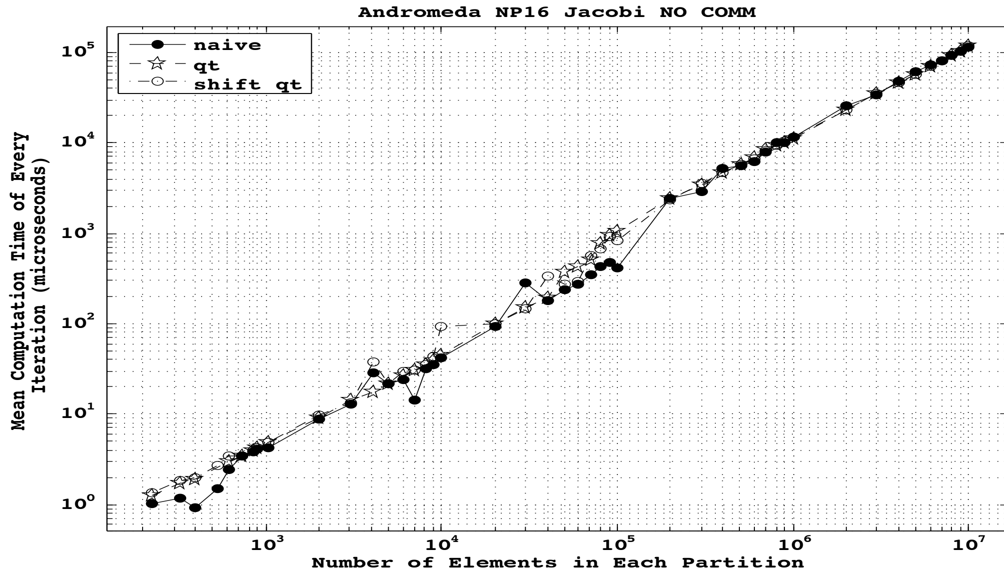


**Figure A.3:** Computation time of a single time step the of naive, $\mathcal{Q}$ , $\mathcal{Q}$ and $\mathcal{SH}$ versions of the two dimensional Jacobi stencil (without MPI communications) executed on Andromeda.

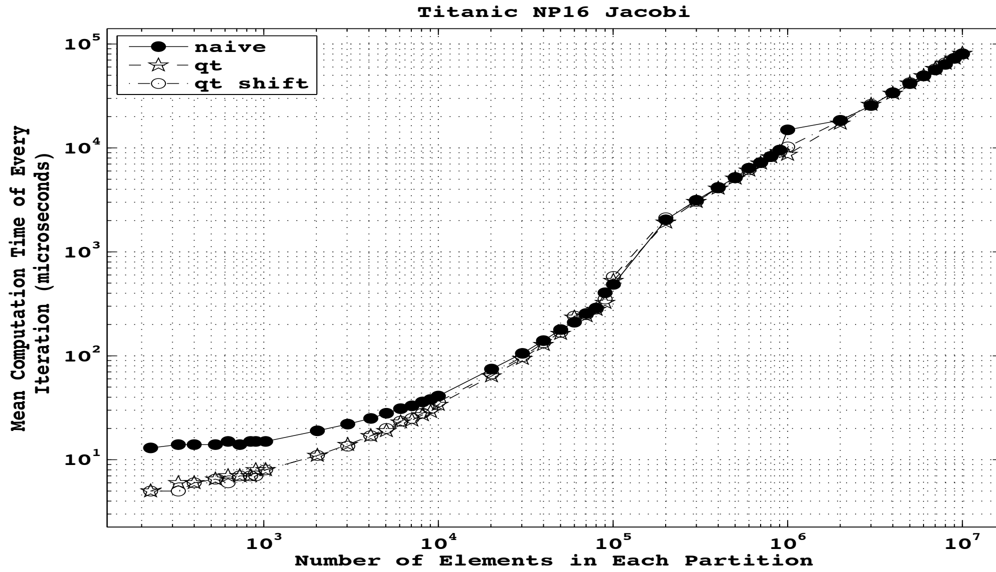


**Figure A.4:** Mean completion time of naive of the naive $\mathcal{Q}$ and $\mathcal{Q} + \mathcal{SH}$ version of the Jacobi stencil executed on Titanic.

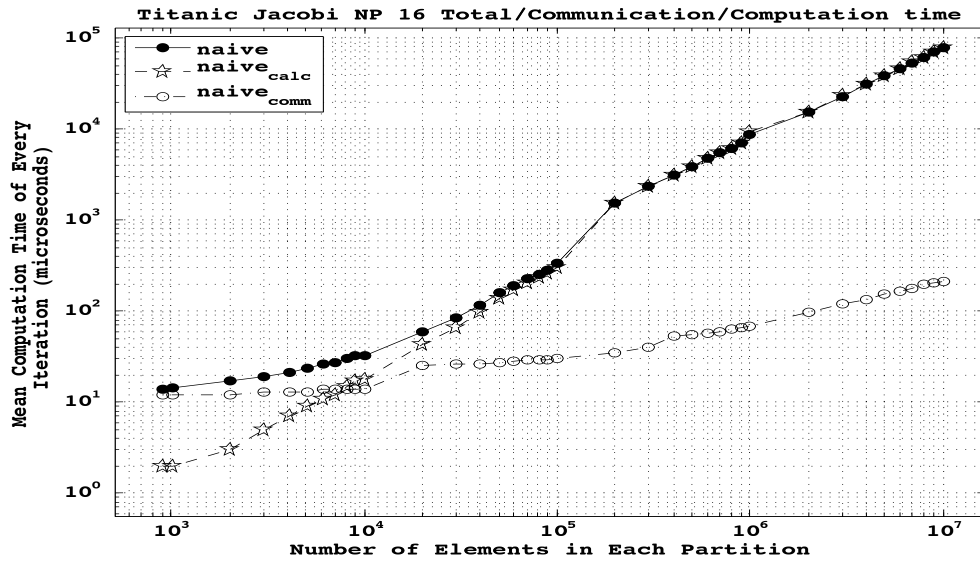


**Figure A.5:** Total, computation and communication time of the two dimensional jacobi stencil (naive version) executed on Titanic.

Figure A.6 shows results for the 3d jacobi stencil. Here the ratio of communication and computation is different; communications have a greater impact in performance and so do our optimization techniques. We can see that there is an increment in performance for partitions ranging from $10^4$ to $10^6$ elements.

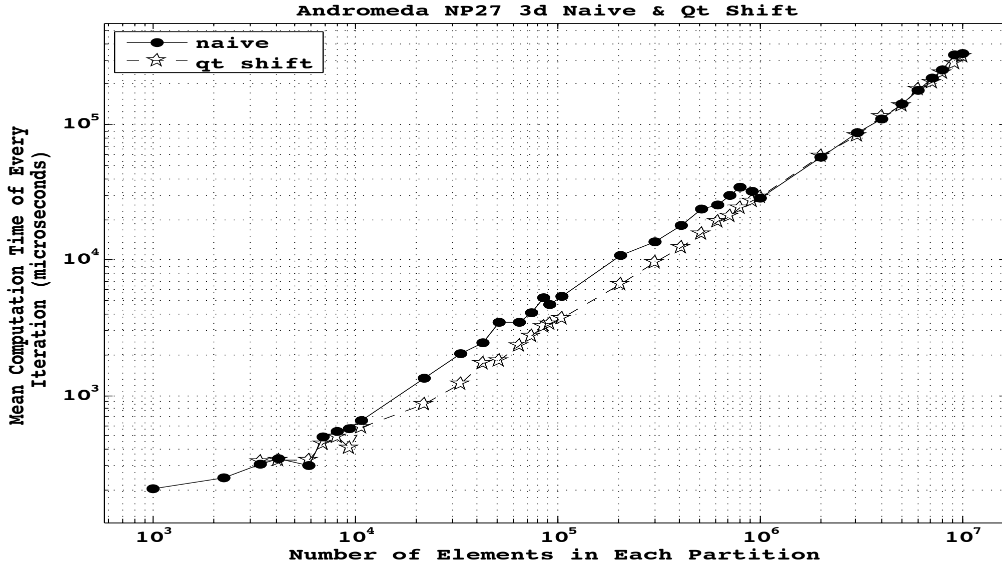


**Figure A.6:** Mean completion time of naive and $\mathcal{Q} + \mathcal{SH}$ version of the 3d Jacobi stencil executed on Andromeda.

Figure A.7 shows the effect of the $\mathcal{SF}$ trasformation on the Jacobi Stencil. These results perfectly reflects our theoretical analysis of Section 5.2. For partitions which can fit entirely in the L1 cache ( L1 is 64Kbyte therefore it can store less than $10^5$ double precision values) $\mathcal{SF}$ trasformation is counter productive since it increase the number of floating point operations. For partitions of bigger size there is a noticeable speed-up with respect to the naive version due primarily to the increased temporal locality (Figure A.8).

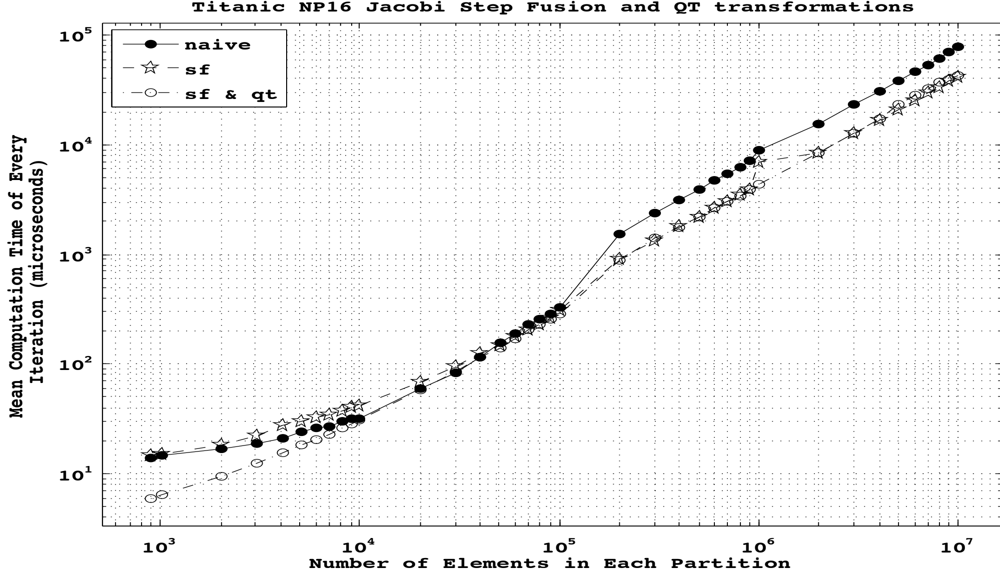


**Figure A.7:** Mean completion time of naive,$\mathcal{SF}^2$ and $\mathcal{SF}^2$ + $\mathcal{SH}$ version of the Jacobi stencil executed on Titanic. Values are normalized w.r.t the number of time steps.

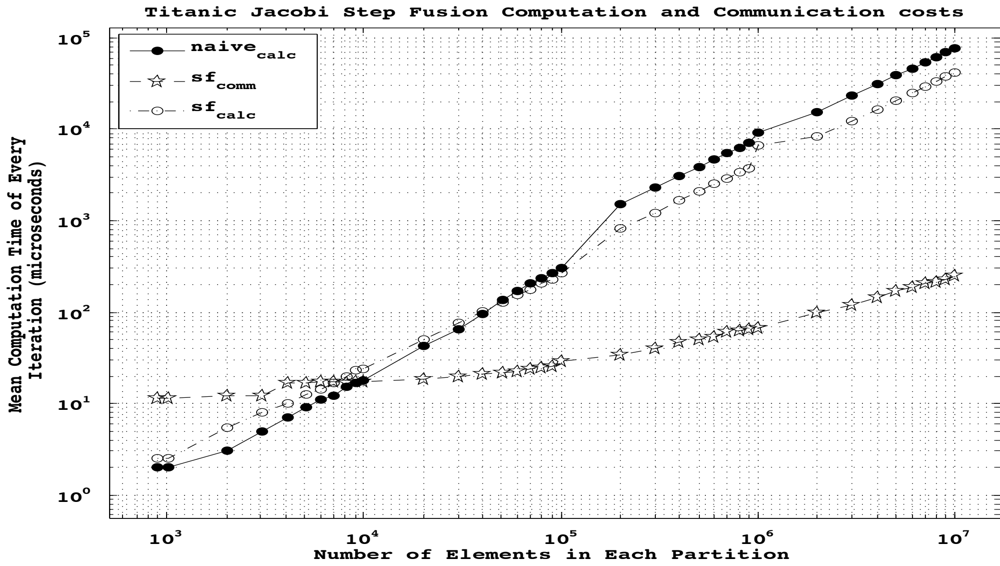


**Figure A.8:** Comparison of Computation cost for both naive and $\mathcal{SF}^2$ version. Communication cost of the naive version is shown for reference. Notice that for sufficiently big partitions the computation cost dominates.

Now we conclude our set of experimental results with the analysis of the loop tiling and loop skewing techniques presented in Section 5.5.1 and Section 5.5.2. We selected the two dimensional laplace stencil and a fixed cache block size of 32x32 for loop tiling and 32x32x32 for loop skewing. Results are presented in Figure A.9. While the performance gain is evident for the time skewing method, tiling only in the spatial dimensions does not produce significant improvements and might be counter productive (Figure A.10). In particular, apart from fluctuations in the experimental results, loop tiling seems more ineffective where it should be beneficial, i.e., when the partition size is big and can not fit in the lower levels of the memory hierarchy.

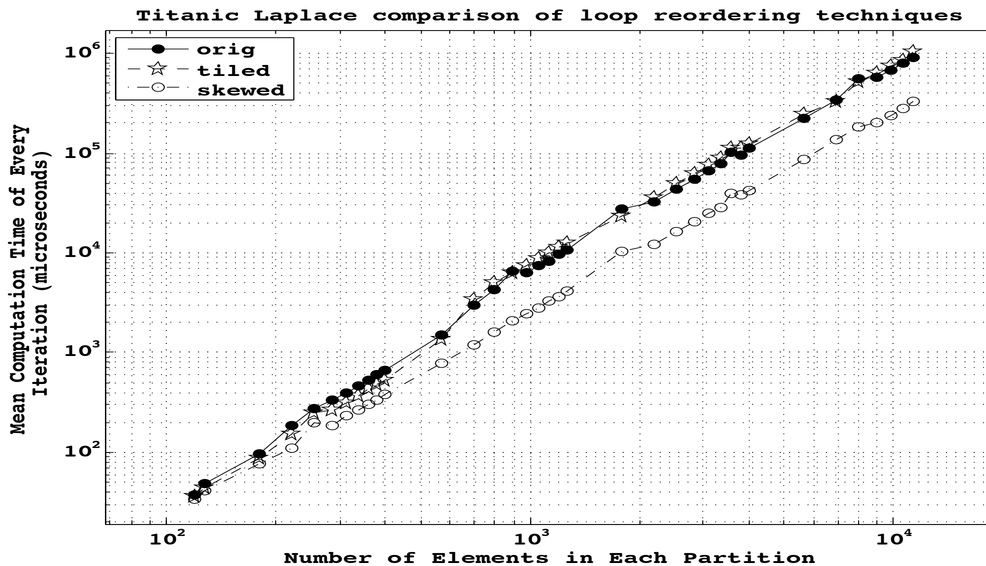


**Figure A.9:** Comparison of naive, loop tiling and time skewing version of the two dimensional laplace stencil.

This strange behaviour of loop tiling is well known in literature[25, 14]. This is caused by both hardware and software pre-fetching mechanisms which produce additional, unnecessary, memory traffic. In fact, hardware mechanisms assume that elements are accessed in a unit stride fashion which is obviously not true if loop tiling is used. On the other hand, eliminating pre-fetching altogether cause a dramatic degradation of performance. In [13] this problem is analyzed in detail. Since compilers (at the state of the art) cannot automatically determine tile's boundaries and insert pre-fetching instruction accordingly, the author concludes that they should be inserted by the programmer. Moreover, hardware pre-fetching mechanisms should be disabled.

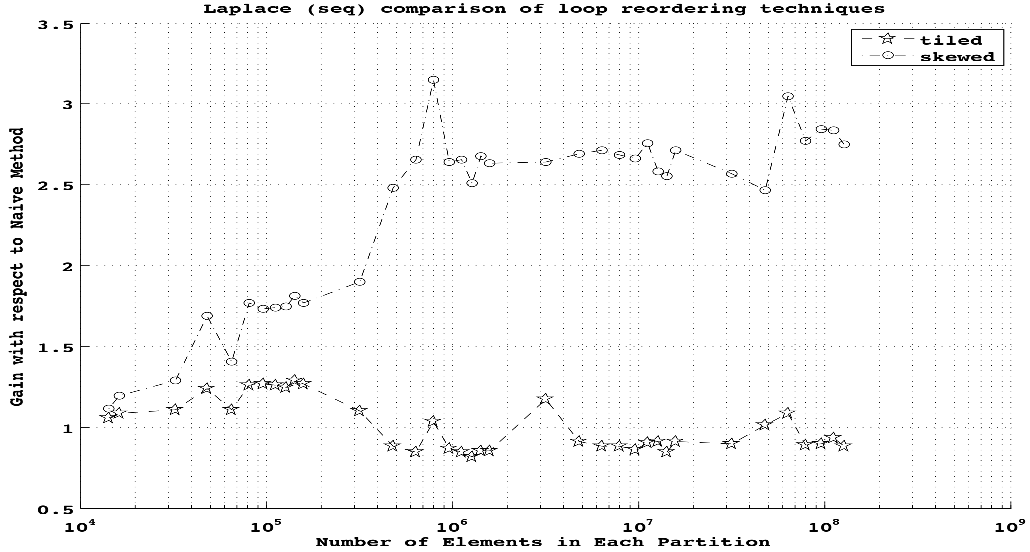


**Figure A.10:** Speedup ($T_{naive}/T_{opt}$) of loop tiled and time skewed version with respect to the naive version of the two dimensional laplace stencil.

Another possible explanation of results in Figure A.9 is that tiling increase code complexity. In particular tiling increase the number of for loops and introduces additional conditional instructions in order to consider partition sizes which are not exact multiples of tile sizes. Therefore, tiled concurrent code is less "compiler friendly" since it is more difficult to apply standard optimizations such as loop unrolling[37]. Even if optimal object code is produced, the number of conditional jumps in object code is inevitably higher and this might affect the computation performance.

# Bibliography

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Tatsuya Akutsu. Dynamic programming algorithms for rna secondary structure prediction with pseudoknots. *Discrete Appl. Math.*, 104(1-3):45–62, August 2000.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[6] Werner Benger, Ian Foster, Jason Novotny, Edward Seidel, John Shalf, Warren Smith, and Paul Walker. Numerical relativity in a distributed environment. In *9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[7] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[8] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality op-

timization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[10] D.R. Chesney and B.H.C. Cheng. Generalising the unimodular approach [program code transformation]. In *Parallel and Distributed Systems, 1994. International Conference on*, pages 398 –404, dec 1994.

[11] Rezaul Alan Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 7(3):495–510, July 2010.

[12] Matthias Christen, Olaf Schenk, Peter Messmer, Esra Neufeld, and Helmar Burkhart. Accelerating stencil-based computations by increased temporal locality on modern multi and many-core architectures. Technical report, University of Basel, 2008.

[13] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, Berkeley EECS, 2009.

[14] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, February 2009.

[15] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[16] James W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[17] C Ding and Y He. A ghost cell expansion method for reducing communications in solving pde problems. *Supercomputing*, Jan 2001.

[18] José A. Font, Mark Miller, Wai-Mo Suen, and Malcolm Tobias. Three-dimensional numerical general relativistic hydrodynamics: Formulations, methods, and code tests. *Phys. Rev. D*, 61:044011, Jan 2000.

[19] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[21] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[22] John C. Hull. *Options, Futures, and Other Derivatives with Derivagem CD (7th Edition)*, pages 427 – 438. Prentice Hall, 7 edition, May 2008.

[23] Shoaib Kamil, Cy Chan, Samuel Williams, Leonid Oliker, John Shalf, Mark Howison, and E. Wes Bethel. A generalized framework for autotuning stencil computations. In *In Proceedings of the Cray User Group Conference, 2009*, 2009.

[24] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 51–60, New York, NY, USA, 2006. ACM.

[25] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, pages 36–43, New York, NY, USA, 2005. ACM.

[26] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[27] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 99–106, New York, NY, USA, 2008. ACM.

[28] Randall LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.

[29] Marco Vanneschi Massimiliano Meneghin. Minimizing communications with $\mathcal{Q} - transformations$ in uniform and affine stencils. Technical report, University of Pisa, 2009.

[30] Renwei Mei, Wei Shyy, Dazhi Yu, and Li-Shi Luo. Lattice boltzmann method for 3-d flows with curved boundary. *J. Comput. Phys.*, 161(2):680–699, July 2000.

[31] Massimiliano Meneghin. *A New Optimization Theory for Structured Stencil-based Applications: An advance in understanding of stencil parallelization techniques*. PhD thesis, 2010.

[32] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[33] S Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[34] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[35] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–20, New York, NY, USA, 1997. ACM.

[36] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Cambridge University Press, 2003.

[37] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 49–59, New York, NY, USA, 2010. ACM.

[38] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[39] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.

[40] Marco Vanneschi. *Architetture degli Elaboratori.* Pisa University Press, 2009.

[41] Marco Vanneschi. High performance computing systems and enabling platforms. Course Notes, Master Degree Program in Computer Science and Networking, University of Pisa, 2011.

[42] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[43] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[44] Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *J. Parallel Distrib. Comput.*, 69(9):762–777, September 2009.

[45] David Wonnacott. Time skewing for parallel computers. In *In Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, pages 477–480. Springer-Verlag, 1999.