Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA

LEVERAGING PERFORMANCE OF 3D FINITE DIFFERENCE SCHEMES IN LARGE SCIENTIFIC COMPUTING SIMULATIONS

by

Raúl de la Cruz Martínez

Advisors:
José María Cela
Mauricio Araya Polo

DISSERTATION

Submitted in partial fulfillment of the requirements for the PhD Degree issued by the Department of Computer Architecture

Universitat Politècnica de Catalunya
Barcelona, Spain

October 2015

Abstract

Gone are the days when engineers and scientists conducted most of their experiments empirically. During these decades, actual tests were carried out in order to assess the robustness and reliability of forthcoming product designs and prove theoretical models. With the advent of the computational era, scientific computing has definetely become a feasible solution compared with empiricial methods, in terms of effort, cost and reliability. The deployment of powerful supercomputers, with thousands of computing nodes, have additionally promoted the extent and use of scientific computing. Large and massively parallel computational resources have reduced the simulation execution times and have improved their numerical results due to the refinement of the sampled domain.

Several numerical methods coexist for solving the Partial Differential Equations (PDEs) governing the physical phenomena to simulate. Methods such as the Finite Element (FE) and the Finite Volume (FV) schemes are specially well suited for dealing with problems where unstructured meshes are frequent owing to the complex domain to simulate. Unfortunately, this flexibility and versatility are not bestowed for free. These schemes entail higher memory latencies due to the handling of sparse matrices which involve irregular data accesses, therefore increasing the execution time. Conversely, the Finite Difference (FD) scheme has shown to be an efficient solution for specific problems where the structured meshes suit the problem domain requirements. Many scientific areas use this scheme for solving their PDEs due to its higher performance compared to the former schemes.

This thesis focuses on improving FD schemes to leverage the performance of large scientific computing simulations. Different techniques are proposed such as the Semi-stencil, a novel algorithm that increases the FLOP/Byte ratio for medium- and high-order stencils operators by reducing the accesses and endorsing data reuse. The algorithm is orthogonal and can be combined with techniques such as spatial- or time-blocking, adding further improvement to the final results.

New trends on Symmetric Multi-Processing (SMP) systems —where tens of cores are replicated on the same die— pose new challenges due to the exacerbation of the memory wall problem. The computational capability increases exponentially whereas the system bandwidth only grows linearly. In order to alleviate this issue, our research is focused on different strategies to reduce pressure on the cache hierarchy, particularly when different threads are sharing resources due to Simultaneous Multi-Threading (SMT) capabilities. Architectures with high level of parallelism also require efficient work-load balance to map computational blocks to the spawned threads. Several domain decomposition schedulers for work-load balance are introduced ensuring quasi-optimal results without jeopardizing the overall performance. We combine these schedulers with spatial-blocking and auto-tuning techniques conducted at run-time, exploring the parametric space and reducing misses in last level cache.

As alternative to brute-force methods used in auto-tuning, where a huge parametric space must be traversed to find a suboptimal candidate, performance models are a feasible solution. Performance models can predict the performance on different architectures, selecting suboptimal parameters almost instantly. In this thesis, we devise a flexible and extensible performance model for stencils. The proposed model is capable of supporting multi- and many-core architectures including complex features such as hardware prefetchers, SMT context and algorithmic optimizations (spatial-blocking and Semi-stencil). Our model can be used not only to forecast the execution time, but also to make decisions about the best algorithmic parameters. Moreover, it can be included in run-time optimizers to decide the best SMT configuration based on the execution environment.

Some industries rely heavily on FD-based techniques for their codes, which strongly motivates the ongoing research of leveraging the performance. Nevertheless, many cumbersome aspects arising in industry are still scarcely considered in academia research. In this regard, we have collaborated in the implementation of a FD framework which covers the most important features that an HPC industrial application must include. Some of the node-level optimization techniques devised in this thesis have been included into the framework in order to contribute in the overall application performance. We show results for a couple of strategic applications in industry: an atmospheric transport model that simulates the dispersal of volcanic ash and a seismic imaging model used in Oil & Gas industry to identify hydrocarbon-rich reservoirs.

To my little Inés,
my resilient wife Mónica
and my dear family

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Preface

1.1 Motivation of this Thesis

Nowadays, most of the scientific codes are run in supercomputers. To achieve full resource utilization, these codes must be able to scale from a single node up to a cluster. This scalable migration involves at least four main tasks to succeed: kernel computation, load balancing, input/output (I/O) and inter/intra-node communications. Therefore, the optimization efforts must be driven to these tasks in order to leverage the scientific computing performance.

The computational kernels in scientific applications usually consist of nested loops. In Von Neumann architectures most of the loop optimization techniques are based on improving the floating point to memory access ratio (FLOPS/Byte). This ratio measures the bytes required from memory with respect to the floating point operations performed. The higher this metric is, the better performance is expected. However, there are no general rules, and the optimization techniques may vary depending on the underlying platform due to the memory system intricacy, e.g. complex multi-level cache hierarchies or scratchpad memories.

The parallelization of scientific codes is conducted at two different levels: inter-node and intra-node, which are usually performed with message passing libraries (e.g. MPI) and shared memory multiprocessing APIs (e.g. OpenMP) respectively. At intra-node level, two parallelism approaches can be distinguished: Thread Level Parallelism (TLP) —that exploits the multiple cores within a node—, and Instruction Level Parallelism (ILP) —that takes advantage of the multiple execution units within a core. In TLP, load-balancing is critical in order to maximize the overall throughput and reduce the execution time. However, thread imbalance may be exacerbated as the trend to many-core integration in the same die is strengthened within Symmetric Multi-Processing architectures (SMP). This issue will require the deployment of efficient load-balancing and auto-tuning techniques to leverage the intra-node performance.

Another concern is the inter-node communication, which is essential in parallel computing to keep the synchronization and coherence of the global results during the time-steps of the simulation. Minimizing the impact of communication between domains is crucial to prevent node stagnation. As a solution, the boundary regions can be reduced and the communication of contiguous allocated data promoted. In this way, the message sizes are minimized and the temporal copies of scattered data across the parallel stack avoided. In this regard, task overlapping can also help allowing the simultaneous execution of kernel computation, I/O and communications, thus hiding the shortest latencies.

The main goal of this thesis is to improve the performance of FD-based codes at every above-mentioned level. Within the scientific computing, the FD scheme is typically employed to solve problems dealing with structured meshes that arise in science and industry. However, some application-based implementations of the FD scheme can only reach up to 30-40% of the peak performance [5]. This inefficient usage of resources has motivated this research, exploring new strategies to improve the critical sections of these codes, either serial or parallel. Since the *stencil* can represent up to 80% of the execution time in some FD-based simulations [9, 42], this kernel is our first target for optimization. To achieve it, we focus on enhancing the memory access pattern and increasing the data re-utilization at core-level, devising also new load-balancing strategies as node-level optimizations.

1.2 Thesis Contributions

This thesis contributes to the performance improvement of FD-based codes focusing on the following main areas:

- A novel algorithm called *Semi-stencil* has been developed for computing stencil operators. This new algorithm establishes an innovative way of computing the spatial operator for medium- and high-order stencils. Its use permits data reuse and reduces the required dataset per point.

- We have unveiled new ways of improving the stencil computations on architectures with Symmetric Multi-Processing (SMP) and Simultaneous Multi-Threading (SMT) capabilities. These strategies are based on a group of schedulers for arranging and traversing the computational domain promoting cache effectiveness and scalability on many-core architectures.

- As improvement to the multi-core and SMT environment, we have developed a straightforward auto-tuner that selects pseudo-optimal tiling parameters (TJ) for computational domains. The auto-tuner, when is combined with domain schedulers, improves even further the expected performance by reducing misses in last level caches.

- A sophisticated performance model for stencil computations has been elaborated to mimic the behavior on a broad variety of computer architectures. This model is a big step forward compared with any previous stencil performance model that can be found in the literature. It includes modern capabilities such as hardware prefetching, multi-level cache hierarchy and many-core support. It has been designed to be modular, incorporating spatial-blocking and Semi-stencil optimizations.

- Our stencil model not only predicts the best parameters for a given optimization algorithm, but also conveys hints of outperforming SMT configurations that can be conducted for a specific architecture.
- Finally, we have collaborated in the development of WARIS, a FD-based framework for High Performance Computing, that deploys efficient performance when is tailored to the industry. This framework is intended to be modular and considers all the requirements for production, including overlapping of computation and communication as well as asynchronous I/O. We demonstrate the effectiveness of our core- and nodel-level optimizations on two real applications used in the industry.

Other minor contributions of this thesis are the following:

- We have extended the StencilProbe micro-benchmark [43] to obtain and assess the experimental results. In order to conduct the experiments, we have incorporated the proposed core- and node-level optimizations using OpenMP to exploit the Thread Level Parallelism.
- The STREAM2 benchmark [55] has been enhanced to fully characterize the memory hierarchy of modern architectures. This benchmark is now able to gather non-streaming bandwidth in multi-threaded environments considering aligned, SIMD and non-temporal accesses.
- We have developed from scratch a new micro-benchmark called *Prefetchers* that aims to characterize the prefetching engines of an architecture. By means of this tool, we can gather insightful metrics such as the prefetching efficiency, the triggering of the prefetcher and the look-ahead parameters.

1.2.1 Thesis Limitations

The current research is targeted at stencil computations on structured cartesian grids (constant coefficients). However, although not explicitly covered in this thesis, our research may be easily extended to suit other structured meshes such as rectilinear and curvilinear grids (variable coefficients). On the other hand, non-structured grids are out of scope due to the incompatibility of their irregular connectivity with our algorithms.

In addition, despite the recent emergence of massively-parallel accelerators such as the Graphics Processing Units (GPUs), this thesis is specially devoted to scratchpad and cache-based architectures with a high emphasis on multi- and many-core processors. Even though, the latest GPUs with some levels of cache may take partial advantage of this work.

1.3 Thesis Outline

This thesis is organized as follows:

In Chapter 2, a brief background of the most representative numerical methods for solving scientific problems is given. Then, we focus on FD schemes, on which this thesis is specially devoted. The FD scheme is introduced, explaining the explicit and implicit approaches for obtaining the numerical solutions.

Chapter 3 presents the different architectures used for the elaboration of this thesis. We also detail the whole set of tools, characterization benchmarks, libraries and parallel paradigms employed.

In Chapter 4, we unveil the stencil-specific optimizations developed at core-level such as the novel Semi-stencil algorithm and its performance when combined with other common strategies.

Chapter 5 proposes specific domain decomposition schedulers that aim to improve the stencil performance when is scaled at node-level. We also propose the combination of these schedulers with auto-tuning methods in order to allow last level cache optimizations.

Chapter 6 introduces the performance models for stencil computations and details the multi-level cache model developed in this research and its insightful results.

In Chapter 7, we present the implementation of some optimization techniques from Chapters 4 and 5 for two strategic applications for the society: a seismic imaging model used by Oil & Gas industry and a volcanic ash dispersal model used by civil aviation authorities.

Finally, Chapter 8 concludes this thesis with a summary of the contributions of each topic, and proposing ideas for possible future work.

1.4 List of Publications

The work presented in this thesis has been presented and published in journals and proceedings of international conferences. The papers and talks categorized by topic are the following:

Semi-stencil:

- Raúl de la Cruz and Mauricio Araya-Polo. Algorithm 942: Semi-stencil. *ACM Transactions on Mathematical Software (TOMS)*, 40(3):23:1–23:39, April 2014

- Raúl de la Cruz, Mauricio Araya-Polo, and José María Cela. Introducing the *Semi-stencil* algorithm. In *Parallel Processing and Applied Mathematics, 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*, pages 496–506, 2009

- Mauricio Araya-Polo and Raúl de la Cruz. Semi-stencil algorithm: Improving data locality and reuse in stencil computation. 14th SIAM Conference on Parallel Processing for Scientific Computing, Seattle, Washington, February 2010. Part of CP4 PDEs (Talk)

Performance Modeling:

- Raúl de la Cruz and Mauricio Araya-Polo. Modeling stencil computations on modern HPC architectures. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 149–171. Springer International Publishing, 2015

- Raúl de la Cruz and Mauricio Araya-Polo. Towards a multi-level cache performance model for 3D stencil computation. In *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*, pages 2146–2155, 2011

- Raúl de la Cruz and Mauricio Araya-Polo. Using modeling to develop stencil codes. 2015 Rice Oil & Gas HPC Workshop, Rice University, Houston, March 2015. Coarse-grained Seismic Algorithms (Talk)

- Raúl de la Cruz and Mauricio Araya-Polo. Modeling stencil code optimizations. 16th SIAM Conference on Parallel Processing for Scientific Computing, Portland, Oregon, February 2014. CP8: Performance Optimization for Stencils and Meshes (Talk)

- Mauricio Araya-Polo and Raúl de la Cruz. Performance model for 3D stencil computation. 2012 Rice Oil & Gas HPC Workshop, Rice University, Houston, March 2012. Parallel session A: Benchmarking, Optimization & Performance (Talk)

Case Studies:

- Raúl de la Cruz, Arnau Folch, Pau Farré, Javier Cabezas, Nacho Navarro, and José María Cela. Optimization of atmospheric transport models on HPC platforms. *Computational Geosciences*, 2015. (Submitted)

- Raúl de la Cruz, Mauricio Hanzich, Arnau Folch, Guillaume Houzeaux, and José María Cela. Unveiling WARIS code, a parallel and multi-purpose FDM framework. In *Numerical Mathematics and Advanced Applications - ENUMATH 2013 - Proceedings of ENUMATH 2013, the 10th European Conference on Numerical Mathematics and Advanced Applications, Lausanne, Switzerland, August 2013*, pages 591–599, 2013

- Mauricio Araya-Polo, Félix Rubio, Mauricio Hanzich, Raúl de la Cruz, José María Cela, and Daniele P. Scarpazza. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Scientific Programming, Special Issue on the Cell Processor*, 17, 2009

- Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. High-performance seismic acoustic imaging by reverse-time migration on the Cell/B.E. architecture. *ISCA2008 - WCSA2008*, 2008

- Raúl de la Cruz, Mauricio Hanzich, and José María Cela. Stencil computations: from academia to industry. 16th SIAM Conference on Parallel Processing for Scientific Computing, Portland, Oregon, February 2014. Part of MS66 Optimizing Stencil-based Algorithms - Part II of II (Talk)

Other Publications:

- Muhammad Shafiq, M. Pericas, Raúl de la Cruz, Mauricio Araya-Polo, Nacho Navarro, and E. Ayguade. Exploiting memory customization in FPGA for 3D stencil computations. In *IEEE International Conference on Field-Programmable Technology*, 2009
- Francisco Ortigosa, Mauricio Araya-Polo, Raúl de la Cruz, and José M. Cela. Seismic imaging and the road to peta-scale capacity. *17th SPE - 70th EAGE Conference*, 2008
- Francisco Ortigosa, Mauricio Araya-Polo, Félix Rubio, Mauricio Hanzich, Raúl de la Cruz, and José Maria Cela. Evaluation of 3D RTM on HPC platforms. *SEG 2008, Las Vegas, USA*, 2008
- Francisco Ortigosa, Hongbo Zhou, Santiago Fernandez, Mauricio Hanzich, Mauricio Araya-Polo, Felix Rubio, Raúl de la Cruz, and José M. Cela. Benchmarking 3D RTM on HPC platforms. Instituto Argentino del Petroleo y del Gas, November 2008

1.5 Acknowledgements

This work could not have been done without the support, in knowledge and inspiration, of many people.

First, I want to thank the support of my family, but specially the one given by my wife. Her resilience and patience to the large process of getting my PhD has endorsed me to success in this endeavour.

I would also want to thank my thesis advisor Mauricio Araya-Polo, for his support and willingness to motivate me in the worst moments to finish my doctoral degree.

I am grateful to Arnau Folch and José María Cela for giving me the opportunity of developing the WARIS-Transport module and the liberty of exposing and implementing some new ideas. It has not been a smooth journey, but it has inspired me in insightful research that has allowed me to fulfill the requirements of this thesis.

I am also indebted to the Barcelona Supercomputing Center and my colleagues at the Computer Applications in Science & Engineering (CASE) department for the nice working atmosphere that motivated to give the best of me. All the years working at the BSC have been enriching and fruitful in intellectual terms.

I would also like to express my gratitude to the Partnership for Advanced Computing in Europe (PRACE) for the grant of computing time on their supercomputing facilities. The

PRACE project receives funding from the EU's Seventh Framework Programme (FP7/2007-2015) under grant agreements RI-211528, 261557, 283493 and 312763. This thesis has been partially supported by the Spanish Ministry of Science and Innovation (CICYT) under grants TIN2007-60625 and 2009501052. Finally, I would like to acknowledge Intel for providing early access to a MIC based system.

Chapter 2

Introduction

2.1 Numerical Methods

Before the transistor revolution, many engineering fields needed costly and time consuming experiments to improve the product design [19, 63]. However, in the last decades, the cost reduction of microprocessors and memory has allowed the advent of a new supercomputers generation [77]. The availability of supercomputers has enabled scientists to conduct large and complex simulations that usually consume days of computational time. Since then, scientific simulations have become a cutting-edge factor for industry. They have permitted to shorten the development cycle of engineering products by assisting in design and optimization stages, thus eliminating lengthy and costly prototype manufacturing. As an example, scientific computing may help to reduce friction coefficients of high speed trains by slightly changing the product geometry.

Today, large scientific computing simulations are widely used to solve numerical problems arising in Aerospace [44], Meteorology [76], Astrophysics [12], Geophysics [58, 65], Quantum Chemistry [4, 15] and Oceanography [38, 45]. Most of these scientific simulations have something in common, the physical and mathematical models are approximated using Partial Differential Equations (PDEs), which can be solved using various numerical analysis schemes.

In general terms, we could enumerate the following main reasons to perform scientific simulations:

- Analysis: in some engineering fields (e.g. Aerospace) it is far too expensive and difficult to build specific experimental models. In addition, simulations can provide data that empirically is impossible or very hard to obtain.

- Prediction: it provide answers to different design implementations in a short time of period, improving the quality of the final product.

- Efficiency: as a consequence of the shortening of the design and development stages, the final product can reach faster to the market.

The most representative fields within scientific computing are Computational Mechanics (CM) and Computational Physics (CP). They are devoted to study phenomena governed by the principles of mechanics and physics by means of computational methods. Both are interdisciplinary fields that are based on three disciplines: physics, mathematics and computer science. The development of computational simulations codes in these fields involves the following steps, which are also represented in Figure 2.1:

1. Mathematical modeling: a physical phenomenon or engineering system is represented by PDEs in the continuous domain.

2. Discretization of the mathematical model: PDEs are turned into forms which are suitable to be solved by computers. In this step, the numerical scheme (e.g. Finite Difference or Finite Elements) discretizes the original continuous PDEs. Commonly, the PDEs involved are translated into a system of algebraic equations.

3. Computational implementation: once the discretized equations are built, the scientific code is written to represent the equations as computer programs. The discretized equations are solved using direct or iterative methods.

4. Validation: the software application, the numerical method and the mathematical model are verified either empirically or using simplified models with exact analytical solutions.

$$\begin{tikzcd} \text{Continuous PDE for } \phi(x,t) \arrow[r, "\text{Numerical method}"] & \text{Discrete Difference Equation } \phi(x_i,t_n) \arrow[r, "\text{Solution method}"] & \phi_i^n \text{ approximation to } \phi(x,t) \end{tikzcd}$$

Figure 2.1: Translation from the continuous to the discrete problem. $\phi(x,t)$: continuous solution (exact solution). $\phi(x_i,t_n)$: continuous solution evaluated at the mesh points. ϕ_i^n: approximated solution by solving the numerical scheme.

High Performance Computational Simulations (HPCS) is a sub-discipline of scientific computing, where the implementation of the scientific simulation runs on large supercomputers. Due to the cluster nature of the supercomputers, HPCS is intrinsically related to highly parallel demanding computations. This large computations involve two main aspects. First the kernel computation, which solves numerically the PDEs and consumes large amount of resources, both computational and memory. Second the parallelization step, that permits to scale the performance up to thousands of nodes. This parallelization may be performed in two levels: intra-node, which takes advantage of the multi-core nature of modern architectures, and inter-node that enables the cluster-level parallelism.

The most popular numerical schemes used nowadays are the Finite Difference (FD), the Finite Element (FE) and the Finite Volume (FV) methods. Table 2.1 shows a summary of the advantages and disadvantages of these methods. FD method is commonly used in thermodynamics, electromagnetism and geophysics, whereas FE and FV are widely used in solid and fluid mechanics problems. There are also other methods to numerically solve PDEs (e.g. Discrete Element, Spectral or Particle-In-Cell), but they are out of scope of this introduction. In

Num. Scheme	Finite Difference	Finite Element	Finite Volume
Appeared in	20s	70s	80s
Robustness	Medium	High	High
Meshes	Structured	Non-structured	Non-structured
Coverage	Specific problems	All kind of problems	All kind of problems
Peak Perf.	Up to 40%	Between 8-12%	Between 8-12%

Table 2.1: Comparison of FD, FE and FV numerical schemes.

the following sections, we will review in detail the FD method, and the ways of numerically solve PDEs using this scheme (implicitly or explicitly).

2.2 Finite Difference Method

The Finite Difference method (FD) is the simplest and straightforward way of numerically solve PDEs [80]. The FD consists in approximating the solutions to the PDEs by replacing derivative expressions with approximately equivalent difference quotients. Therefore, the PDE is converted into a set of finite difference algebraic equations. As a result, the continuous domain of the problem is transformed into a structured mesh of discrete points. In this structured mesh, by using different forms of the finite different equations, a wide variety of physical phenomena can be conformed.

Consider the case of a function with one variable $u(x)$. Given a set of points $x_i; i = 1, ..., N$ in the domain of $u(x)$, as shown in Figure 2.2, the numerical solution is represented by a discrete set of function values $\{u_1, ..., u_N\}$ that approximate u at these points, i.e., $u_i \approx u(x_i); i = 1, ..., N$.

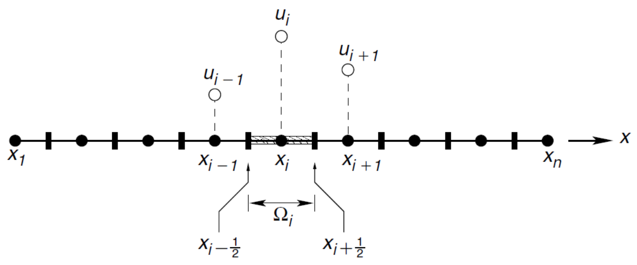


Figure 2.2: Discretization of $u(x)$ the domain.

Generally, and for simplicity, the points are equally spaced along the domain with a Δx constant distance ($\Delta x = x_{i+1} - x_i$), so we write $u_{i+1} \approx u(x_{i+1}) = u(x_i + \Delta x)$. This discretized domain is referred to as structured mesh or grid.

PDEs involve unknown functions of several independent variables (in time and space) and their partial derivatives. The derivative of a unknown $u(x)$ with respect to x can be approximated by linear combinations of function values at the grid points,

$$
\begin{aligned}
\frac{\partial u}{\partial x}(x_i) &= \lim_{\Delta x \to 0} \frac{u(x_i + \Delta x) - u(x_i)}{\Delta x} \approx \frac{u(x_{i+1}) - u(x_i)}{\Delta x} \quad \text{(FD)} \\
&= \lim_{\Delta x \to 0} \frac{u(x_i) - u(x_i - \Delta x)}{\Delta x} \approx \frac{u(x_i) - u(x_{i-1})}{\Delta x} \quad \text{(BD)} \\
&= \lim_{\Delta x \to 0} \frac{u(x_i + \Delta x) - u(x_i - \Delta x)}{2\Delta x} \approx \frac{u(x_{i+1}) - u(x_{i-1})}{2\Delta x} \quad \text{(CD)}\ .
\end{aligned}
\tag{2.1}
$$

All these expressions are equivalent and they converge to the derivative of $u(x)$ when $\Delta x \to 0$. However, when Δx is finite but small enough, the Equations 2.1 can be used to obtain approximations of the derivative of $u(x)$. In that case, we may use u_i, u_{i-1} and u_{i+1} to obtain the derivative.

Such approximations (Equations 2.1), depending on the points used for the derivative, are referred to as forward (FD: x_i, x_{i+1}), backward (BD: x_{i-1}, x_i) and centered (CD: x_{i-1}, x_i, x_{i+1}) finite difference approximations of $u(x_i)$. The precision of such approximations may differ from each other, where the accuracy depends on the number of neighbors used (order of accuracy) and the discretization of the mesh.

A way to compute a better accuracy of first and high order derivatives approximations is using Taylor series expansions around the point x_i,

$$
\begin{aligned}
u(x) &= \sum_{n=0}^{\infty} \frac{(x - x_i)^n}{n!} \frac{\partial^n u}{\partial x^n}(x_i) \\
u_{i+1} &= u_i + \Delta x \left(\frac{\partial u}{\partial x}\right)_i + \frac{(\Delta x)^2}{2} \left(\frac{\partial^2 u}{\partial x^2}\right)_i + \frac{(\Delta x)^3}{6} \left(\frac{\partial^3 u}{\partial x^3}\right)_i + R_n(x) \\
u_{i-1} &= u_i - \Delta x \left(\frac{\partial u}{\partial x}\right)_i + \frac{(\Delta x)^2}{2} \left(\frac{\partial^2 u}{\partial x^2}\right)_i - \frac{(\Delta x)^3}{6} \left(\frac{\partial^3 u}{\partial x^3}\right)_i + R_n(x)\ .
\end{aligned}
\tag{2.2}
$$

Taylor expansions permit to elaborate high-order accuracy derivatives using more neighbors of the grid. Depending on the truncation point of the Taylor serie, we may have a bigger or smaller residual term ($R_n(x)$). This truncation error is proportional to the mesh discretization and the number of expanded series ($\mathcal{O}(\Delta x^n)$/n-th order accuracy). Table 2.2 summarizes the coefficient terms of various order finite difference formulas up to fourth-order derivatives.

Two different approaches can be used in scientific computing in order to obtain the numerical solutions of time-dependent PDEs: explicit and implicit methods. These two methods are discussed in detail in Section 2.3, but generally, when FD methods are implemented in scientific simulations, explicit time integration schemes are mostly used as direct solvers. Explicit methods can solve time-dependent PDEs directly, involving both the current state of the system and some previous ones, advancing the solution through small intervals of time (Δt).

Forward Difference - $\mathcal{O}(\Delta x)$

Order	Disc. coeff	u_i	u_{i+1}	u_{i+2}	u_{i+3}	u_{i+4}
$\partial u/\partial x$	Δx	-1	1			
$\partial^2 u/\partial x^2$	Δx^2	1	-2	1		
$\partial^3 u/\partial x^3$	Δx^3	-1	3	-3	1	
$\partial^4 u/\partial x^4$	Δx^4	1	-4	6	-4	1

Backward Difference - $\mathcal{O}(\Delta x)$

Order	Disc. coeff	u_{i-4}	u_{i-3}	u_{i-2}	u_{i-1}	u_i
$\partial u/\partial x$	Δx				-1	1
$\partial^2 u/\partial x^2$	Δx^2			1	-2	1
$\partial^3 u/\partial x^3$	Δx^3		-1	3	-3	1
$\partial^4 u/\partial x^4$	Δx^4	1	-4	6	-4	1

Central Difference - $\mathcal{O}(\Delta x^2)$

Order	Disc. coeff	u_{i-2}	u_{i-1}	u_i	u_{i+1}	u_{i+2}
$\partial u/\partial x$	$2\Delta x$		1	0	1	
$\partial^2 u/\partial x^2$	Δx^2		1	-2	1	
$\partial^3 u/\partial x^3$	$2\Delta x^3$	1	2	0	2	1
$\partial^4 u/\partial x^4$	Δx^4	1	-4	6	-4	1

Table 2.2: Different order Finite Difference formulas for BD, FD and CD.

Algorithm 1 General control flow structure of a generic Finite Difference code. The tasks to be performed can be categorized in four different types: Kernel Computation (KC), Data Communication (DC), Input/Output (IO) and Load Balancing (LB).

```
1:  Domain decomposition of the structured mesh                          (LB)
2:  for $t = time_{start}$ to $time_{end}$ in $\Delta t$ steps do
3:      Read input for $t$                                               (IO)
4:      Pre-process input                                                (KC)
5:      Inject source                                                    (KC)
6:      Apply boundary conditions                                        (KC)
7:      for all discretized points in my domain do
8:          Stencil computation (update $u^t$)                           (KC)
9:      end for
10:     Exchange overlap points between neighbor domains                 (DC)
11:     Post-process output                                              (KC)
12:     Write output for $t$                                             (IO)
13: end for
```

A parallel implementation of the explicit FD solver involves, amongst many others steps, the so-called *stencil computation* (see Algorithm 1 line 8), which solves the spatial and temporal differential operators of the governing equations. The stencil computation consists in accumulating the contribution of the neighbors points (u^{t-1}) in the discretized domain

($i = 1, ..., N$) along the cartesian axis, updating in this way the unknown for the next time step (u_i^t). This operation is usually the main computational bottleneck of FD codes and it may require up to 80% of the total execution time for some simulations [8, 9, 67]. Nevertheless, the stencil is conducted through an outer loop advancing the simulation over time (t) and composed of other tasks that concern minor kernel computations (KC), data communication (DC) and Input/Output (IO). Likewise, prior to the loop time iteration, a load balancing process must be carried out to ensure appropriate decomposition of the global domain among the computational resources in charge of the parallel simulation.

At any time step, the simulation may require acquiring and pre-processing input data (lines 3 and 4) as new solver parameters and post-processing and writting the current system state as the output solution (lines 11 and 12). The simulation also involves applying the boundary conditions (line 6), which are numerical tricks to confine the problem in a finite domain. This is done by prescribing the boundary elements (u_1 and u_N) of the domain to specific values that replicate the environment outbounds. Last but not least important, the data communication process (line 10) is conducted after the computation of the stencil. In this step, each computational resource exchanges its boundary areas with their neighbors in order to proceed with the local update of the system state.

2.3 Implicit and Explicit Methods

In spatial approximations, PDEs can be solved through two different methods: implicit and explicit.

In order to illustrate how we can find numerical solutions using both methods, let us consider a heat equation problem (see Appendix A.1). The heat equation is a PDE that describes the evolution of the heat distribution in a medium over time. This kind of equations can be used to describe the heat conduction in a metal rod. In one-dimensional problems and in the continuum case, this equation is written as

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{2.3}$$

where α is the thermal diffusivity of the material being described, and u is the unknown temperature.

This first-order in time and second-order in space PDE can be discretized by using Finite Differences in a equally spaced grid (Δx). Then, we can formalize the heat problem as

$$u_i^t = u(t, i) = u(t\Delta t, i\Delta x)\ ;\ i = 1, \ldots, N\ ;\ t \geq 0. \tag{2.4}$$

At this point, we continue the discretization by replacing the continuous derivatives in space and time with discrete approximations of FD. In the interest of simplicity, we use a central difference (CD) in space with respect to x ($u_{i-1}^t - 2u_i^t + u_{i+1}^t$). However, we use a backward difference (BD) for the derivative with respect to t. Using this numerical scheme, the heat equation is discretized as

$$\frac{1}{\Delta t}(u^t - u^{t-1}) = \frac{\alpha}{\Delta x^2}(u^t_{i-1} - 2u^t_i + u^t_{i+1}) \tag{2.5}$$

obtaining an implicit scheme where the unknown (u^t) function value includes terms involving as well unknown derivative values. In other words, we obtain a system where linear combinations of unknown derivative values (u^t) are given in terms of known function values (u^{t-1}). Then, rearranging the terms, we can obtain the following algebraic equation for each discrete point to solve in the grid,

$$\underbrace{-ru^t_{i-1} + (1+2r)u^t_i - ru^t_{i+1}}_{A \text{ coefficients}} = \underbrace{u^{t-1}_i}_{RHS}, \quad r = \frac{\alpha \Delta t}{\Delta x^2}. \tag{2.6}$$

Figure 2.3.Left shows an example with 5 points where the Backward-Time-Central-Space (BTCS) scheme has been used.

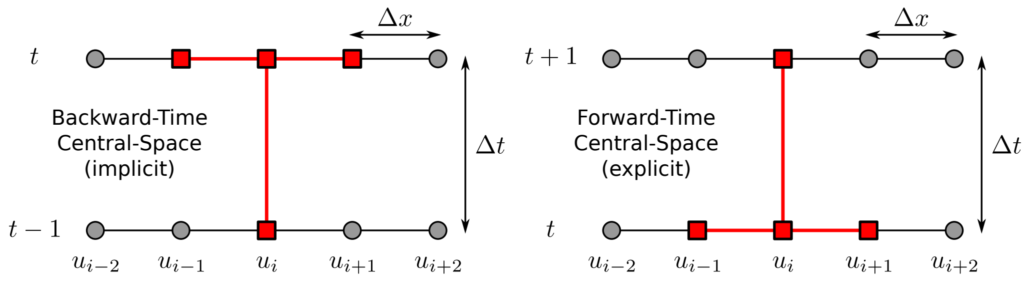


Figure 2.3: FD example of 1D problem with a 5 points grid domain. Left: Backward in Time and Central in Space scheme (implicit). Right: Forward in Time and Central in Space scheme (explicit). Red squares depict the data dependencies for each scheme.

To obtain a well-defined problem, some initial conditions must be applied on the boundary nodes of the domain (u_0 and u_4 of the 5 point example). These two end points depend on values that are outside of the computational domain (u_{-1} and u_5), and therefore we must enforce boundary conditions to confine the problem in the discrete domain. There are different boundary conditions that can be applied, one of them is the Dirichlet boundary condition that imposes the boundary to a fixed value,

$$u^t_0 = \beta, \quad u^t_4 = \gamma, \quad \forall t \tag{2.7}$$

Once the problem has been well-defined enforcing the boundary condition to the end nodes of the grid, we can proceed to build the linear system. For the 5 point grid example, the final system of equations would be

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^t \\ u_1^t \\ u_2^t \\ u_3^t \\ u_4^t \end{bmatrix} = \begin{bmatrix} \beta \\ u_1^{t-1} \\ u_2^{t-1} \\ u_3^{t-1} \\ \gamma \end{bmatrix} \tag{2.8}$$

where the boundary nodes (u_0^t and u_4^t) have already been imposed as Dirichlet boundary conditions (β and γ). Structured meshes produce banded matrices in the linear system which hold large sparsity. This banded sparsity is deterministic and can be optimally stored using specific matrix storage formats as DIAG. Solving the $Ax = b$ linear system requires any of the different solver algorithms available, either the classical iterative (CG or GMRES) or direct (Gaussian or LU factorization). By solving this system, we finally propagate the solution one time-step (Δt).

So far, we have seen how to formulate the heat equation as an implicit problem, where linear algebra problems arise. Actually, it is the backward difference discretization in time which leads to implicit schemes. Then, by discretizing the Equations 2.3 and 2.4 using a first-order forward differences in time (FTCS) instead of backward differences, we obtain the following discretization of the heat equation:

$$\frac{1}{\Delta t}(u^{t+1} - u^t) = \frac{\alpha}{\Delta x^2}(u_{i-1}^t - 2u_i^t + u_{i+1}^t) \tag{2.9}$$

where the only changed terms are the time superscripts of the left unknowns ($u^{t+1} - u^t$). This other way of discretizing in time produces the following explicit scheme for the heat equation,

$$u_i^{t+1} = ru_{i-1}^t - (1+2r)u_i^t + ru_{i+1}^t, \quad r = \frac{\alpha \Delta t}{\Delta x^2} \tag{2.10}$$

which can be solved explicitly due to the knowledge of all its dependences. While the implicit scheme leads to a set of algebraic equations, the explicit scheme results in a formula that gives us the solution without the requirement of an iterative or direct solver. This explicit formula that performs nearest neighbor computations is called *stencil*, and gives us the solution of the PDE for the next time-step.

In a stencil computation, each point of the structured grid is traversed and updated with the weighted contributions from a subset of neighbors points over each direction of the cartesian axis in space ($u_{i\pm1}^t$) and time (u^{t-1}). Figure 2.3.right shows the 5 point grid example with an explicit scheme where all dependencies are known (u^t). The structured grid points represent the unknowns (u_i^t) in space for the PDE problem to be solved in time-step t. The number of neighbors, their coefficients ($-(1+2r)$ and r in Equation 2.10) and the discretization (Δx) set the order accuracy of the spatial differential operator. These coefficients may be the same for each point of the grid or may not be. This is often related to uniformly spaced meshes (constant coefficients) and rectilinear and curvilinear meshes (variable coefficients), where discretization over the cartesian axis vary for each point of the grid.

As in the implicit scheme, specific boundary conditions must be applied to boundary nodes (u_0 and u_4 in 5 point grid example) so as to confine the problem to the discretized grid. To do this, explicit schemes incorporate to the computational domain the so-called *ghost points* (see pink outer area in Figure 2.4.Left). This technique is two-fold, first it intends to impose those boundary nodes to an specific boundary condition (e.g. Dirichlet), and second it allows to use the same stencil operator throughout the whole computational domain.

In order to ensure numerical stability, explicit methods must conform the Courant-Friedrichs-Levy (CFL) condition. This condition ensures no numerical instabilities during the time-marching of the explicit solver by setting the time-step (Δt) to a sufficient small value. The timestep must assure that the numerical domain dependence of any point in space and time must include the analytical domain of dependence. In particular, the CFL condition for the 1D heat equation using FTCS (2nd order in space and 1st in time) is

$$r = \frac{\alpha \Delta t}{\Delta x^2} < \frac{1}{2}, \quad \Delta t < \frac{\Delta x^2}{2\alpha}. \tag{2.11}$$

Once the critical time-step has been computed, we have a well-defined problem that can be solved numerically. The FTCS scheme is easy to implement in parallel because u_i^{t+1} values can be updated independently from each other and only depend on older steps (u_i^t). The entire solution of the PDE is conducted in two nested loops, an outer loop over all time-steps, and an inner loop that traverses all grid points.

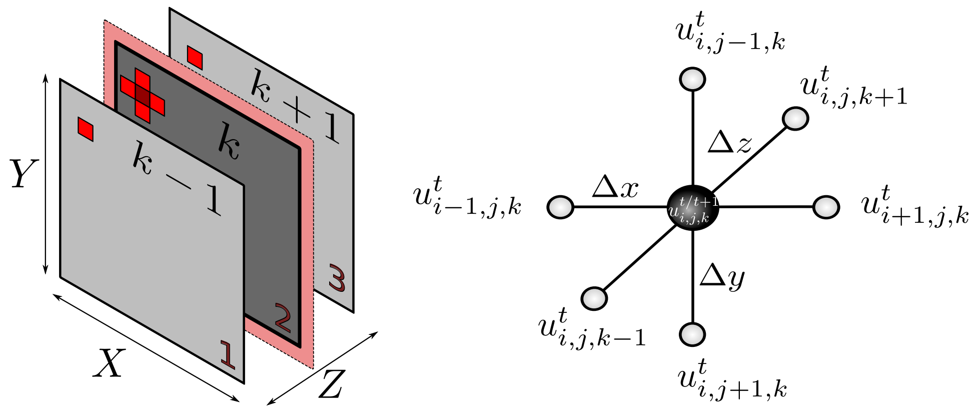


Figure 2.4: 7-point stencil layout. Left: Data involved in the computation of a 7-point stencil. The layout structure is totally different to 1D or 2D stencil computations. The pink halo in plane k depicts the *ghost points* required to perform the boundary conditions. Right: structure of the 7-point stencil. Data is accessed in the three axes.

Unlike the 1D example exposed here on account of simplicity, stencil computations are usually performed on 3D domains, on which this thesis is specially focused. In 3D stencils, the working dataset required is much bigger than in 1D or 2D problems (see Figure 2.4.Right). For example, the formulation of the 7-point 3D stencil operator (FTCS scheme: 2nd order in space and 1st in time) for the heat equation problem is the following,

$$\begin{aligned} u^{t+1}_{i,j,k} &= (1-2r_i-2r_j-2r_k)u^t_i + r_i(u^t_{i-1,j,k}+u^t_{i+1,j,k}) \\ &+ r_j(u^t_{i,j-1,k}+u^t_{i,j+1,k}) + r_k(u^t_{i,j,k-1}+u^t_{i,j,k+1}), \\ r_i &= \frac{\alpha\Delta t}{\Delta x^2},\ r_j = \frac{\alpha\Delta t}{\Delta y^2},\ \ r_k = \frac{\alpha\Delta t}{\Delta z^2}. \end{aligned} \tag{2.12}$$

Two main problems can be inferred from 3D computations. The first one is the memory access pattern. Accesses across the j and k indexes may be significantly more expensive in a hierarchy cache architecture. The spatial order of the stencil has direct impact on this problem. The more neighbors are accessed, the more expensive is in latency-wise, in particular for the k index which accesses over the least-stride dimension. The second stencil problem is the low ratio of floating point operations to memory accesses, which is related to the poor data reuse. This ratio is the result of dividing the floating-points instructions per load and stores accesses of u^t and u^{t-1} data grids. These two problems provoke that 3D stencil computations are often memory-bound, requiring special algorithmic treatments to sort out these issues.

2.4 Summary

This chapter has introduced a big picture of the most important numerical methods that can be found today. We have paid special attention to Finite Difference scheme, to which this thesis is devoted. As shown, this scheme is specially well suited to solve problems that can be represented in structured meshes in a reasonable execution time. We have also dedicated a section to explain the difference between implicit and explicit methods to solve numerically PDEs. Through a simple PDE problem as the heat equation, we have shown step by step how to deduce the linear system equations for implicit methods but also the nearest neighbor formula (*stencil*) for explicit methods. In order to obtain well-defined problems, we have dealt with the boundary conditions, and how to apply them to the boundary nodes of the domain to confine the problem to the discretized grid. Finally, we have also explained the difference of solving 1D or 2D against 3D stencil computations in terms of memory access complexity, unveiling the main computational problems that scientists and developers must face.

Chapter 3

Experimental Setup

In this chapter, the testbeds for the experiments are presented and detailed at each hardware and software implementation level. From the point of view of the hardware, all the peculiarities of the architectures and their memory hierarchy systems used in this thesis are discussed. Likewise, the software stack chosen for each platform to sort out aspects such as the parallel programming model, programming language and compiler versions are also addressed. Finally, we explain the details about the methodology considered to collect the performance results for the experiments and the instrumentation support used to measure the metrics.

3.1 Architecture Overview

The architecture variety used for the elaboration of this thesis is rather extent (see Tables 3.1 and 3.2 for further details about their specifications). Many types of architectures have been covered to assess the effectiveness of the optimizations proposed in most of the platforms available nowadays. The architectures analysed range from heterogeneous architectures with low latency scratchpad memories to homogeneous architectures with tenths of cores, which are able to run multiple threads simultaneously, and complex multi-level cache hierarchies that sport software and hardware prefetching capabilities.

3.1.1 Intel Xeon X5570 (Nehalem-EP)

The Intel Nehalem chip is the next multi-core processor generation of the Intel Core family, which includes technologies that bestow better performance compared to its predecessors. The main difference is the replacement of the front side bus architecture, and the integration of a memory controller (IMC) in the same die. Due to this new on-chip integration, an inter-chip network connection, called Quick Path Interconnect (QPI), is required for the multi-socket implementation. The QPI is also in charge of conducting the coherence protocol and the access to the I/O devices. This technology revamps the memory performance shortening the processor and memory performance gap.

As multi-socket architecture, memory can be attached to different processors, and therefore the cost of accessing data from one processor to the neighbor processor's memory has a higher penalty. This memory organization is called *non-uniform memory access* (NUMA), and requires careful consideration for the programmer in order to minimize remote memory accesses. However, the memory performance improvement overcomes the cost of the additional specific code.

The Intel Xeon 5500 series includes a quad-core processor with Simultaneous Multi-Threading (SMT) capabilities (see Figure 3.1). Each core has 2 independent FP units (FPADD and FPMUL) and is able to compute 8 floating points in SIMD mode (SSE) for single-precision and 4 floating points in double-precision. This gives a peak performance of 46.88 GFlops and 23.44 GFlops respectively for balanced codes (parity of adds and multiplications).

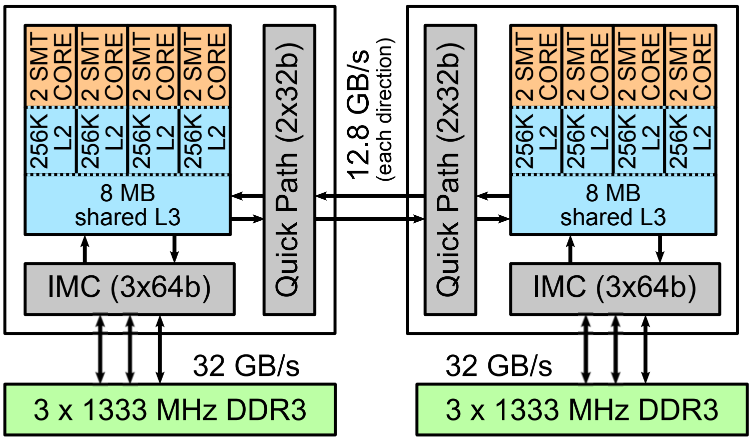


Figure 3.1: Diagram of a dual-socket configuration with Intel Nehalem-EP.

3.1.2 IBM POWER6

IBM introduced the POWER6 architecture in 2007. This was a higher improvement over the POWER5 version which was based on the two-way Simultaneous Multi-Threading (SMT) and dual-core technologies that were proven in its predecessor. In this version, IBM coupled high-frequency cores with a massive cache hierarchy and a complex memory subsystem specifically tuned for high-end computing.

The POWER6 processor incorporates two high-frequency (up to 5.0 GHz) two-way SMT cores, which are able to operate in 32- and 64-bit modes. Each core includes a 64 KB L1-D and L1-I caches, and a private on-chip 4 MB L2 cache. Finally a 32 MB L3 victim cache located off-chip is shared among the two cores. Each core also contains a sophisticated prefetching engine that reduces the effective latency by detecting 16 independent streams on L1 and L2 caches for reading and writing operations.

Likewise, two integrated memory controllers and an integrated SMP coherence and data interconnect switch are included in the same die. Both memory controllers can operate at 3.2 GHz, reading up to 16 bytes of data and simultaneously writing up to 8 bytes per cycle (peaks of 51.2 GB/s and 25.6 GB/s in read/write operations) when using 800 MHz DDR2

DRAM. Besides, the SMP switch enables scalable connectivity for up to 32 POWER6 chips through chip-to-chip and node-to-node links.

System name	Bull NovaScale	IBM Server	IBM System	Cray XT5
Architecture	Intel Nehalem-EP	IBM POWER6	IBM PowerPC 450	AMD Opteron
Model	X5570	p-575	BlueGene/P	2356
Execution	out-of-order	in-order	in-order	out-of-order
SMT	2	2	1	1
SIMD	SSE4.2 128-bit	VMX 128-bit	HUMMER 128-bit	SSE4.2 128-bit
ISA	amd64	POWER	PowerPC	amd64
Chips$\times$Cores	2×4	16×2	1×4	2×4
Clock (GHz)	2.93	4.7	0.850	2.3
SP GFlops†	93.76	75.2	13.6	73.6
DP GFlops†	46.88	37.6	13.6	36.8
L1 D-Cache	32 KB	64 KB	32 KB	64 KB
L2 Cache	256 KB	4096 KB	1920 B	512 KB
L3 Cache	8 MB (inclusive)	32 MB (victim)	2×4 MB	2 MB (victim)
Main memory	24 GB	128 GB	2 GB	8 GB
Bandwidth	32 GB/s	51.2 GB/s	13.6 GB/s	21.3 GB/s
Watts $\times$ hour†	95	100	39	75
Compiler	Intel v11.1 GCC v4.1.2	IBM XL v12.1/v10.01	IBM XL v9.0/v11.01	Portland v10.2 GCC v4.1.2

Table 3.1: Architectural configuration of the platforms used. †Only one chip is considered.

3.1.3 IBM Blue Gene/P

The IBM BG/P architecture, the successor of BG/L, was conceived to emphasize the high-power efficiency in platforms while keeping the race to the future petascale energy-efficient supercomputers. The BG/P version maintains the application-specific integrated circuit (ASIC) and the 3D torus interconnection topology of the BG/L, but incorporating significant important enhancements that boost the throughput of this architecture.

The BG/P die includes four IBM PowerPC 450 32-bit cores running at 850 MHz. Each core contains a dual-pipeline FPU, that supports two simultaneous floating-point operations in SIMD mode (either single or double precision). These units are able to conduct fused multiple-add operations (2 FLOPs), conferring a peak throughput of 13.6 GFlops per chip. In addition, each PPC450 core sports a 32 KB L1-I and L1-D caches, a private L2 stream buffer on-chip able to keep 7 streams simultaneously (2 cache-lines per stream), and a 8 MB L3 cache shared among the cores. All four cores are interconnected with the DDR2 main memory through a 16 byte wide dual-channel able to keep a sustainable bandwidth of 13.6

GBytes/s. In order to fully utilize the compute node resources, this platform supports three execution modes: SMP mode (one MPI task with up to four threads per process), dual mode (two MPI tasks with up to two threads per process) and quad mode (four MPI tasks with one single thread per process).

3.1.4 AMD Opteron (Barcelona)

The AMD Opteron codename Barcelona was the first AMD chip to combine in the same die a quad-core processor. In this NUMA architecture, each die include an integrated memory controller that uses the AMD HyperTransport (HT) technology for point-to-point communications between chips. Although the AMD Opteron is similar to Intel Nehalem in number of cores, clock rate and cache sizes, performance results can differ significantly. The lack of SMT parallelism and the number of channels of the memory controllers and their frequency are the main reasons of its lower performance.

Each Opteron core has a 32 KB L1-I and L1-D caches and a private 512 KB L2 cache, including also a 2 MB L3 victim cache shared among all four cores. It supports SSE4 ISA able to issue four double-precision floating-point operations per cycle. For the NUMA system, each socket is connected to a DDR2 667 MHz memory bank through the dual-channel (8 bytes wide) HT memory controller, thus delivering 10.66 GB/s per socket (21.3 GB/s aggregated).

3.1.5 IBM Cell/B.E.

The IBM Cell/B.E., shown in Figure 3.2, is a heterogeneous architecture with a multi-core chip composed of a general 64-bit PowerPC Processor Element (PPE) and 8 Synergistic Processor Elements (SPEs) running at 3.2 GHz. The PPE is the main processor capable of running the OS, context switching and logical codes, whereas the SPEs have 128-bit wide SIMD units exempt from caches, suitable for intensive computing. These SIMD units allow to process simultaneously four single-precision and two double-precision floating-point operations per cycle. In addition, each SPE instead of having caches has direct access to a small but low latency scratchpad memory of 256 KB, called Local Store (LS).

Another important feature of the Cell/B.E. is the Element Interconnect Bus (EIB). The EIB is a high speed bus that connects all components together and enables efficient memory transfers between the PPE, the SPEs (up to 200 GB/s) and the main memory (up to 25.6 GB/s). It is composed of four circular data rings (16 bytes wide) and each of them can handle up to 3 concurrent transfers (12 simultaneous transfers). Each SPE has a dedicated DMA controller for transferring data from and to the LS asynchronously. By using double-buffering techniques, the DMA controller can be used to overlap data transfers with computations, and can therefore potentially eliminate the memory latencies normally associated with traditional cores. Thus, in this architecture the memory management is completely explicit, and its burden is left to the programmer, who can take advantage for achieving efficient scheduling of computation and memory transfers.

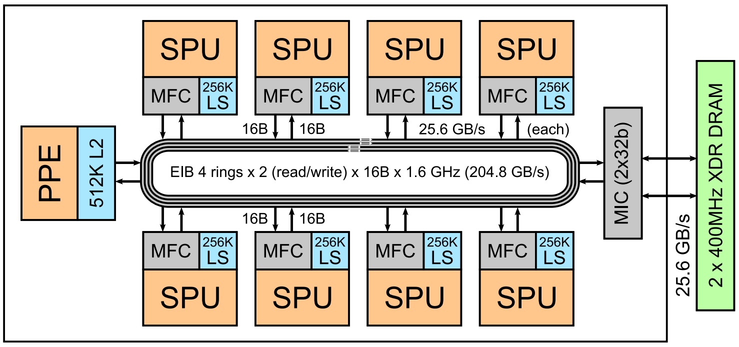


Figure 3.2: Diagram of the IBM Cell/B.E. processor.

3.1.6 IBM POWER7

The IBM BladeCenter PS701 node is equipped with an 8-core 64-bit POWER7 processor operating at 3.0 GHz. Each core contains 4 Floating-Point units, which are able to perform fused multiply-add (FMA) instructions in double precision. These features give an impressive 192 GFlops of peak performance per socket. Additionally, a 32KB L1 data cache and a 256KB L2 cache are included on each core, and an on-chip 32MB L3 adaptative victim cache is shared among all eight cores. The POWER7 socket processor has eight DDR3-1066 RDIMM memory channels. Each channel operates at 6.4 Gbps and can address up to 32 GB of memory. The RDIMM capacity of the PS701 system examined in our studies is 128 GB (16×8).

IBM has vastly increased the parallel processing capabilities of POWER7 cores by enabling a 4-way SMT (simultaneous multi-threading) system. Therefore, up to 32 threads can run at the same time per chip, leading to a quasi-massive parallel processor. This high thread level parallelism makes the POWER7 platform a very appealing target to test the OpenMP versions of our benchmarks. Our PS701 node only includes one POWER7 chip; therefore no NUMA-aware code is required since all the data is allocated in a single memory bank. Otherwise, data should be initialized in parallel by all threads due to the first touch page mapping policy. In this way, the dataset memory region would be pinned to its corresponding socket.

3.1.7 Intel Xeon E5-2670 (Sandy Bridge-EP)

The Intel Sandy Bridge chip is the successor of the Intel Nehalem family, that includes a higher SMP degree and lower latency and larger caches than its predecessor. Our Intel Xeon system is based on the E5-2670 Sandy Bridge-EP chip, a high-end efficient performance processor for servers.

This system has two sockets connected through QPI links (NUMA configuration), where each socket holds eight 2-way SMT cores running at 2.6 GHz. As in the Nehalem architecture, the core memory hierarchy is composed of a 32 KB L1-I, a 32 KB L1-D and a 256 KB L2 caches on-chip. However, the L3 cache is much larger than in the Nehalem with 20 MB and is linked

System name	IBM BladeCenter	IBM BladeCenter	IBM System X	Intel Xeon Phi
Architecture	IBM Cell/B.E.	IBM POWER7	Intel Sandy Bridge	Intel MIC
Model	QS22	PS701	E5-2670	KC (SE10P)
Execution	in-order	out-of-order	out-of-order	in-order
SMT	1	4	2	4
SIMD	VMX/SIMD 128-bit	VMX/SIMD 128-bit	AVX 256-bit	AVX512 512-bit
ISA	PowerPC/SPU	POWER	amd64	x86-64-k1om
Chips×Cores	$2 \times 1 + 8$	1×8	2×8	1×61
Clock (GHz)	3.2	3.0	2.6	1.1
SP GFlops†	204.8	384 (w/o VMX)	332.8	2129.6
DP GFlops†	102.4	192 (w/o VMX)	166.4	1064.8
Scratchpad	256 KB	-	-	-
L1 D-Cache	32 KB	32 KB	32 KB	32 KB
L2 Cache	256 KB	256 KB	256 KB	512 KB
L3 Cache	-	32 MB	20 MB	-
Main memory	8 GB	128 GB	32 GB	8 GB
DRAM type	XDR	RDIMM-DDR3	DDR3-1600	GDDR5-2750
Bandwidth	25.6 GB/s	102.4 GB/s	51.2 GB/s	352 GB/s
Watts × hour†	92	170	115	300
Compiler	IBM XL v9.0 GCC v4.1.1	IBM XL v12.1/v10.01	Intel v13 GCC v4.7	Intel v14

Table 3.2: Architectural configuration of the platforms used. †Only one chip is considered.

to the cores through a bi-directional ring interconnection. The sockets are connected to the DDR3-1600 main memory through four-channel memory controllers (4x64b) in order to provide better performance for 8 core chips (theoretical peak of 51.2 GB/s).

A new feature of this Intel architecture is the support of AVX, a 256-bit wide SIMD instruction set that can compute 4 double-precision FP instructions per cycle. This new ISA confers an astonishing peak performance of 166.4 GFlops per chip when dealing with mul-add balanced numerical codes in double-precision. Finally, it also includes a complex hardware prefetcher system able to prefetch data to the L1 cache (DCU and IP, 2 streams in total), and to the L2 and L3 through the streamer and spatial prefetcher mechanism (up to 32 streams are keep simultaneously).

3.1.8 Intel Xeon Phi (MIC)

In 2011 Intel released the first stable Many Integrated Core architecture (MIC), known as Intel Xeon Phi product family, incorporating the research work of the discontinued Larrabee many-core architecture. Since then, Intel has released MIC versions with different number of cores. Our two 22 nm MIC (Knight Corner) prototypes include 60 and 61 cores with 4-

way SMT capabilities running at 1.05 and 1.1 GHz (see Figure 3.3). Each core contains an in-order execution pipeline and a 512-bit wide vector unit (VPU) that performs 8 double-precision fused multiply-add operations (8×2 FLOPs) per cycle. This many-core architecture was initially designed for crunching numbers conferring the impressive aggregated peak performance of 1.07 TFLOPs per chip in SIMD mode.

The MIC memory hierarchy is composed of a 32 KB L1-D cache, a 32 KB L1-I cache and a private 512 KB L2 cache per core. The L2 cache also includes a hardware prefetcher able to prefetch 16 forward or backward sequential streams into 4KB page-size boundaries. All cores are connected together and to 8 GB of main memory through their L2 caches via a bi-directional ring bus with the MESI protocol for maintaining the coherence state. This mechanism enables to access other private L2 cache units from any core, providing a shared L2 cache of 30 MB per chip.

The high TLP of our MICs (up to 4-way SMT and 61 cores) offers a vast number of parallelism configurations regarding the work-balance distribution among threads (up to 244 threads) and their pinning to cores. In order to ease this task, the MIC runtime environment offers new scheduling options for thread affinity settings, such as: scatter, compact and balanced.

The initial versions of the MIC device were intended to be used as a coprocessor, and therefore are required to be attached to a host system via the PCI express bus. Even though, the MIC platform supports two execution models: native (run from the MIC device) and offload (run from the host). The former is accessed via the Intel compiler flag `-mmic` which produces an executable targeted specifically for MIC. The latter is based on specific language pragmas that allow the programmer to specify which data must be transferred back and forth from the host to the MIC device.

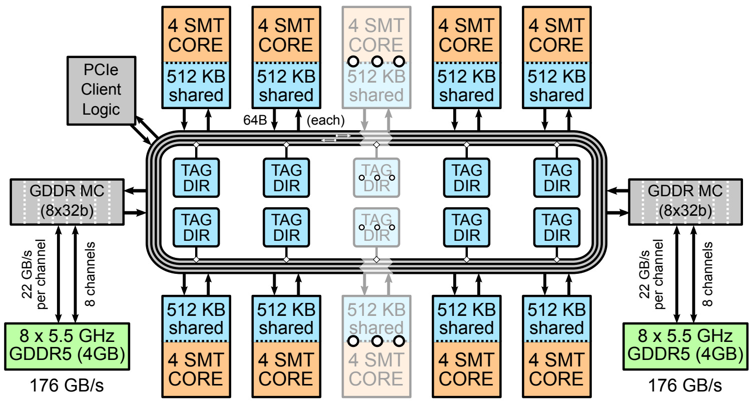


Figure 3.3: Diagram of the Intel Xeon Phi processor.

3.2 Parallel Programming Models

This thesis does not endeavour to evaluate the myriad alternatives of the programming models. The intent is simply to select the appropriate programming model that maximizes the performance in clusters of nodes with multi-core processors and several sockets. To succeed in this task, the parallelization must be conducted at two different levels: node-level (intra-node) and cluster-level (inter-node).

At intra-node level, where the shared memory within the node is exploited, two parallel programming models dominate the arena: the POSIX Threads (Pthreads) and the OpenMP standard. These two models take advantage of the shared memory to communicate data across threads. Pthreads is a POSIX standard for threads that defines an API for creating and manipulating threads on UNIX-like operating systems. The advantage of Pthreads over OpenMP is the explicit control that the user has over the lightweight created threads and the pinning to the computational resources (affinity). On the other hand, OpenMP standard permits a fast prototyping through specific clauses that allow work-sharing constructs of computational areas at the cost of less thread control and flexibility. Users must annotate their codes with pragmas that specify how computational loops or regions should be parallelized. Lately, some OpenMP implementations have improved the thread control through new affinity variables.

At cluster-level (inter-node), MPI is a message passing library that has become the *de facto* standard for distributed memory systems. The MPI implementation is much costly than Pthreads or OpenMP. The MPI API spawns processes instead of threads on each node (Single Program Multiple Data - SPMD), that run the same program with their own address space where communication between nodes occurs explicitly through two-sided messaging. The SPMD parallelism adds memory overheads, and thus its use is only recommended to perform domain decomposition at cluster-level where shared memory paradigm is not possible. The parallelization of the explicit FD solver within the cluster requires *boundary points* (see Figure 3.4), which are additional data that must be allocated in each node to ensure consistency across nodes. Therefore, every computational node must exchange (*Send/Receive*) the boundary areas with their neighbors.

As we will show in Chapter 7, high performance applications can be obtained by the combination of the three previous parallel paradigms. Although it seems to be counter-productive to use both shared memory models at the same time (Pthreads and OpenMP), it can be beneficial when they are carefully combined for specific tasks. In this regard, we have employed the three parallel models as follows. First, Pthreads are used to create a set of independent threads in charge of dissimilar tasks that encompass computation, communication and I/O. Second, OpenMP threads are spawned at these regions or kernels of the computation threads that are compute-intensive. This is the primary mechanism that is used for loop parallelism constructs. Finally, MPI is explicitly used in communication threads to conduct the message passing and to scale up to several number of nodes. The flexibility of POSIX threads allows

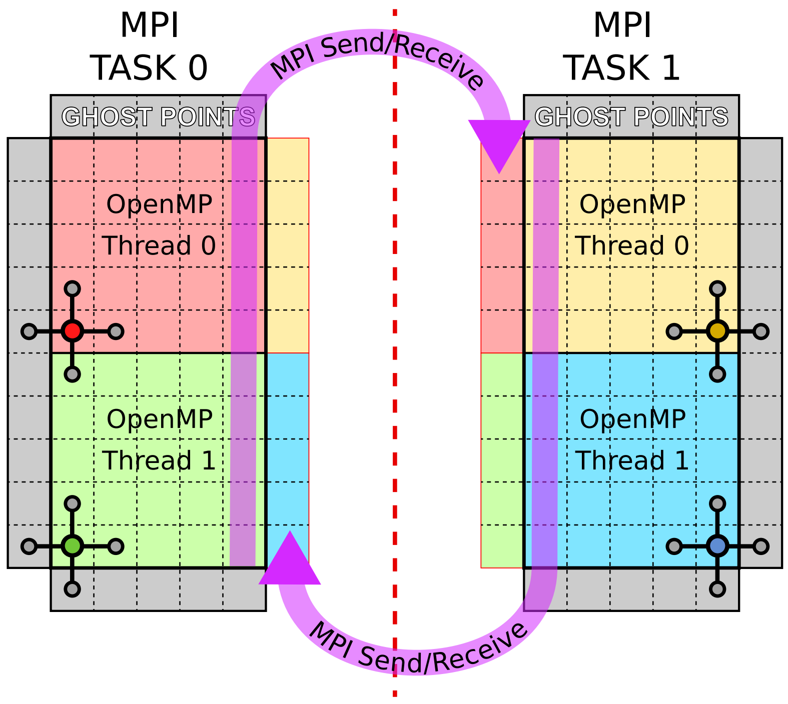


Figure 3.4: Shared and distributed memory layout using OpenMP and MPI APIs. In this case, we have an SPMD execution with 2 MPI tasks, each with 2 threads managed by OpenMP. Ghost points are depicted in grey, whereas the remaining colors represent the thread owner of the computed point. The shared memory in each MPI task is arranged sequentially in memory, thus avoiding any replication of data across threads. In contrast, the distributed memory layout across MPI tasks requires message passing of the internal nodes adjacent to the boundary nodes.

us to have a high level of control and synchronize them in their tasks.

3.3 Programming Languages and Compilers

Two main programming languages are used for this thesis: Fortran and C. Fortran is a programming language that has been widely used in scientific codes. Most of the old numerical codes written in 60s and 70s were developed in this language. In contrast, C is a more flexible and low level programming language that allows us additional control. For instance, OpenMP and MPI APIs have runtime implementations of Fortran, not being the case of Pthreads, where only private and non-standard implementations can be found. The decision was to select C as preferred language when fine-grain control was required through the combination of Pthreads, OpenMP and MPI models. On the other hand, Fortran was selected in cases where this was not necessary.

The best optimization parameters that we are aware of have been used for each specific compiler. An analysis of the optimization flags was performed in advance to find out the most suitable in each case. This set of parameters includes `-O3` for GCC compiler, `-O3`, `-O4` and `-O5` for IBM XL compiler and `-fast` option for Intel compiler. Moreover, the most recommended architectural optimizations for the instruction set and the cache configuration were employed. Options such as `450`, `450d` and `pwr6` were used for `-qarch` and `-qtune` flags in BlueGene/P and POWER6 architectures respectively, whereas `-mSSEx` flag, `core2` and `opteron` options for `-mcpu`, `-mtune` and `-xHost` flags were utilized in GCC and Intel compilers for x86 architectures. The additional `-mmic` flag was used for Intel Xeon Phi in order to enable cross compiling in the host architecture.

Our aim was not to obtain the fastest implementation of each proposed algorithm or binary, but to show that the proposed methods are valid strategies to boost the numerical codes that are led by explicit FD methods.

3.4 Performance Measurement

Two sets of data have been collected for evaluation from these platforms: execution times and hardware counters. The former allows to compare the performance of our different implementations; while the latter gives an accurate profiling picture of the underlying events, such as the use of the CPU pipelines and the memory hierarchy transfers. Some of the counters measured are: floating-point operations, total cycles & instructions, load & store issued, and cache misses, hits & accesses for L1, L2 and L3 levels. These counters permit measurement of some interesting metrics such as: GFlops, *FP/Cache* ratio and actual memory traffic, among others.

In order to gather the hardware counter profiling, the following tools and frameworks have been deployed in the testbed systems:

- **PapiEx (PAPI)**: PapiEx is a performance analysis tool designed to measure transparently and passively the hardware performance counters of an application using the PAPI [61] framework.

- **LikwidPerfCtr (Likwid)**: LikwidPerfCtr [83] is a lightweight command tool to measure hardware performance counters on Intel and AMD processors using the Linux *msr* module.

- **hpmcount/libhpm (HPCT/HPM)**: hpmcount and libhpm provide hardware metric and resource utilization statistics after application execution. They are developed by IBM to support Power-based systems.

- **Valgrind:** Valgrind is an instrumentation framework for building dynamic analysis tools. It contains a set of tools to perform debugging and profiling. It also contains a cache simulator which can be used to build a memory footprint map of a given application.

- **Extrae:** Extrae [1] is a dynamic instrumentation package to trace programs with shared memory models (OpenMP and Pthreads) and message passing (MPI). It generates trace files that can be visualized with Paraver [70].

The information gathered through these tools is used as reference data in the validation and evaluation process. However, some steps are necessary to ensure reliable and accurate metrics. First, tests must be executed several times to minimize collateral effects from any artifact. Second, cache hierarchy must be cleared during each test repetition to avoid cache-line hits due to hot cache effects. Finally, in high-order stencils, a reduced set of coefficient terms ($C_{Z,X,Y}$) is used to prevent pollution of load and store metrics due to register spilling.

Cache misses gathered with these performance tools can be classified into three categories [79], also known as 3C's misses:

- Compulsory misses (*cold misses*) occur when a cache-line that has never been accessed before must be brought into the cache. Any accessed cache-line should be included in this category at the very beginning. Prefetching can partially hide the miss latency by eagerly fetching the cache-lines. Another way is to reduce the required dataset to reduce the memory footprint. A scenario with only compulsory misses sets the upper-bound of a memory-bound application where the maximum attainable performance is obtained.

- Capacity misses are produced on cache-lines that have been previously accessed but are no longer in cache. The accessed dataset is larger than the cache hierarchy, leading to evictions and reloads of data with a certain temporal locality. Capacity misses can be reduced by increasing the capacity of the cache hierarchy or by reorganizing the computation in such a way that accesses to cache-lines with certain temporal reuse are shorten in time.

- Conflict misses are caused by cache-lines that are mapped to the same cache location, therefore competing for cache-lines and entailing the *cache thrashing* or *ping-pong* effect. As a solution to mitigate this effect, we can increase the cache associativity, the layout of data in memory (e.g. using padding) and the access pattern.

3.5 STREAM2

In order to obtain memory bandwidth measures on the testbed architectures, we have used the synthetic STREAM2 [55] benchmark, which aims to extend the functionality of STREAM [54] in two aspects. First, it measures sustained bandwidth, providing accurate information across the memory hierarchy. Second, STREAM2 exposes more clearly the performance differences between reads and writes. It is based on the same ideas as STREAM, but a different set of vector kernels are used: FILL, COPY, DAXPY and DOT. Table 3.3 details the operations carried out in each STREAM2 kernel.

The intent of STREAM2 is to decouple the measurement of the memory subsystem from the hypothetical peak performance of the machine, obtaining the best attainable bandwidth over each hierarchy level using standard Fortran. In this respect, the benchmark is complementary to LINPACK, which is optimized to obtain the maximum computational performance in FLOPS, disregarding the performance of memory subsystems.

Kernel	Operation	Type	Bytes	FLOPs
FILL	`a(i) = 0`	WRITE	8	0
COPY	`a(i) = b(i)`	READ & WRITE	16	0
DAXPY	`a(i) = a(i) + q*b(i)`	READ & WRITE	24	2
DOT	`sum += a(i) * b(i)`	READ	16	1

Table 3.3: The four computational kernels in STREAM2 micro-benchmark.

The technique for measuring the memory bandwidth in STREAM2 is based on performing the bandwidth kernels over long vectors (size N in double precision). The array sizes are defined so that each array is large enough to exceed the whole cache hierarchy of the platform to be tested. The vector data is traversed and computed linearly so that data reuse is avoided. The maximum bandwidth attainable for each kernel is then computed by dividing the total bytes transferred across the memory hierarchy by the elapsed time ($N \times Bytes_{kernel}/sec$).

In order to obtain reliable cache and memory performance with STREAM2, the characterization must be performed using similar memory access conditions in terms of alignment, SIMD code, parallel context and temporal writes with respect to the application to analyze. As a consequence, during the progress of this thesis, the STREAM2 benchmark was extended adding new features that were required to characterize the testbed platforms. First, to build lower-bound models for the analyzed platforms, the benchmark was updated to capture non-streaming bandwidths of each cache level. Second, multi-core and SMT support were added through the OpenMP standard, resizing vector lengths to be multiple of number of threads. Third, to ease the SIMDization, arrays are carefully resized and aligned to natural vector-size boundary, assisted by auto-vectorization pragmas. Finally, specific pragmas to enable non-temporal writes (streaming stores through cache-bypass technique) were also included. Adding all the different extensions, our STREAM2 version supports a wide variety of execution modes, which are shown in Table 3.4.

Modifiers	SIMD	Aligned	Streaming store
`-DSSE -DAVX -DMIC`	✓	✓	✓
$\rightarrow$ `-DNONTEMPORAL`	✓	✓	✗
$\rightarrow$ `-DUNALIGNED`	✓	✗	✗
`-DNOVECTOR`	✗	✗	✗

Table 3.4: The different STREAM2 execution modes supported.

For the development of this thesis, DOT and FILL kernels have been mainly deployed to isolate read and write bandwidths over each cache hierarchy level. In addition, streamed and non-streamed bandwidths were also gathered for both kernels. Actually, obtaining non-streaming bandwidths can be complex depending on the underlying architecture. To succeed in this task, three strategies (alone or combined) have been employed. First, some compilers provide prefetch pragmas (`#pragma noprefetch`) that permit to disable software prefetching when this feature is present in the underlying architecture. Second, on some cases the processor manufacturer provides MSR (Model Specific Registers) specifications that allow the user to completely disable hardware prefetching. If none of the previous cases are available for a specific hardware, we devised an alternative strategy implemented through software that estimates this bandwidth by avoiding the triggering of hardware prefetchers. For this purpose, STREAM2 kernels were modified in such a way that a stride parameter ($offset$) is carefully set to prevent consecutive reads and writes, disabling the unit-stride prefetcher. Moreover, most modern architectures also support next-stride cacheline prefetching. Thus, we have used a combination of two techniques in DOT and FILL kernels to avoid triggering the prefetching engine. The stride parameter of STREAM2 kernels has been redefined as,

$$offset = \underbrace{stride}_{\text{adjacent}} + \underbrace{MOD(I, epsilon)}_{\text{stride}} \tag{3.1}$$

where $stride$ is a constant large enough to avoid next cacheline prefetching mechanism (adjacent). The second term $MOD(I, epsilon)$, which is a cyclic variable that depends on the I iteration of the loop and a constant $epsilon$, prevents the stride cacheline prefetcher engine to be triggered.

Finally, after selecting the proper execution mode and running the benchmark, the different cache level bandwidths can be deduced analyzing the STREAM2 output (see Figure 3.5). This is observed in the areas where bandwidth drops substantially due to cache overflow when vector size increases (X axis). Every plateau represents the attainable bandwidth for each cache level in the hierarchy.

3.6 *Prefetchers*

Modern computer architectures commonly incorporate prefetching engines in their cache hierarchy. Its aim is to reduce the memory latency by eagerly fetching data that is expected to be required in the near future. Consequently, in memory bound applications, hardware prefetchers play an important role in the overall performance. Therefore, the characterization of the prefetching mechanism is meaningful to understand the causes behind poor performance of codes such as stencil computation.

In order to characterize the prefetching mechanism, we developed from scratch *Prefetchers*, a micro-benchmark that measures the prefetching efficiency [52, 56]. This metric weighs the number of data fetched using the prefetcher mechanism with respect to the entire data

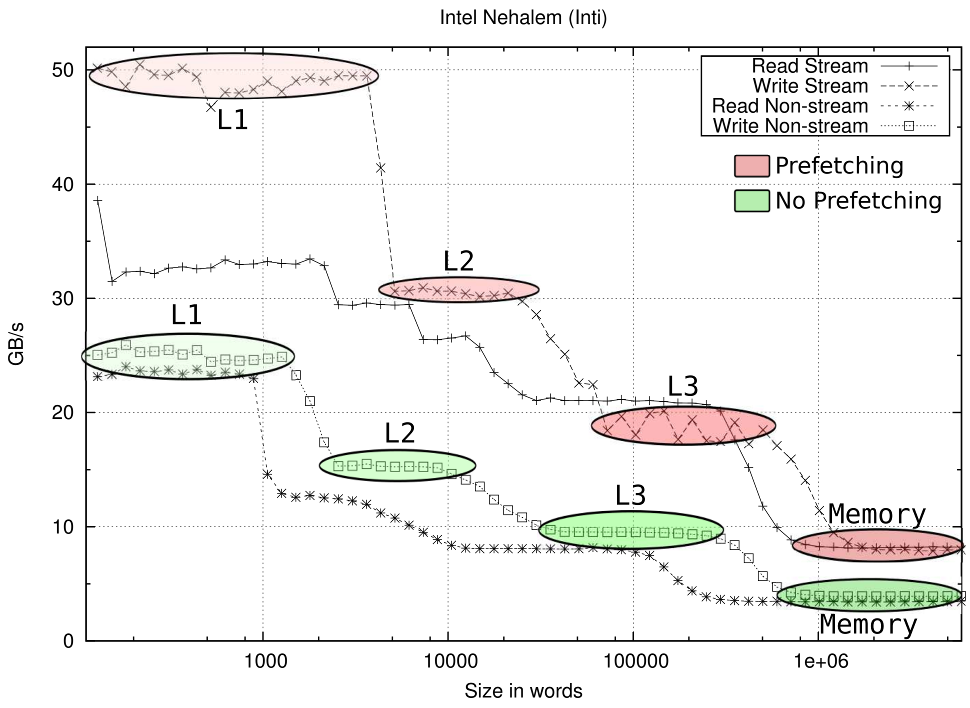


Figure 3.5: STREAM2 results of stream and non-stream bandwidths for read and write operations in the memory hierarchy of Intel Nehalem platform.

transferred through the whole memory accesses. As we will see in Chapter 6, this metric is a good predictor of the memory requests initiated by streaming prefetchers.

The technique used to gather this information is very similar to the one followed in STREAM2 benchmark; a large array is traversed accessing successive cachelines and thus triggering the hardware prefetchers. Then, the results are presented using hardware counters, collecting the cachelines read by the prefetcher and the total data requested through the hierarchy. Finally, the following formula is computed,

$$Prefetching_{efficiency} = prefetched/total\,. \tag{3.2}$$

Two kernels are implemented (see Table 3.5), one for reading and a second one for writing operations when write-allocate policy is present in the cache hierarchy. The former performs a cumulative sum over the vector to be streamed. The latter reverses this operation, filling the destination vector with incremental values that cannot be replaced with compiler intrinsics.

In multi-core and SMT architectures, where hardware prefetchers included in caches are shared among the processing units, the streaming effectiveness can vary drastically due to the limited number of data streams that hardware prefetchers can track simultaneously. Therefore, the term streaming concurrency, relating the parallel and logical data streams, is cap-

Kernel	Operation	Streams	Type
CUMUL	`sum = sum + a(`$stream_i^{thrd}$`)`	Up to 32	READ
INCFILL	`a(`$stream_i^{thrd}$`) = sum++`	Up to 32	WRITE

Table 3.5: The two computational kernels in Prefetchers micro-benchmark. INCFILL kernel can be used to verify whether writes are also affected by the prefetching engine.

tured by increasing the concurrent buffers computed in a round-robin fashion and observing its behavior. Figure 3.6 shows the pattern of how data is traversed to measure this concurrency. Using this access pattern, three characteristics can be deduced: the triggering of the prefetcher (TP), the prefetcher efficiency (PE) and the look-ahead (LAP). The former specifies the warm-up time required to trigger the engine. The latter indicates how many cache-lines are additionally fetched in advance.

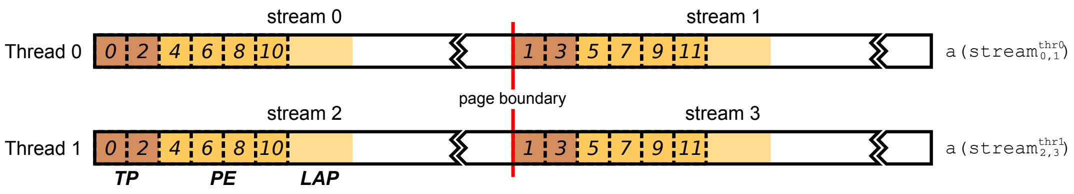


Figure 3.6: Round-robin access of streaming vectors among concurrent threads. In this example 4 streams are computed simultaneously by two threads (2-way SMT) accessing cache-lines in the depicted order.

In order to collect consistent information about the prefetchers, a couple facts must be considered. Firstly, cache sets are equally spaced along threads to avoid collided accesses that promote eviction of prefetched data before used (cache pollution). Secondly, buffers are aligned to page boundaries (e.g. 4 KB in GNU/Linux) preventing disruption of the hardware prefetcher.

3.7 The Roofline Model

An easy-to-understand and visual performance model is proposed through the *Roofline* [86]. It aims to improve parallel software and hardware for floating point computations by relating processor performance to off-chip memory traffic. To achieve that goal, the term *Operational Intensity* (OI) is defined as the operations carried out per byte of memory traffic (Flops/Byte). The total bytes accessed are those that hit the main memory after being filtered by the cache hierarchy. Therefore, OI measures the traffic between the caches and memory rather than between the processor and the caches. Thus, OI quantifies the DRAM bandwidth needed by a kernel on a particular architecture.

As shown in Figure 3.7, floating-point performance, OI and memory performance can be tied together in a two-dimensional graph with this model. The horizontal line shows peak floating-point performance of the given architecture, and therefore no kernel can exceed this since it is a hardware limit. The X axis depicts the GFlops/byte, whereas the Y axis shows the GFlops/second. The 45° angle of the graph represents the bytes per second metric, the ratio of (GFlops/second)/(GFlops/byte). As a consequence, a second line can be plotted, which gives the highest floating-point performance that the memory system can achieve for a given OI. Through these two lines, the model is completely roof-limited and therefore the performance limit can be obtained with the following formula:

$$\begin{aligned}\text{Attainable GFlops/second} = \text{MIN}(&\text{Peak Floating Point Performance},\\ &\text{Peak Memory Bandwidth} \times \text{Operational Intensity})\end{aligned}$$

The point where the two lines intersect is the peak computational performance and the peak memory bandwidth. It is important to bear in mind that the *Roofline* limits are fixed for each architecture and do not depend on the kernel that is being characterized.

The ceiling for any scientific kernel can be found through its OI. Drawing a vertical line on the X axis at the OI point, the maximum attainable performance for the tested architecture can be estimated from the intersection of the vertical line and the roof line.

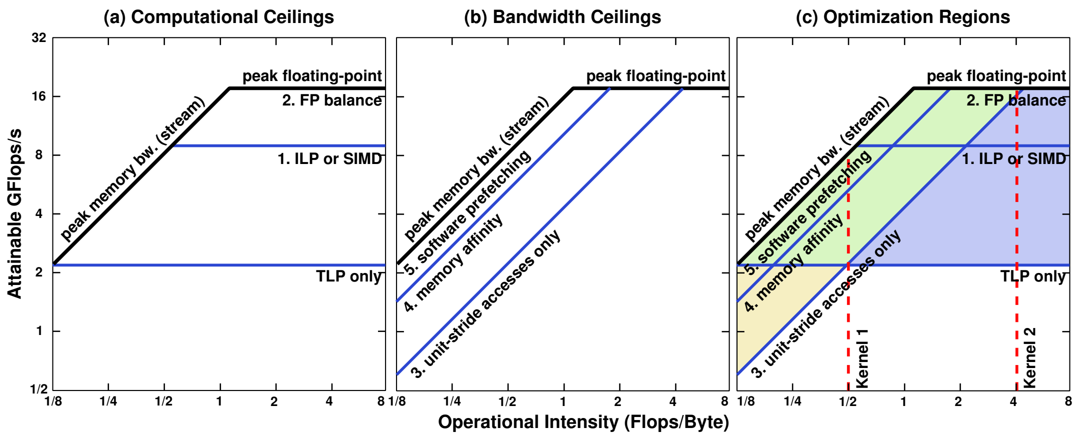


Figure 3.7: Roofline model example [86] with computational and bandwidth ceilings and its optimization regions.

The Roofline sets an upper bound in a kernel's performance depending on its OI. If the vertical projection of the line intersects the flat part of the roof, the kernel performance is compute bound, whereas it is memory bound if it intersects the sloped part of the roof. In addition, the model shows what the authors call the *ridge point*, where the horizontal and the 45° roof lines meet. The x-coordinate of this point provides an estimate of the minimum OI required by a kernel to achieve the maximum performance on an architecture. A *ridge point* shifted to the right implies that only kernels with a high OI can achieve the maximum performance. On the other hand, a *ridge point* shifted to the left on the axis means that most of the kernels can potentially obtain the maximum performance. Hence, the ridge point is related to the difficulty for programmers and compilers to achieve the peak performance in a given architecture.

3.8 The StencilProbe Micro-benchmark

In order to evaluate the performance of new stencil-based implementations, the StencilProbe [43], a compact, flexible and self-contained serial micro-benchmark, has been used. This micro-benchmark was developed as a proof of concept to explore the behavior of 3D stencil-based computations without endeavouring the modification of any actual application code. This tool is especially suitable for experimenting with different implementation states on cache-hierarchy architectures by rewriting the computations carried out in the inner loop of the benchmark. Indeed, the basic idea is to mimic explicit FD kernels from scientific applications, thus reproducing the same memory access patterns of application-based computational kernels.

Initially, the regular StencilProbe benchmark only included serial implementations of a low-order (1st order) heat diffusion solver derived from Chombo toolkit (heattut) [48]. This FD kernel, shown in Algorithm 2, was implemented using five different stencil-based algorithms: naive, space-blocking, time-skewing, cache-oblivious and circular queue algorithms.

Algorithm 2 *Heattut* kernel of Chombo toolkit. This is a low-order stencil (7-point), 2nd order in space (Central Difference) and 1st order in time (Backward Difference).

$$\begin{aligned}\mathcal{X}^{t}_{i,j,k} &= -6\mathcal{X}^{t-1}_{i,j,k}/factor^2 \\ &+ \mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k} + \mathcal{X}^{t-1}_{i,j-1,k} + \mathcal{X}^{t-1}_{i,j+1,k} + \mathcal{X}^{t-1}_{i,j,k-1} + \mathcal{X}^{t-1}_{i,j,k+1}\end{aligned}$$

For purposes of the current thesis, the StencilProbe infrastructure has been adapted and extended. In this way, this tool has allowed to verify and evaluate the contribution of new stencil optimizations and algorithms.

Four main changes were made to the micro-benchmark. First, five new cross-shape stencils were included for each algorithmic implementation, ranging from low-order to high-order stencils ($\ell = \{1, 2, 4, 7, 14\}$). This new benchmark version implements the stencil scheme shown in Algorithm 3, where star-like stencils with symetric and constant coefficients are computed using 1st order in time and different orders in space (depending on ℓ). Second, apart from the classical stencil implementation, two Semi-stencil algorithms were introduced (see Chapter 4): a full-axis (Z, X and Y) and a partial-axis (X and Y) version. Third, some straightforward pipeline optimizations related to the inner loop of the stencil were coded. These optimizations include loop fusion (computation carried out in one single loop) and loop fission (computation split into two and three loops). Finally, the StencilProbe micro-benchmark were extended to include the parallel implementation of each algorithm in or-

StencilProbe Features	Possible range of values
Stencil sizes (ℓ)	1, 2, 4, 7 and 14 (7, 13, 25, 43 and 85-point respectively)
SIMD code	SSE, AVX and AVX-512 intrinsics, and Intel auto-vectorization pragmas for scalar code
Store hints	Temporal and non-temporal stores
Memory alignment	Aligned and non-aligned read/writes
Prefetching	Enable/disable Hardware & Software prefetchers
Multi-core/SMT	OpenMP pragmas for parallelization
Decomposition policies	Static (either X or Y axis) and SMT Thread Affinity
Traversing algorithms	Naive, Rivera, Time-skewing and Cache-oblivious
Baseline algorithms	Classical and *Semi-stencil* (Partial X-Y, Full Z-X-Y axis)
Inner loop optimizations	Loop fusion (1 loop) and loop fission (2 and 3 loops)
Trace/HWC analysis	PAPI, Extrae and Likwid
Peak performance	Get the maximum performance for an implementation
Plot options	Pretty printing for plotting data
Trial options	Try several executions (min, max and average values)

Table 3.6: Features supported by the extended version of the StencilProbe. Some option combinations may depend on the architecture and the algorithm.

der to analyze their multi-core and SMT scalability. The shared-memory OpenMP API was used as the parallel programming paradigm. Other minor upgrades conducted into the micro-benchmark are the inclusion of SIMD code, memory alignment, and instrumentation support. Table 3.6 shows a summary of all the features of the new StencilProbe benchmark.

3.9 Summary

So far, in this chapter, we have established the experimental setup, presenting the architectures, software and benchmarks that will be used as tools to unveil and corroborate our findings. The following chapters will focus on detailing this thesis contributions.

Chapter 4

Optimizing Stencil Computations

Astrophysics [12], Geophysics [58, 65], Quantum Chemistry [4, 15] and Oceanography [39, 45] are examples of scientific fields where large computer simulations are frequently carried out. On these simulations, the physical models are represented by Partial Differential Equations (PDE) which can be solved by the Finite Difference (FD) method. The FD method uses stencil computations to obtain values for discrete differential operators. These large scale simulations may consume days of supercomputer time, and in particular most of the execution time is spent on the stencil computation. For instance, the Reverse-Time Migration (RTM) is a seismic imaging technique from Geophysics used in the Oil & Gas industry, where up to 80% of the RTM execution time [9] is spent on stencil computation.

The basic structure of stencil computation is that the central point accumulates the contribution of neighbor points in every axis of the cartesian system. The number of neighbor points in every axis relates to the accuracy level of the stencil, where more neighbor points lead to higher accuracy. The stencil computation is then repeated for every point in the computational domain, thus solving the spatial differential operator.

Two inherent problems can be identified from the structure of the stencil computation:

- First, the non-contiguous memory access pattern. In order to compute the central point of the stencil, a set of neighbors have to be accessed. Some of these neighbor points are distant in the memory hierarchy, requiring many cycles in latencies to be accessed [42, 43]. Furthermore, with increasing stencil order, it becomes more expensive to access the required data points.

- Second, the low computational-intensity and reuse ratios. After gathering the set of data points, just one central point is computed and only the accessed data points in the sweep direction might be reused for the computation of the next central point [36]. Thus, some of the accessed data are not reused and the current hierarchical memory is poorly exploited.

In order to deal with these issues and improve the stencil computation, we introduce the *Semi-stencil* algorithm. This algorithm changes the structure of the stencil computation, but it can be generally applied to most stencil-based problems. Indeed, the *Semi-stencil* algorithm computes half of the axis contributions required by several central points at the same loop iteration. By just accessing the points required to compute half of the stencil axis, this algorithm reduces the number of neighboring points loaded per iteration. At the same time, the number of floating point operations remains the same, but because the number of loads is reduced, the computation-access ratio is increased.

In this chapter, we present a comprehensive study of the *Semi-stencil* strategy on homogeneous multi-core architectures with hierarchical memories. Unlike other stencil optimization works [20, 21, 22] where a 7-point academic stencil (single Jacobi iteration) is tackled through auto-tuning environments, we perform an evaluation on medium and high-order stencils (from academic up to widely used in the industry). The aim of this research is not to achieve the fastest stencil implementation on the chosen platforms, but to prove the soundness of the *Semi-stencil* algorithm when scientific codes are optimized.

4.1 The Stencil Problem

The stencil computes the spatial differential operator, which is required to solve PDEs numerically on FD schemes. A multidimensional structured grid (often a huge 3D data structure) is traversed and the elements ($\mathcal{X}^t_{i,j,k}$) are updated with weighted contributions. Figure 4.1.b depicts a generic 3D stencil pattern, where the ℓ parameter represents the number of neighbors to be used in each direction of the Cartesian axis. Hence, the computation of a $\mathcal{X}^t_{i,j,k}$ point at time-step t requires ℓ weighted neighbors in each axis direction at the previous time-step $(t-1)$. The domain contains interior points (the solution domain) and ghost points (outside of the solution domain). This operation is repeated for every point in the solution domain, finally approximating the spatial differential operator.

The first problem identified from this computation is the sparse memory access pattern. Data is stored in Z-major order (unit-stride dimension), and therefore accesses across the other two axes (X and Y) may be significantly more expensive latency-wise (see Figure 4.1.a). The ℓ parameter has a direct impact on this problem, the larger the ℓ value the more neighbors at each axis have to be loaded to compute the $\mathcal{X}^t_{i,j,k}$ point.

The second problem to deal with is the low floating-point operations to data cache accesses ratio, which is related to the poor data reuse. In this work, we use the *FP/Cache* metric to quantify these issues; this is the ratio of floating-point operations to the number of loads and stores issued to L1 data cache for one $\mathcal{X}^t$ point computation:

$$\begin{aligned} FP/Cache_{Classical} &= \frac{FloatingPoint\ Operations}{Data\ Cache\ Accesses} = \frac{2 * MultiplyAdd\ Instructions}{\mathcal{X}^{t-1}\ Loads + \mathcal{X}^t\ Stores} \\ &= \frac{2*2*dim*\ell+1}{\underbrace{(2*(dim-1)*\ell+1)}_{\mathcal{X}^{t-1}\ Loads} + \underbrace{(1)}_{\mathcal{X}^t\ Stores}} = \frac{4*dim*\ell+1}{2*dim*\ell-2*\ell+2} \end{aligned} \tag{4.1}$$

Equation 4.1 states this metric for the classical stencil computation shown in Algorithm 3. dim is the number of dimensions of our PDE (3 in a 3D problem), where $2*\ell$ Multiply-Add instructions are required at each dimension. Furthermore, one extra multiplication operation

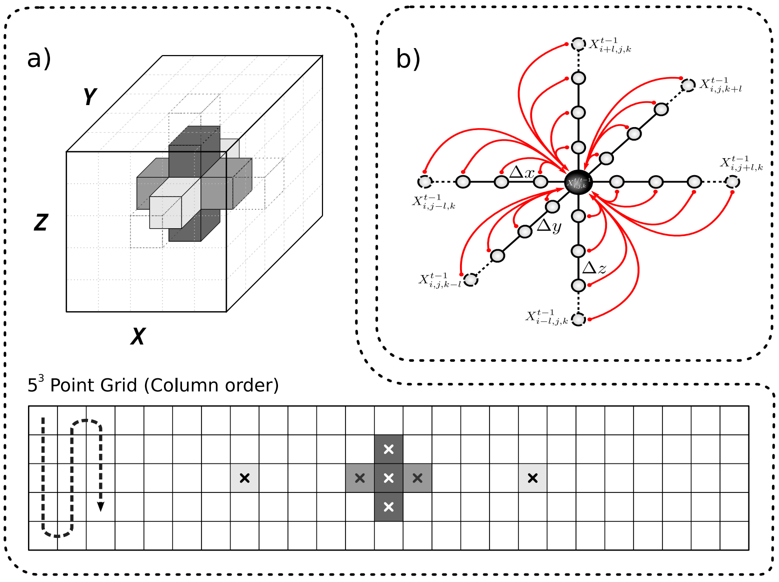


Figure 4.1: (a) The memory access pattern for a 7-point stencil. The sparsity of the required data to compute the stencil is higher in the last two axes (X and Y). (b) A 3D symmetric cross-shaped stencil. Each $\mathcal{X}^{t}_{i,j,k}$ point is updated with the weighted neighbors ($\mathcal{X}^{t-1}_{i\pm1..\ell,j\pm1..\ell,k\pm1..\ell}$) of the previous time-step ($t-1$).

must be considered for the self-contribution ($\mathcal{X}^{t-1}_{i,j,k}$). The number of loads needed to compute a stencil point differs depending on the axis. Those axes that are not unit-stride dimension (X and Y) require $2 * \ell$ loads at each loop iteration. However, the Z unit-stride dimension tends to require just one load due to the load reuse from preceding loop iterations. Finally, to conclude the $\mathcal{X}^{t}_{i,j,k}$ point computation requires one store to save the result.

As shown in Equation 4.1, the *FP/Cache* ratio depends on the dim and ℓ variables. Taking into account that ℓ is the only variable that may increase, the *FP/Cache* ratio tends to a factor of ≈ 3 flops per data cache (d-cache) access for 3D problems. This relatively low ratio along with some previous works' results [36, 53, 72] shows that the stencil computation is usually memory-bound. In conclusion, the execution time of the stencil computation is mainly dominated by the memory access latency.

These concerns force us to pay special attention to how data is accessed during the computation. It is crucial to improve the memory access pattern (by reducing the overall number of data transfers) and exploit the memory hierarchy as much as possible (by reducing the overall transfer latency). Section 4.2 reviews the main approaches that can be found in the literature to address these issues.

Algorithm 3 The classical stencil algorithm pseudo-code for a 3D problem. $\mathcal{X}^t$ is the space domain for time-step t, where Z, X and Y define the dimensions (ordered from unit to least-stride) of the datasets including ghost points (points outside of the solution domain). ℓ denotes the number of neighbors used for the central point contribution. $C_{Z1...Z\ell}$, $C_{X1...X\ell}$, $C_{Y1...Y\ell}$ are the spatial discretization coefficients for each dimension and C_0 for the self-contribution. z_s, z_e, x_s, x_e, y_s and y_e denote the area in $\mathcal{X}^t$ where the stencil operator is computed. In order to compute the entire dataset, the stencil pseudo-code must be called as follows: $\text{Stencil}(\mathcal{X}^t, \mathcal{X}^{t-1}, \ell, \ell, Z-\ell, \ell, X-\ell, \ell, Y-\ell)$. Notice that the discretization coefficients are considered constant.

```
1: procedure Stencil($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $\ell$, $z_s$, $z_e$, $x_s$, $x_e$, $y_s$, $y_e$)
2:     for $k = y_s$ to $y_e$ do
3:         for $j = x_s$ to $x_e$ do
4:             for $i = z_s$ to $z_e$ do
5:                 $\mathcal{X}^t_{i,j,k} = C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                       $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                       $+ C_{X1} * (\mathcal{X}^{t-1}_{i,j-1,k} + \mathcal{X}^{t-1}_{i,j+1,k}) + \ldots + C_{X\ell} * (\mathcal{X}^{t-1}_{i,j-\ell,k} + \mathcal{X}^{t-1}_{i,j+\ell,k})$
                       $+ C_{Y1} * (\mathcal{X}^{t-1}_{i,j,k-1} + \mathcal{X}^{t-1}_{i,j,k+1}) + \ldots + C_{Y\ell} * (\mathcal{X}^{t-1}_{i,j,k-\ell} + \mathcal{X}^{t-1}_{i,j,k+\ell})$
6:             end for
7:         end for
8:     end for
9: end procedure
```

4.2 State of the Art

Most of the contributions for stencil computations can be divided into three dissimilar groups: space blocking, time blocking and pipeline optimizations. Space and time blocking are related to tiling strategies widely used in multi-level cache hierarchy architectures. Pipeline optimizations refer to those optimization techniques that are used at the CPU pipeline level to improve the instruction throughput performance (IPC). In addition, these three groups contain incremental optimization techniques which can be combined with techniques from other groups.

Figure 4.2 categorizes these three groups by complexity of implementation (effort), benefit improvement (performance) and implementation tightness regarding hardware (dependency). Even though each algorithm property (effort, performance and dependency) can differ widely depending on the code complexity and the underlying hardware, this classification can be treated as the big picture of the state-of-the-art. In this diagram, an interesting optimization algorithm would be classified in the top-left corner, whereas an inefficient one would be in the bottom-right corner. The next subsections will review the advantages and disadvantages of these optimizations methods.

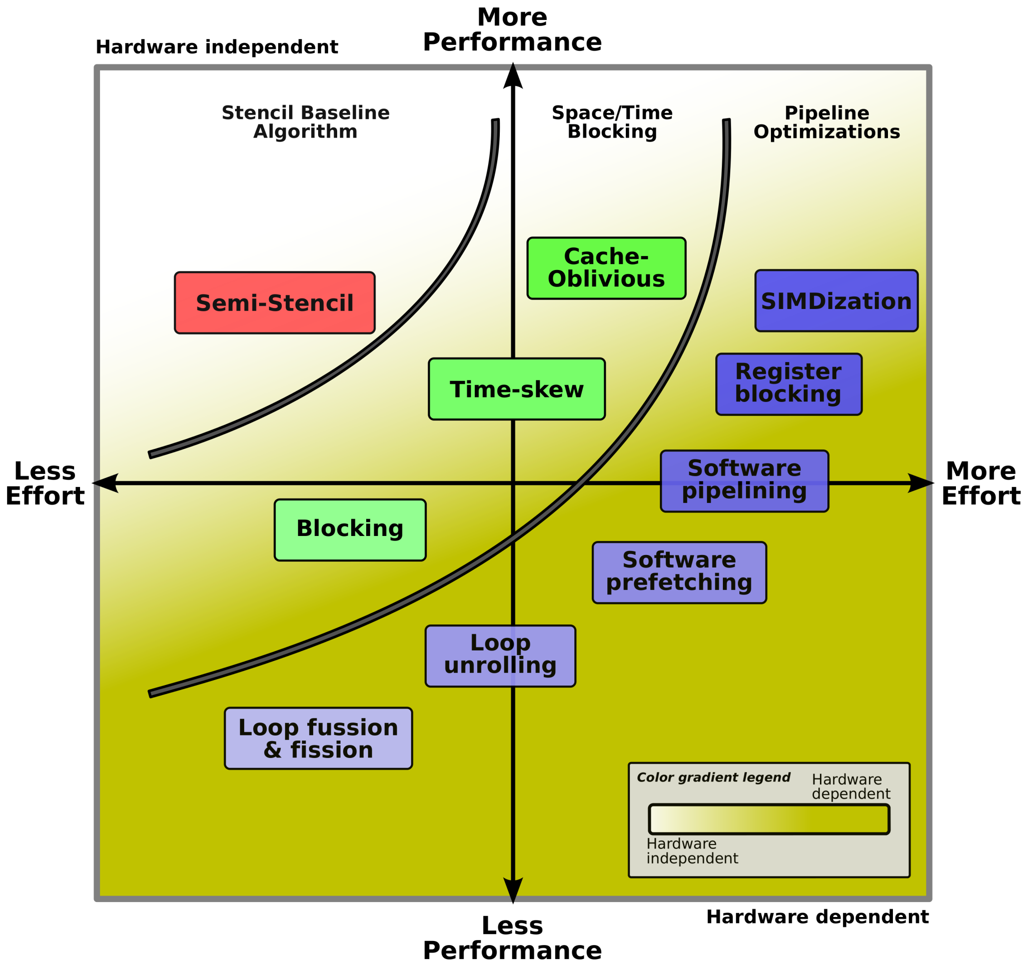


Figure 4.2: Characterization of different optimization schemes. This diagram sorts each optimization method according to the three properties: programming effort required to implement it, performance boost by optimizing the code and the dependency of the implementation with respect to the underlying hardware.

4.2.1 Space Blocking

Space blocking algorithms promote data reuse by traversing data in a specific order. Space blocking is especially useful when the dataset structure does not fit into the memory hierarchy. The most representative algorithms of this kind are:

- Tiling or blocking techniques: Rivera and Tseng [72] propose a generic blocking scheme for 3D stencil problems. The entire domain is divided into small blocks of size $TI \times TJ \times TK$ which must fit into the cache. Rivera and Tseng showed that a good blocking scheme configuration can be achieved when a $TI \times TJ$ 2D block is set along the less-stride dimension. Later, Kamil *et al.* [43] found out that the best configuration is usually given when TI is equal to the grid size I, as shown in Figure 4.3. This traversal order reduces cache misses in less-stride dimensions (X and Y), which may increase data locality and overall performance. Note that a search of the best block size parameters ($TI \times TJ$) must be performed for each problem size and architecture.

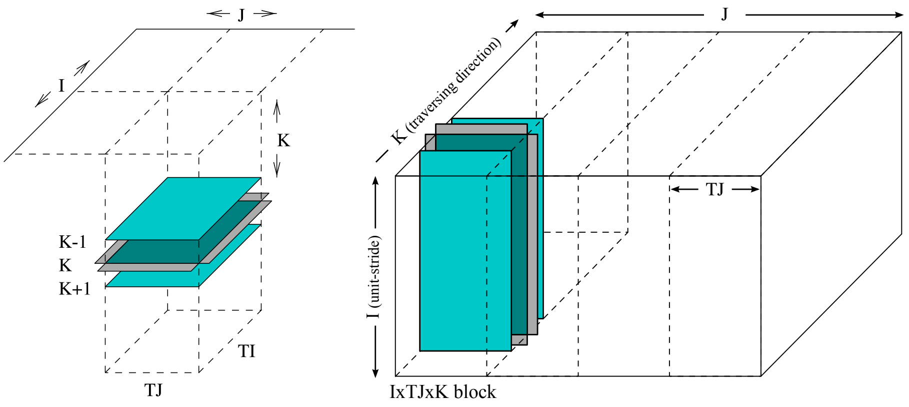


Figure 4.3: Left: 3D cache blocking proposed by Rivera, where $TI \times TJ$ cuts are computed sweeping through the unblocked dimension K. Right: $I \times TJ \times K$ blocking configuration suggested by Kamil, where the I and K dimensions are left unblocked. In this example, I and K are the unit-stride and the least-stride dimensions respectively.

- Circular queue: Data *et al.* [22] employ a separate queue data structure to perform the actual stencil calculations. This structure stores only as many 2D planes as are needed for the given stencil. After completing a plane, the pointer to the lowest read plane is moved to the new top read plane, thus making a *circular* queue. In general, the circular queue stores $(t-1)$ sets of planes, where t is the number of iterations performed.

4.2.2 Time Blocking

Time blocking algorithms perform loop unrolling over time-step sweeps to exploit the grid points as much as possible, and thus increase data reuse. Such techniques have shown some effectiveness in real infrastructures [48], but they require careful code design. However, due to the time dependency scenario, these techniques may pose implementation issues when other tasks must be carried out between stencil sweeps. For instance, boundary condition computation, intra- and extra-node communication or I/O may occur in many scientific applications during time-step updates. Time blocking techniques can be divided into explicit and implicit algorithms:

- Time-skewing: McCalpin and Wonnacott [53] algorithm try to reuse data in cache as much as possible. As a result, memory transfers are reduced and execution stalls are minimized. Essentially, several cache blocks of size $TI \times TJ \times TK$ are generated over space dimensions of the grid and each of those blocks is unrolled over time, as shown in Figure 4.4(a). To keep data dependencies of stencil computations, block computations must be executed in a specific order. This constraint makes the time-skewing algorithm only partially effective in parallel executions. However, time-skewing has been already parallelized [42, 87] with reasonable results. Wonnacott devised the parallel version of the time-skewing algorithm by performing space cuts in the least-stride dimension. Each thread block is divided into three parallelepipeds, which are computed in a specific order to preserve data dependencies and enable parallel computation. In this algorithm, as in space tiling, a search of the best block size parameters must be performed prior to the start of the computation.

- Cache-oblivious: Frigo and Strumpen [36] time blocking algorithm tiles both space and time domains. In contrast with time-skewing, the cache-oblivious algorithm does not require explicit information about the cache hierarchy. As Figures 4.4(b,c,d) show, an $(n + 1)$ dimensional space-time trapezoid is considered, where all the spatial dimensions plus time are represented. Cuts can be performed recursively over the trapezoid in space or time to generate two new, smaller trapezoids ($T1$ and $T2$), where the stencil can be computed in an optimal way due to size constraints of the cache hierarchy. Cutting in space produces a left side trapezoid ($T1$) where there is no point depending on the right side trapezoid ($T2$), thus allowing $T1$ to be fully computed before $T2$. In addition, recursive cuts can be taken over the time dimension. In this cut, the original time region ($t0$, $t1$) is split into a lower $T1$ trapezoid ($t0$, tn) and an upper $T2$ trapezoid (tn, $t1$). As in space cut, $T1$ does not depend on any point of $T2$, and $T1$ can be computed in advance. Cache-oblivious has been also parallelized [37] by creating space cuts (either in the Z or Y dimensions) of inverted and non-inverted trapezoids, which are computed in order to preserve dependencies.

4.2.3 Pipeline Optimizations

Low level optimizations at the CPU pipeline level include several well-known techniques. These techniques may be categorized into three groups: loop transformations, data access and streaming optimizations. Loop unrolling, loop fission and loop fusion are part of the loop transformations group. They can reduce the overall execution time of scientific codes by reducing register pressure and data dependency as well as improving temporal locality [51, 57].

Data access optimizations include techniques such as software prefetching, software pipelining and register blocking [3, 14, 46, 60, 73], all of these relate to improving data access latency within the memory hierarchy (from register to main memory level). Finally, Symmetric Multi-Processing (SMP), Single-Instruction Multiple-Data (SIMD) or Multiple-Instruction Multiple-Data (MIMD) are some programming paradigms which are included in the streaming optimizations group. The multiprocessing term refers to the ability to execute multiple processes, threads and instructions concurrently in a hardware system.

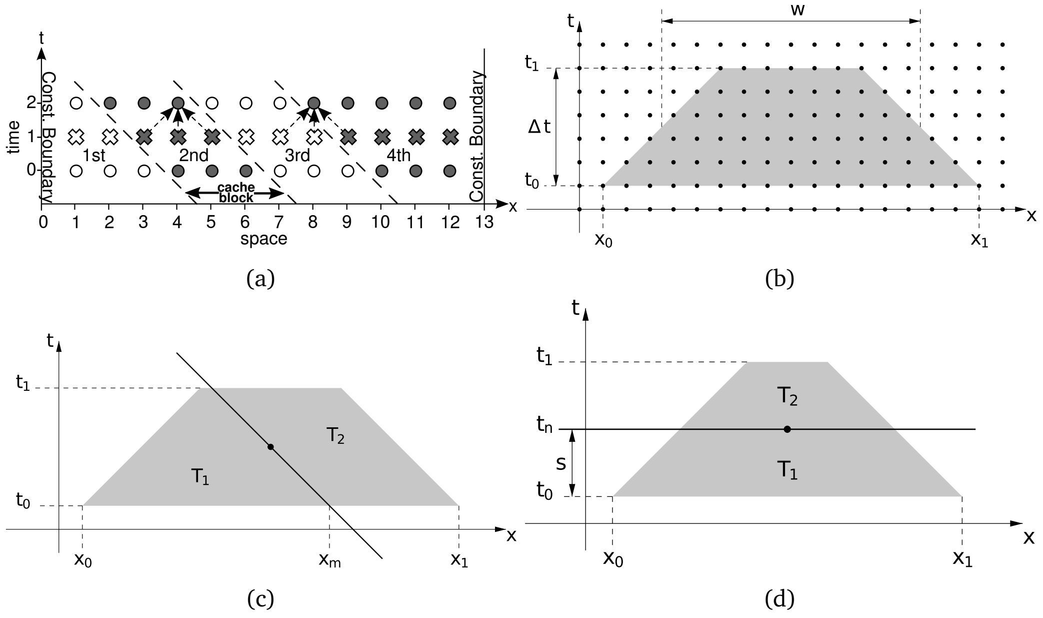


Figure 4.4: Execution sequence of time blocking algorithms on 2D problems. (a): the cache blocks in time-skewing, depicted here with different patterns, must be traversed in a specific order to preserve stencil dependencies. (b): $(n + 1)$ dimensional space-time trapezoid used in cache-oblivious. (c) and (d): space and time cuts where new trapezoids are generated cutting through the center of the trapezoid and $\Delta t/2$ horizontal line respectively.

All the previous techniques have been successfully used in many computational fields to improve the processing throughput and increase the IPC metric (Instructions Per Cycle). Furthermore, the performance of the codes can be improved significantly by combining several of these, although collateral effects (performance issues) can appear from time to time. Some modern compilers are able to generate code automatically that takes advantage of some of these techniques, thus relieving the developer of the tedious implementation work. However, some pipeline optimizations may increase both instruction code size and register pressure. Therefore, they must be used carefully, since overuse can lead to register spilling, which would produce slower code due to additional saves and restores of register code from the stack.

The stencil code representation in terms of arithmetic instructions is another issue to bear in mind. Depending on the hardware, one type of stencil code codification could perform better than others. Table 4.1 shows two different ways of stencil representation: factored and expanded. The former uses add and multiplication instructions, while the latter maps more naturally to fused multiply-add and multiplication instructions.

Method	Stencil code	Operations	Instructions
Factored (Muls and Adds inst. based)	$\mathcal{X}^{t}_{i,j,k} = C_0 * \mathcal{X}^{t-1}_{i,j,k}$ $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k})$ $+ \ldots +$ $+ C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$	$3 * dim * \ell + 1$ Muls: $dim * \ell + 1$ Adds: $2 * dim * \ell$	$3 * dim * \ell + 1$ Muls: $dim * \ell + 1$ Adds: $2 * dim * \ell$
Expanded (Fused Multiply-Add inst. based)	$\mathcal{X}^{t}_{i,j,k} = C_0 * \mathcal{X}^{t-1}_{i,j,k}$ $+ C_{Z1} * \mathcal{X}^{t-1}_{i-1,j,k} + C_{Z1} * \mathcal{X}^{t-1}_{i+1,j,k}$ $+ \ldots +$ $+ C_{Z\ell} * \mathcal{X}^{t-1}_{i-\ell,j,k} + C_{Z\ell} * \mathcal{X}^{t-1}_{i+\ell,j,k}$	$4 * dim * \ell + 1$ Muls: $2 * dim * \ell + 1$ Adds: $2 * dim * \ell$	$2 * dim * \ell + 1$ MAdds: $2 * dim * \ell$ Muls: 1

Table 4.1: Operation and instruction cost of two representations of stencil codes. Depending on the instruction set and the latency (in cycles) of each arithmetic instruction in the CPU pipeline, one specific representation (factored or expanded) may outperform the other.

4.3 The Semi-stencil Algorithm

The Semi-stencil algorithm involves noticeable changes to the structure as well as the memory access pattern of the stencil computation. This new computation structure (depicted in Figure 4.5) consists of two phases: *forward* and *backward* updates, which are described in detail in Section 4.3.1. Additionally, due to this new structure, three parts of code, called *head*, *body* and *tail* need to be developed to preserve the numerical soundness; these are described at the end of this section. To conclude this section, it will be shown how Semi-stencil is orthogonal and may be combined with any other optimization technique.

As shown in Section 4.1, there are two main issues in stencil computations. First, the low floating-point operation to data cache access ratio and, second, the high latency memory access pattern especially for the last two dimensions. Basically, the Semi-stencil algorithm tackles these two issues in the following ways:

- It improves data locality since less data is required on each axis per loop iteration. This may have an important benefit for hierarchical cache architectures, where the non-contiguous axes (X and Y) of a 3D domain are more expensive latency-wise.

- The new memory access pattern reduces the total number of loads per inner loop iteration, while keeping the same number of floating-point operations. Thereby, increasing the data reuse and thus the *FP/Cache* ratio, relative to the classical estimate of Equation 4.1.

As mentioned before, two phases (*forward* and *backward* updates) are needed to carry out the new memory access pattern of the Semi-stencil algorithm. These are the core of the algorithm, and the following subsection elaborates them.

4.3.1 *Forward* and *Backward* Updates

Line 5 of Algorithm 3 shows, in computational terms, the generic spatial operator for FD codes. This line of pseudo-code updates the value of $\mathcal{X}^t_{i,j,k}$ in one step (iteration $\{i, j, k\}$ of the loop). The essential idea of the Semi-stencil is to break up this line into two phases, thereby partially updating more than one point per loop iteration. To carry out this parallel update, the factored *add* and *mul* operations $(c_i * (x_{-i} + x_{+i}))$ must be decomposed into multiply-add instructions $(c_i * x_{-i} + c_i * x_{+i}\ )$ in order to split up the classical computation into *forward* and *backward* updates.

The *forward* update is the first contribution that a $\mathcal{X}^t$ point receives at time-step t. During this phase, when the i iteration of the sweep axis is being computed, the $\mathcal{X}^t_{i+\ell}$ point is updated with ℓ $\mathcal{X}^{t-1}$ rear contributions (as depicted in Figure 4.5.a). Through this operation we obtain a precomputed $\mathcal{X}'^t_{i+\ell}$ value. Recall that ℓ represents the number of neighbors used in each axis direction. Equation 4.2 shows, in mathematical terms, a summary of the *forward* operations performed over $\mathcal{X}'^t_{i+\ell}$ point in a one-dimensional (1D) problem,

$$\mathcal{X}'^t_{i+\ell} = C_1 * \mathcal{X}^{t-1}_{i+\ell-1} + C_2 * \mathcal{X}^{t-1}_{i+\ell-2} + \cdots + C_{\ell-1} * \mathcal{X}^{t-1}_{i+1} + C_\ell * \mathcal{X}^{t-1}_{i} \tag{4.2}$$

where the prime character (′) denotes that the point has been partially computed, and some contributions are still missing. In addition, only ℓ neighbor points of $\mathcal{X}^{t-1}$ have been loaded and one $\mathcal{X}^t$ point, $\mathcal{X}'^t_{i+\ell}$, stored so far.

In a second phase, called *backward*, a previously precomputed $\mathcal{X}'^t_i$ point in the *forward* update is completed by adding in the front axis contributions ($\mathcal{X}^{t-1}_i$ to $\mathcal{X}^{t-1}_{i+\ell}$). More precisely,

$$\begin{aligned} \mathcal{X}^t_i = \mathcal{X}'^t_i &+ C_0 * \mathcal{X}^{t-1}_i + C_1 * \mathcal{X}^{t-1}_{i+1} + C_2 * \mathcal{X}^{t-1}_{i+2} \\ &+ \quad \cdots \quad + C_{\ell-1} * \mathcal{X}^{t-1}_{i+\ell-1} + C_\ell * \mathcal{X}^{t-1}_{i+\ell} \end{aligned} \tag{4.3}$$

(see also Figure 4.5.b).

Only two loads are required in this process to complete the *backward* computation, since most of the $t-1$ time-step points were loaded during the *forward* update of the $X^t_{i+\ell}$ computation and hence they can be reused. The two loads required for this second phase are: the $\mathcal{X}^{t-1}_{i+\ell}$ point and the precomputed $\mathcal{X}'^t_i$ value. This phase also needs one additional store to write the final $\mathcal{X}^t_i$ value.

Finally, in order to carry out both updates, $2 * \ell + 1$ floating-point operations are issued in the inner loop (ℓ Multiply-Add instructions for neighbor contributions and one multiplication for the self-contribution). Moreover, in particular scenarios the computation can be reused; if ℓ is a multiple of two, the $C_{l/2} * \mathcal{X}^{t-1}_{i+l/2}$ operation may be used for *forward* and *backward* updates when performing the $\mathcal{X}^t_i$ and $\mathcal{X}^t_{i+\ell}$ computation. These scenarios may lead to a slight reduction of the inner loop instructions and a further improvement in the execution time.

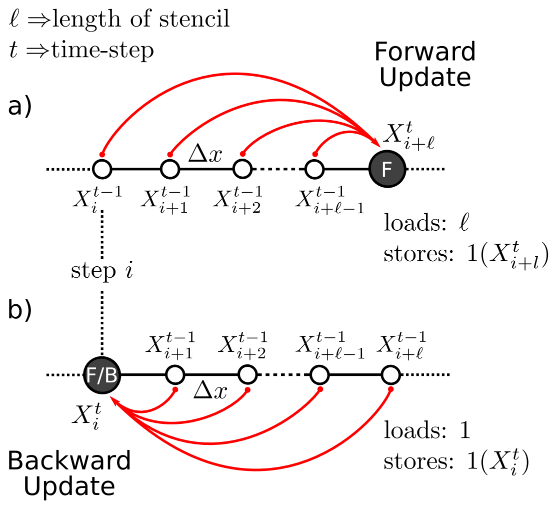


Figure 4.5: Detail of the two phases for the *Semi-stencil* algorithm at iteration i of a 1D domain. a) Forward update on $\mathcal{X}^t_{i+\ell}$ point and b) Backward update on $\mathcal{X}^t_i$ point. Notice that the $\mathcal{X}^{t-1}_{i+1,\cdots,i+\ell-1}$ interval can be reused.

4.3.2 Floating-Point Operations to Data Cache Access Ratio

For the purpose of comparing the classical and the Semi-stencil algorithms, we calculate the *FP/Cache* ratio for the latter algorithm as follows,

$$\begin{aligned} FP/Cache_{Semi} &= \frac{FloatingPoint\ Operations}{Data\ Cache\ Accesses} = \frac{2 * MultiplyAdd\ Instructions}{\mathcal{X}^{t-1}\ Loads + \mathcal{X}^t\ Loads + \mathcal{X}^t\ Stores} \\ &= \frac{2*2*dim*\ell+1}{\underbrace{((dim-1)*\ell+1)}_{\mathcal{X}^{t-1}\ Loads} + \underbrace{(dim-1)}_{\mathcal{X}^t\ Loads} + \underbrace{(dim)}_{\mathcal{X}^t\ Stores}} = \frac{4*dim*\ell+1}{dim*\ell-\ell+2*dim} \end{aligned} \tag{4.4}$$

where the total number of floating-point operations remains constant with respect to the classical stencil implementation (see Equation 4.1). Equation 4.4 also shows that $\mathcal{X}^{t-1}$ loads to the L1 cache have decreased substantially, by almost a factor of 2. Nevertheless, the $\mathcal{X}^t$ stores have increased and a new term ($\mathcal{X}^t$ loads) has appeared in the equation due to the partial computations applied to the X and Y axes. Reducing the number of loads results in less cycles to compute the inner loop, as well as a lower register pressure. The lower register use means the compiler has the opportunity to perform more aggressive low level optimizations, such as: loop unrolling, software pipelining and software prefecthing.

However, as shown in Figure 4.5, the Semi-stencil algorithm updates two points per iteration leading to double the number of $\mathcal{X}^t$ stores. Depending on the architecture, this could result in loss of performance. For instance, in caches with write-allocate policy, a store miss produces a load block action followed by a write-hit, which will produce cache pollution and a pipeline stall. Nowadays, some cache hierarchy architectures implement cache-bypass techniques to address this issue.

Reviewing Equation 4.4, we see that the new computation structure of the Semi-stencil allows the *FP/Cache* ratio to increase to ≈ 5 flops per d-cache access for 3D problems. For ℓ in the range of 4 to 14, the *FP/Cache* ratio is a factor of between 1.3 and 1.7 times better than the classical stencil implementation, which will have a clear effect on performance.

4.3.3 *Head*, *Body* and *Tail* computations

FD methods require what are called interior points (inside the solution domain) and ghost points (outside the solution domain). The new algorithm structure of Semi-stencil takes this feature into account to preserve the numerical soundness.

To obtain consistent results with the two-phase update on border interior points, the algorithm must be split into three different sections: *head*, *body* and *tail*. The *head* section updates the first ℓ interior points with the rear contributions (*forward* phase). In the *body* section, the interior points are updated with back and forth neighbor interior elements (both *forward* and *backward* phases are carried out). Finally, in the *tail* section, the last ℓ interior points of the axis are updated with the front contributions (*backward* phase). Figure 4.6 shows a 1D execution example of Semi-stencil where the three sections are clearly depicted.

4.3.4 Orthogonal Algorithm

As stated in Section 4.2, the state-of-the-art in stencil computations has improved in recent years with the publication of several optimization techniques. Some of these can significantly increase the execution performance under specific circumstances. However, most of them can not be combined due to traversing strategy incompatibilities. In other words, some state-of-the-art techniques share a particular data traversal method when the stencil operator is computed, for example: space blocking or space-time blocking based. Therefore, not all combinations of these techniques are feasible.

In contrast to the previous group of optimization algorithms, the Semi-stencil specifies the manner in which the stencil operator is calculated and how data is accessed in the inner loop (see Algorithm 4). This kind of algorithm, which we term a *structural-based optimization algorithm*, is orthogonal and can easily be implemented without modifying the sweeping order of the computational domain. Thus, the Semi-stencil structure is complementary with traversing optimization algorithms such as Rivera, Time-skewing or Cache-oblivious. At the same time, Semi-stencil can be combined with low level pipeline optimizations such as loop-level transformations, software prefetching or software pipelining.

Note that Semi-stencil can be applied to any axis of an n-dimensional stencil problem, from unit-stride to least-stride dimension. Our recent studies in 3D stencil problems have shown that the Semi-stencil algorithm is most suitable for the least-stride dimensions of 3D problems (X and Y in our Cartesian axes). There are four main reasons for this behavior. First, most modern compilers can take advantage of unit-stride accesses by reusing previous

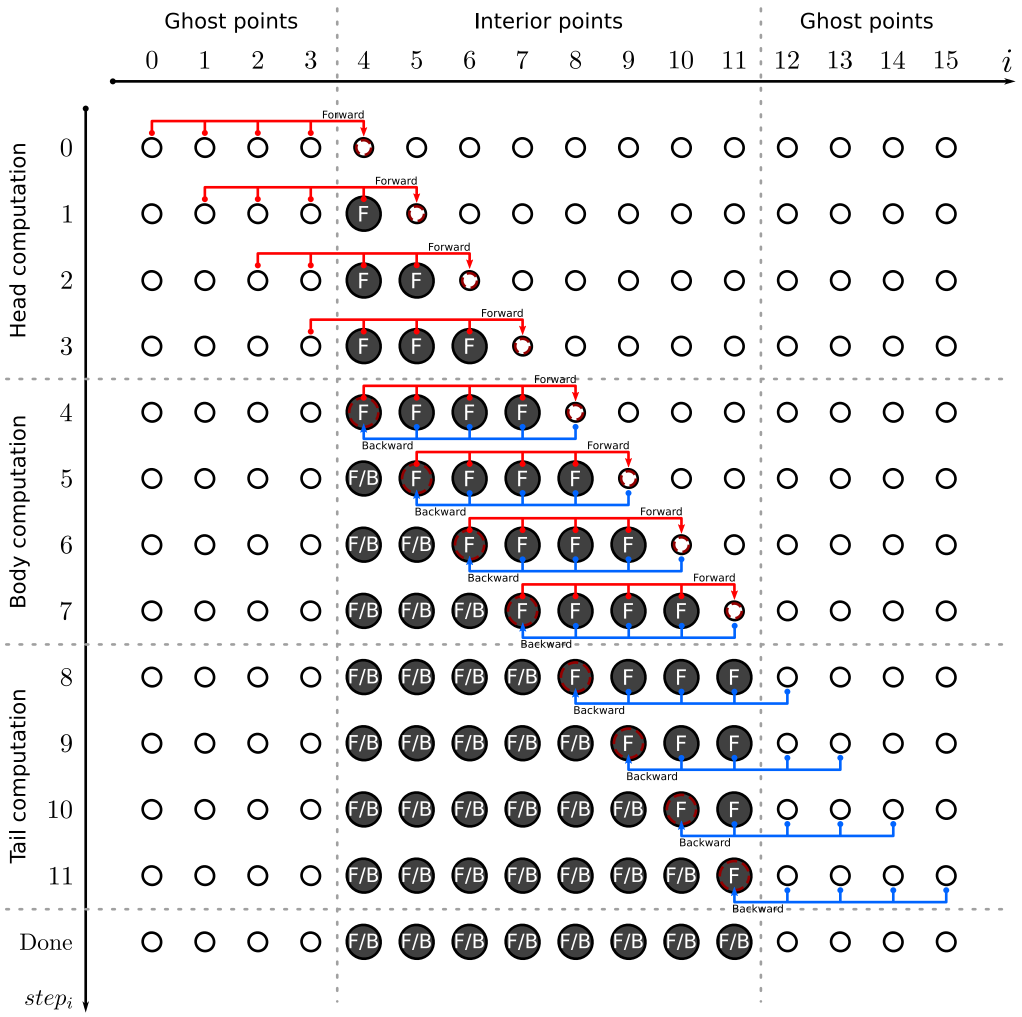


Figure 4.6: Execution example of Semi-stencil algorithm for a 1D problem, where $\ell = 4$. F stands for a *forward* update and B stands for a *backward* update. The horizontal axis represents the space domain ($Z = 16$ including ghost points) of the problem. The vertical axis shows the execution time (example completed in 12 steps) for each algorithm section.

loads and fetching one value per iteration in a steady state. Second, the programmer may easily add some pipeline optimizations that would improve performance for the unit-stride dimension in the same way that Semi-stencil does. Third, current architectures usually sport hardware prefetchers which may especially help to reduce unit-stride latency accesses. And fourth, our memory access model for stencil computations (see the summary in Table 4.2) has shown that a full-axis Semi-stencil algorithm implementation has a slightly higher penalty for $\mathcal{X}^t$ loads and $\mathcal{X}^t$ stores than a partial Semi-stencil implementation.

Algorithms 6, 7 and 8 show the pseudo-code implementation of Rivera, Time-skewing and Cache-oblivious algorithms respectively with a partial Semi-stencil implementation (detailed

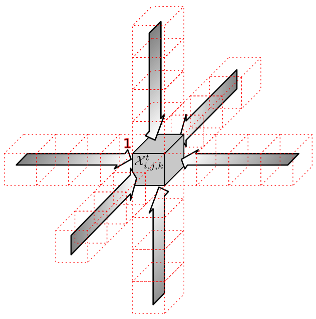
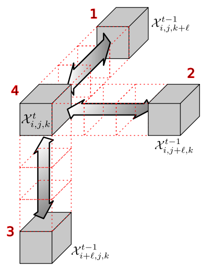
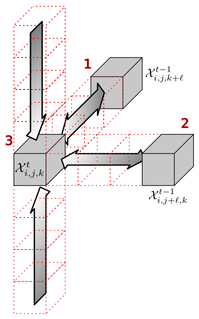
Dataset accesses	Classical Stencil	Full *Semi-stencil* (Z, X and Y axis)	Partial *Semi-stencil* (X and Y axis)
Structure of the stencil		 1 $\mathcal{X}^{t-1}_{i,j,k+\ell}$ 2 4 $\mathcal{X}^t_{i,j,k}$ $\mathcal{X}^{t-1}_{i,j+\ell,k}$ 3 $\mathcal{X}^{t-1}_{i+\ell,j,k}$	 1 $\mathcal{X}^{t-1}_{i,j,k+\ell}$ 2 3 $\mathcal{X}^t_{i,j,k}$ $\mathcal{X}^{t-1}_{i,j+\ell,k}$
$\mathcal{X}^{t-1}$ loads	$2 * (dim - 1) * \ell + 1$	$(dim - 1) * \ell + 1$	$(dim - 1) * \ell + 1$
$\mathcal{X}^t$ loads	0	dim	$dim - 1$
$\mathcal{X}^t$ stores	1	$dim + 1$	dim

Table 4.2: Dataset accesses per stencil implementation in order to compute one point of the domain. The figures show the structure of each stencil while dark boxes and numbers represent the updated points and their computation order respectively. Recall that data reuse for load operation is considered for the unit-stride dimension (Z) through compiler optimizations.

in Algorithms 4 and 5). All these implementations are freely available in our version of the StencilProbe micro-benchmark [25].

Algorithm 4 The Semi-stencil algorithm pseudo-code (Part 1). $\mathcal{X}^t$ is the space domain for time-step t, where Z, X and Y are the dimensions of the datasets including ghost points. ℓ denotes the neighbors used for the central point contribution. $C_{Z1\ldots Z\ell}$, $C_{X1\ldots X\ell}$ and $C_{Y1\ldots Y\ell}$ are the spatial discretization coefficients for each direction and C_0 for the self-contribution. z_s, z_e, x_s, x_e, y_s and y_e denote the area of $\mathcal{X}^t$ where the stencil operator is computed.

```
1:  procedure Semi-stencil($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $\ell$, $z_s$, $z_e$, $x_s$, $x_e$, $y_s$, $y_e$)
2:      for $k = y_s - \ell$ to MIN($y_s, y_e - \ell$) do                      ▷ Head $Y$ / Forward $Y$
3:          for $j = x_s$ to $x_e$ do
4:              for $i = z_s$ to $z_e$ do
5:                  $\mathcal{X}^t_{i,j,k+\ell} = C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{Y\ell-1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+\ell-1}$
6:              end for
7:          end for
8:      end for

9:      for $k = y_s$ to $y_e - \ell$ do                                          ▷ Body $Y$
10:         for $j = x_s - \ell$ to MIN($x_s, x_e - \ell$) do                  ▷ Head $X$ / Forward $X$
11:             for $i = z_s$ to $z_e$ do
12:                 $\mathcal{X}^t_{i,j+\ell,k} \overset{\pm}{=} C_{X\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{X\ell-1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j+\ell-1,k}$
13:             end for
14:         end for
15:         for $j = x_s$ to $x_e - \ell$ do                                      ▷ Body $X$
16:             for $i = z_s$ to $z_e$ do                              ▷ Backward $X$,$Y$ / Forward $X$,$Y$
17:                 $\mathcal{X}^t_{i,j,k} \overset{\pm}{=} C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                        $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                        $+ C_{X1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{X\ell} * \mathcal{X}^{t-1}_{i,j+\ell,k}$
                        $+ C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k+\ell}$

18:                 $\mathcal{X}^t_{i,j+\ell,k} \overset{\pm}{=} C_{X\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{X\ell-1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j+\ell-1,k}$

19:                 $\mathcal{X}^t_{i,j,k+\ell} = C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{Y\ell-1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+\ell-1}$
20:             end for
21:         end for
22:         for $j$ =MAX($x_e - \ell, x_s$) to $x_e$ do                              ▷ Tail $X$
23:             for $i = z_s$ to $z_e$ do                                ▷ Backward $X$,$Y$ / Forward $Y$
24:                 $\mathcal{X}^t_{i,j,k} \overset{\pm}{=} C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                        $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                        $+ C_{X1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{X\ell} * \mathcal{X}^{t-1}_{i,j+\ell,k}$
                        $+ C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k+\ell}$

25:                 $\mathcal{X}^t_{i,j,k+\ell} = C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{Y\ell-1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+\ell-1}$
26:             end for
27:         end for
28:     end for                                                          ▷ Continues in part 2
```

Algorithm 5 Continuation of the Semi-stencil algorithm pseudo-code (Part 2). This algorithm must be called as follows to compute the entire data set: $\mathrm{Semi\text{-}stencil}(\mathcal{X}^t, \mathcal{X}^{t-1}, \ell, \ell, Z - \ell, \ell, X - \ell, \ell, Y - \ell)$. Note that, in this work, the coefficients are considered constant.

```
29:     for $k$ =MAX$(y_e - \ell, y_s)$ to $y_e$ do                                         ▷ Tail $Y$
30:         for $j = x_s - \ell$ to MIN$(x_s, x_e - \ell)$ do                          ▷ Head $X$ / Forward $X$
31:             for $i = z_s$ to $z_e$ do
32:                 $\mathcal{X}^t_{i,j+\ell,k} \overset{\pm}{=} C_{X\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{X\ell-1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j+\ell-1,k}$
33:             end for
34:         end for
35:         for $j = x_s$ to $x_e - \ell$ do                                                ▷ Body $X$
36:             for $i = z_s$ to $z_e$ do                                   ▷ Backward $X$,$Y$ / Forward $X$
37:                 $\mathcal{X}^t_{i,j,k} \overset{\pm}{=} C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                        $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                        $+ C_{X1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{X\ell} * \mathcal{X}^{t-1}_{i,j+\ell,k}$
                        $+ C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k+\ell}$
38:                 $\mathcal{X}^t_{i,j+\ell,k} \overset{\pm}{=} C_{X\ell} * \mathcal{X}^{t-1}_{i,j,k} + C_{X\ell-1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{Y1} * \mathcal{X}^{t-1}_{i,j+\ell-1,k}$
39:             end for
40:         end for
41:         for $j$ =MAX$(x_e - \ell, x_s)$ to $x_e$ do                                         ▷ Tail $X$
42:             for $i = z_s$ to $z_e$ do                                                ▷ Backward $X$,$Y$
43:                 $\mathcal{X}^t_{i,j,k} \overset{\pm}{=} C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                        $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                        $+ C_{X1} * \mathcal{X}^{t-1}_{i,j+1,k} + \ldots + C_{X\ell} * \mathcal{X}^{t-1}_{i,j+\ell,k}$
                        $+ C_{Y1} * \mathcal{X}^{t-1}_{i,j,k+1} + \ldots + C_{Y\ell} * \mathcal{X}^{t-1}_{i,j,k+\ell}$
44:             end for
45:         end for
46:     end for
47: end procedure
```

Algorithm 6 Pseudo-code for the space tiling implementation of the Semi-stencil algorithm. TI and TJ define the $TI \times TJ$ block size. The less-stride dimension (Y) is left uncut.

```
1: procedure Rivera($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $Z$, $X$, $Y$, $TI$, $TJ$, $timesteps$, $\ell$)
2:     for $t = 0$ to $timesteps$ do                               ▷ Compute blocks of size $TI \times TJ$
3:         for $jj = \ell$ to $X, TJ$ do
4:             for $ii = \ell$ to $Z, TI$ do
5:                 Semi-stencil($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $\ell$, $ii$, MIN($ii + TI$,$Z - \ell$),
                                              $jj$, MIN($jj + TJ$,$X - \ell$), $\ell$, $Y - \ell$)
6:             end for
7:         end for
8:     end for
9: end procedure
```

Algorithm 7 Pseudo-code for the time-skewing implementation of the Semi-stencil algorithm.

```
procedure Time-skewing($\mathcal{X}^t$, $\mathcal{X}^{t-1}$, $Z$, $X$, $Y$, $TI$, $TJ$, $TK$, $timesteps$, $\ell$)
    for $kk = \ell$ to $Y - \ell, TK$ do
        $negYslope = \ell$
        $posYslope = -\ell$
        $tyy =$MIN($TK$,$Y - kk - \ell$)
        if $kk = \ell$ then
            $negYslope = 0$
        end if
        if $kk = Y - tyy - \ell$ then
            $posYslope = 0$
        end if
        for $jj = \ell$ to $X - \ell, TJ$ do
            $negXslope = \ell$
            $posXslope = -\ell$
            $txx =$MIN($TJ$,$X - jj - \ell$)
            if $jj = \ell$ then
                $negXslope = 0$
            end if
            if $jj = X - txx - \ell$ then
                $posXslope = 0$
            end if
            for $ii = \ell$ to $Z - \ell, TI$ do
                $negZslope = \ell$
                $posZslope = -\ell$
                $tzz =$MIN($TI$,$Z - ii - \ell$)
                if $ii = \ell$ then
                    $negZslope = 0$
                end if
                if $ii = Z - tzz - \ell$ then
                    $posZslope = 0$
                end if

                $p\mathcal{X}^t = \mathcal{X}^t$
                $p\mathcal{X}^{t-1} = \mathcal{X}^{t-1}$
                for $t = t0$ to $timesteps$ do        ▷ Compute stencil on 3D trapezoid
                    Semi-stencil($p\mathcal{X}^t$, $p\mathcal{X}^{t-1}$, $\ell$,
                            MAX($\ell$, $ii - t * negZslope$), MAX($\ell$, $ii + tzz * posZslope$),
                            MAX($\ell$, $jj - t * negXslope$), MAX($\ell$, $jj + txx * posXslope$),
                            MAX($\ell$, $kk - t * negYslope$), MAX($\ell$, $kk + tyy * posYslope$))
                    $tmpPtr = p\mathcal{X}^t$
                    $p\mathcal{X}^t = p\mathcal{X}^{t-1}$
                    $p\mathcal{X}^{t-1} = tmpPtr$
                end for
            end for
        end for
    end for
end procedure
```

Algorithm 8 Pseudo-code for the cache-oblivious implementation of the Semi-stencil algorithm. The trapezoid to compute is defined by $\tau(t0, t1, z0, dz0, z1, dz1, x0, dx0, x1, dx1, y0, dy0, y1, dy1)$.

```
1: procedure Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t1$,
                              $z0$, $dz0$, $z1$, $dz1$, $x0$, $dx0$, $x1$, $dx1$, $y0$, $dy0$, $y1$, $dy1$)
2:     $dt = t1 - t0$
3:     $wz = ((z1 - z0) + (dz1 - dz0) * dt * 0.5)$                    ▷ Compute 3D trapezoid volume
4:     $wx = ((x1 - x0) + (dx1 - dx0) * dt * 0.5)$
5:     $wy = ((y1 - y0) + (dy1 - dy0) * dt * 0.5)$
6:     if $dt = 1$ or $vol <$ CUTOFF then
7:         for $t = t0$ to $t1$ do                       ▷ Base case: compute stencil on 3D trapezoid
8:             Semi-stencil($X[\text{MOD}(t+1,2)]$, $X[\text{MOD}(t,2)]$, $ds$,
                        $z0 + (t - t0) * dz0$, $z1 + (t - t0) * dz1$,
                        $x0 + (t - t0) * dx0$, $x1 + (t - t0) * dx1$,
                        $y0 + (t - t0) * dy0$, $y1 + (t - t0) * dy1$)
9:         end for
10:    else if $dt > 1$ then
11:        if $wy \geq 2 * ds * dt$ then                                        ▷ Space Y-cut
12:            $ym = (2 * (y0 + y1) + (2 * ds + dy0 + dy1) * dt)/4$
13:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t1$,
                              $z0$, $dz0$, $z1$, $dz1$, $x0$, $dx0$, $x1$, $dx1$, $y0$, $dy0$, $ym$, $-ds$)
14:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t1$,
                              $z0$, $dz0$, $z1$, $dz1$, $x0$, $dx0$, $x1$, $dx1$, $ym$, $-ds$, $y1$, $dy1$)
15:        else if $wx \geq 2 * ds * dt$ then                                   ▷ Space X-cut
16:            $xm = (2 * (x0 + x1) + (2 * ds + dx0 + dx1) * dt)/4$
17:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t1$,
                              $z0$, $dz0$, $z1$, $dz1$, $x0$, $dx0$, $xm$, $-ds$, $y0$, $dy0$, $y1$, $dy1$)
18:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t1$,
                              $z0$, $dz0$, $z1$, $dz1$, $xm$, $-ds$, $x1$, $dx1$, $y0$, $dy0$, $y1$, $dy1$)
19:        else                                                                  ▷ Time cut
20:            $s = dt/2$
21:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0$, $t0 + s$,
                              $z0$, $dz0$, $z1$, $dz1$, $x0$, $dx0$, $x1$, $dx1$, $y0$, $dy0$, $y1$, $dy1$)
22:            Cache-oblivious($X[t,t-1]$, $Z$, $X$, $Y$, CUTOFF, $ds$, $t0 + s$, $t1$,
                              $z0 + dz0 * s$, $dz0$, $z1 + dz1 * s$, $dz1$,
                              $x0 + dx0 * s$, $dx0$, $x1 + dx1 * s$, $dx1$,
                              $y0 + dy0 * s$, $dy0$, $y1 + dy1 * s$, $dy1$)
23:        end if
24:    end if
25: end procedure
```

4.4 Experiments

In this section, the experimental results of the Semi-stencil algorithm are presented. All these experimients have been conducted using our modified version of the StencilProbe micro-benchmark (see Section 3.8), which has been updated including the Semi-stencil algorithm. It is worth noting that, with the new benchmark features, the parameter space to be explored becomes quite large. In order to bound the search area for the experiments, we set some parameters as either constants or within a reasonable range of values (e.g., problem size and blocking parameters). Table 4.3 shows a summary of all the parameters used in the micro-benchmark.

Parameters	Range of values
Problem sizes	$512 \times 512 \times 512$ (in double-precision floating-point format)
Stencil sizes (ℓ)	1, 2, 4, 7 and 14 (7, 13, 25, 43 and 85-point respectively)
Time-steps	1, 2, 4 and 8
Algorithms	{Naive, Rivera, Time-skew, Cache-oblivious} $\times$ {Classical, *Semi-stencil*}
Block sizes	$\{16, 32, 64, 128, 256, 512\} \times \{16, 32, 64, 128, 256, 512\} \times 512$
CUTOFF	$\{256, 512, 1k, 2k, 4k, 8k, 16k, 32k\}$
Inner loop opts.	{loop fusion (1 loop), loop fission (2 loops), loop fission (3 loops)}

Table 4.3: List of parameters used for the extended version of StencilProbe. Some combinations may depend on the architecture and the algorithm; for example, block sizes are only used for Rivera and Time-skew, whereas CUTOFF is only used for the Cache-oblivious algorithm. Due to memory constraints on some platforms, the problem size may be reduced to 256^3.

In the following subsections, first, we prove through the *FP/Cache* metric, that our experimental model is consistent with classical and Semi-stencil codes' results. Second, the Roofline model (see Section 3.7) is used to demonstrate through Operational Intensity metric how the Semi-stencil algorithm behaves. Finally, the performance results are shown and evaluated for each testbed platform, including also interesting scalability results for multi and many-core archhitectures.

4.4.1 Data Cache Accesses

As Equations 4.1 and 4.4 convey, the proposed algorithm performs increasingly well in *FP/Cache* and data access terms with respect to the classical approach. Figure 4.7 (top) plots the three models: the classical and the two Semi-stencil (partial-axis X-Y and full-axis Z-X-Y). We see that the classical and the Semi-stencil models tend to a factor of $\approx$ 3 and 5 flops per d-cache access respectively.

To evaluate the robustness of our results, we proceed to compare the model with real performance data. Unfortunately, if the measured data and the projected *FP/Cache* model are compared, we note a slight difference. The main reason for this is due to interference

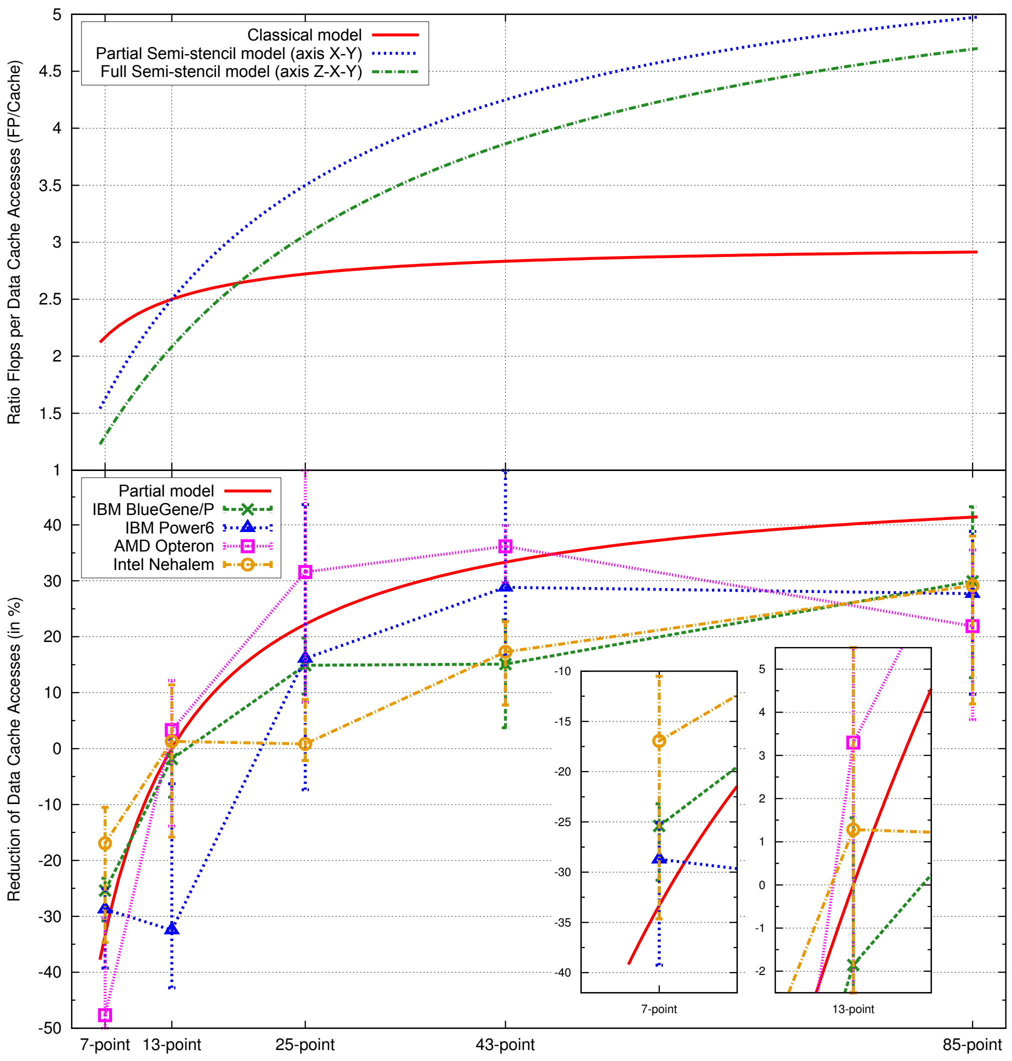


Figure 4.7: Top: Ratio of floating-point operations to data cache accesses for all three stencil models. The horizontal axis represents the number of neighbors used to compute the stencil. The larger the stencil, the better ratio obtained. Bottom: Comparison of data cache access reduction between partial-axis Semi-stencil and classical versions for model and measured data on all four platforms.

from other loads and stores not related to $\mathcal{X}^t$ and $\mathcal{X}^{t-1}$ datasets; coefficients terms, loop control variables and register spilling are all involved in this cache traffic increase. The larger the stencil length (ℓ), the worse the traffic becomes.

In order to achieve an accurate comparison, a new metric must be defined. Taking into account that this issue affects any stencil computation and the artifact should remain constant for a specific stencil size (ℓ); it would be a good idea to compare data cache accesses between different implementations of the same stencil length. Therefore, results are compared based on the accesses in terms of gain or loss using a reduction factor between *FP/Cache* metrics. This new metric is defined as: $AccessesGain = (Accesses_{Classical} - Accesses_{Semi})/Accesses_{Classical}$, and is measured as a percentage. A negative value signifies higher data cache traffic, while a positive percentage signals a lower traffic, and, hence, better performance.

Figure 4.7 (bottom) shows the comparison of data cache accesses for best performance cases using Semi-stencil and classical implementations. Examining the figure, we note that both data groups, the model and the real performance data, are quite close and the experiments follow closely the reduction model curve. In addition, the Semi-stencil algorithm performs poorly for low order stencils ($\ell = \{1, 2\}$), where the negative reduction factor represents an increase in traffic. This result is also in line with the theoretical *FP/Cache* ratio models presented at Figure 4.7 (top), where both Semi-stencil implementations perform worse than the classical code for $\ell \leq 2$. Nevertheless, as expected, the Semi-stencil algorithm behaves better for medium-high order stencils due to a decrease in data cache traffic.

4.4.2 Operational Intensity

In order to demonstrate the robustness of the Semi-stencil algorithm, we conducted the Roofline model on the Intel Nehalem and AMD Opteron architectures where memory traffic counters (DRAM bytes accessed) were gathered using *likwidperfctr*. We can then use the Roofline model to assess how far our empirical results are from the attainable peak performance.

To build the roof part of the model, the DRAM bandwidth and the theoretical peak performance were measured. The DRAM bandwidth measurements were conducted using our modified version of STREAM2 [24] to estimate stream and non-stream bandwidths. The stream bandwidth determines the upper bound whereas the non-stream bandwidth sets the lower bound. These bandwidths determine how far to the left or right the ridge point is on the X-axis. Next, to obtain the Y-axis ceilings, theoretical peak performance, the processor frequency and the floating-point units on a single core were considered. The AMD Opteron and Intel Nehalem cores have two floating-point units, one multiplication and one add unit. Hence, applications must be multiplication/add instruction balanced in order to reach the maximum peak performance. Table 4.4 shows the gathered ceiling data on both architectures.

To gain a better understanding of the Roofline model, the theoretical *Operational Intensity* (OI) for each stencil algorithm and size were also estimated. As described in Section 3.4, three OI groups are devised depending on the stencil 3C's misses: only compulsory, compulsory + capacity and compulsory + capacity + conflict misses. Compulsory misses set the upper bound while the compulsory + capacity + conflict misses set the lower bound in the X-axis of the Roofline model. Table 4.5 shows the estimated OIs using write-back and write-through policies.

Roofline ceilings	Intel Nehalem	AMD Opteron
Peak Stream DRAM (GBytes/s)	8.2	4.6
Peak Non-stream DRAM (GBytes/s)	3.4	1.4
Peak 2 FP units (GFlops/s)	5.86	4.6
Peak 1 FP unit (GFlops/s)	2.93	2.3

Table 4.4: Computational and bandwidth ceilings for Roofline model. Notice that only one core is being considered.

	Classical			*Semi-stencil*		
Stencil sizes	Compulsory	Compulsory + Capacity	Compulsory + Capacity + Conflict	Compulsory	Compulsory + Capacity	Compulsory + Capacity + Conflict
7-point	0.54 (0.81)	0.32 (0.40)	0.23 (0.27)	0.23 (0.27)	0.20 (0.23)	0.18 (0.20)
13-point	1.04 (1.56)	0.44 (0.52)	0.28 (0.31)	0.45 (0.52)	0.35 (0.39)	0.28 (0.31)
25-point	2.04 (3.06)	0.55 (0.61)	0.32 (0.34)	0.88 (1.02)	0.55 (0.61)	0.41 (0.44)
43-point	3.50 (5.31)	0.62 (0.66)	0.34 (0.35)	1.51 (1.77)	0.76 (0.82)	0.50 (0.53)
85-point	7.04 (10.56)	0.68 (0.70)	0.36 (0.37)	3.01 (3.52)	1.00 (1.05)	0.60 (0.62)

Table 4.5: Theoretical OI for stencils depending on their 3C's misses. The gray section depicts where *Semi-stencil* obtains a better ratio compared to the classical. Values in parenthesis are obtained using write-through policy.

Furthermore, we calculated the actual computational peak performance to provide a realistic bound on the Roofline ceilings. Such information was obtained by running the StencilProbe micro-benchmark several times using a small dataset as input (32^3), but without clearing the cache (warm cache effect). Then, the fastest execution time was selected as the reference for the stencil algorithm and size. Using this technique, the optimal computational performance was obtained disregarding DRAM accesses and considering only traffic between processor and cache.

Figure 4.8 combines all the gathered data in the Roofline model for the Intel Nehalem and AMD Opteron architectures to predict the attainable performance. Graphs are on log2-log2 scale, where the Y-axis is the attainable GFlops per second and the X-axis is Flops per DRAM byte accessed. Memory bandwidths (stream and non-stream) and computational peaks (mul/add instruction balanced and imbalanced) determine the optimization regions within the graphs. Remember that the memory measurement is the steady state bandwidth potential of the memory in only one core, and it is not the pin bandwidth of the DRAM chips. The model also suggests, through three regions, which optimizations would be appropriate. The darkest trapezoid suggests working mainly on computational optimizations, the lightest parallelogram proposes trying just memory optimizations, and the region in the corner suggests both types of the above optimizations. The vertical lines on the graphs show bounds for 3C's regions obtained for a 25-point stencil computation, and hence their theoretical OI limits. Additionally, the remaining horizontal lines depict the actual peak performance for each stencil size. Finally, the crosses and circles on the graphs mark the performance achieved by

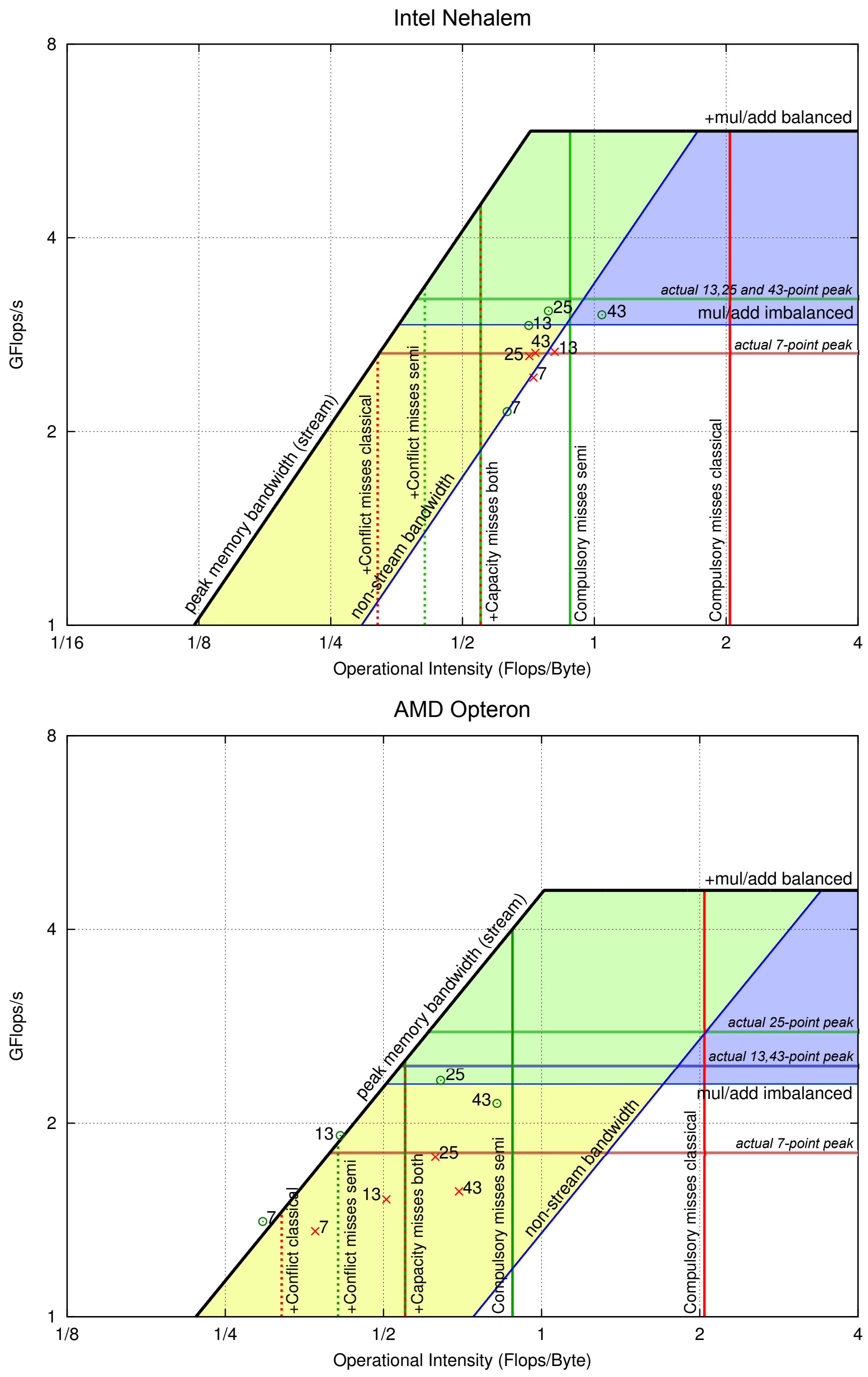


Figure 4.8: Roofline models for Intel Nehalem and AMD Opteron. Computational and bandwidth ceilings bound the theoretical limits for the architecture, whereas the vertical 3C's lines and the horizontal actual peak lines set the limits for each stencil kernel.

Naive and Naive+*Semi-stencil* runs respectively. Numbers shown next to the marks represent their stencil size.

The peak double precision performance for the Intel Nehalem is the highest of the two architectures. However, Figure 4.8 (top) shows that this performance can only be achieved with an OI greater than 0.71 in the best scenario (stream bandwidth) and 1.72 Flops per byte in the worst scenario (non-stream bandwidth). Reviewing the achieved performances, we can note that they are all in the bottom part of the memory bandwidth (45° non-stream line). This behavior may be a consequence of a front side bus limitation and the ineffective work of the snoop filter, which carries coherency traffic and may consume half of the bus bandwidth. Despite the memory bandwidth limitation, the 25 and 43-point stencil computations of Naive+*Semi-stencil* get close to the actual computational peak thanks to their higher OI.

Figure 4.8 (bottom) shows the Roofline model for the AMD Opteron architecture. In this architecture the ridge point of the model is at an OI of 1 Flop per byte for stream bandwidth and at 3.28 Flops per byte for non-stream bandwidth. The model clearly shows that the 7 and 13-point results of Naive+*Semi-stencil* are already limited by peak memory and their performance could only be improved by increasing OI. On the other hand, Naive results fall into the lightest region, indicating not so efficient memory access. Notice that if a vertical line is projected from each Naive result, these lines meet the actual stencil peak performance approximately at the point that diagonal roof part is reached. This behavior confirms the accuracy of the gathered data for the Roofline model. In addition, large stencil size computations (25 and 43-point) of Semi-stencil lead the GFlops/second results despite only showing slightly better OI ratios with respect to their Naive competitors. This metric gap is because OI depends on traffic between the caches and memory, whereas the Semi-stencil algorithm mostly reduces traffic between the processor and caches (see Equation 4.4).

4.4.3 Performance Evaluation and Analysis

This research not only claims that the Semi-stencil approach improves the stencil computation performance, but also that it can be combined with others stencil optimization techniques to achieve even higher performance. Due to the sheer number of possible experiments (see Table 4.3) the most relevant results across all testbed platforms are highlighted and analyzed in this section. All computations have been carried out in double precision, since this is used by most scientific codes.

In order to support our claims, the results of our experiments are shown combined in four arrangements. First, the execution times of classical and Semi-stencil versions are compared when the stencil size (ℓ) and the problem size are changed. Second, hardware counter profilings are given and discussed for each algorithm. Third, a general view of algorithm speedups is outlined, and finally, performance results are grouped by varying blocking parameters ($TI \times TJ \times TK$).

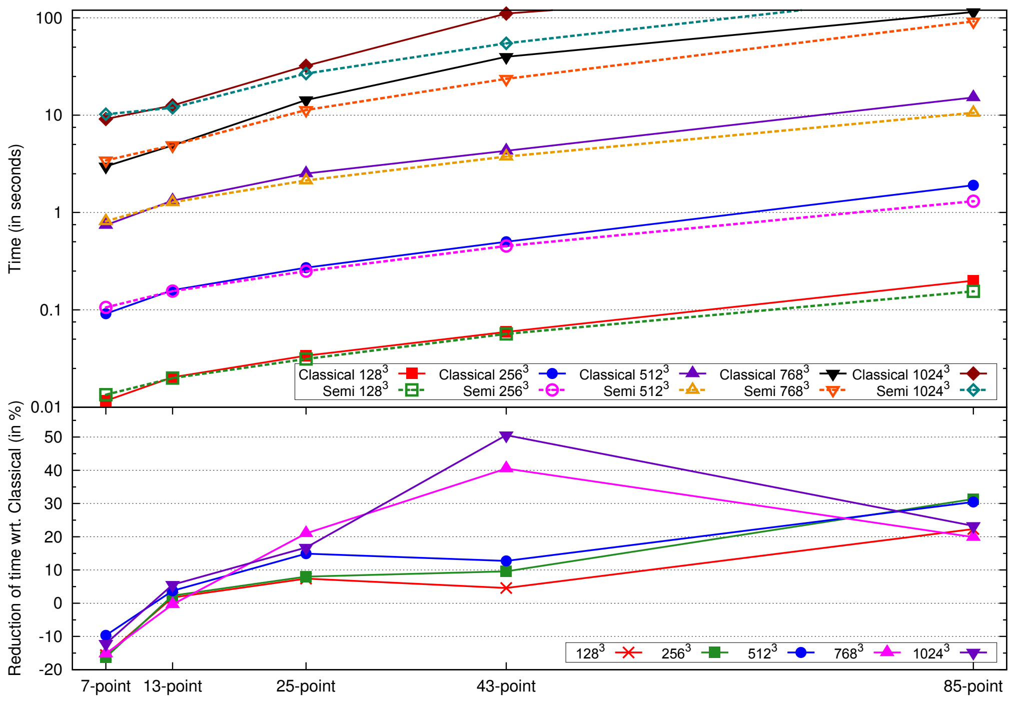


Figure 4.9: Comparison between classical and Semi-stencil runs using different stencil lengths and problem sizes on Intel Nehalem. Top: elapsed time comparison in seconds. Lower is better. Bottom: reduction of time in percentage. A negative value represents a Semi-stencil slower than the classical approach, whereas a positive value depicts a faster one. Notice how important the problem size is in order to obtain further improvement in low-medium order stencils.

Figure 4.9 (top) shows how efficient the Semi-stencil algorithm performs as ℓ and the problem size (N^3) are varied. Here, all parameters have been fixed, except for the stencil length. Each pair of classical and Semi-stencil runs was conducted for a particular problem size, ranging from unusually small sizes (128^3 and 256^3) to realistic cases (512^3, 768^3 and 1024^3). In order to obtain a fair and clear comparison, the reduction in time with respect to the classical algorithm is presented in Figure 4.9 (bottom).

As expected, Semi-stencil versions perform worse for low order stencil computations ($\ell \leq 2$) where the elapsed time is slightly higher compared to the classical implementation. However, for medium and high order stencils ($\ell \geq 4$), our proposed algorithm performs extremely well. Nevertheless, as the problem size grows, the elapsed time difference becomes negligible when $\ell = 2$.

Considering precision and performance requirements, the stencil computation depends on the number of neighbor points. A large number of neighbors will provide high order results at increased computational cost. Given the previous statement and the results shown in Figure 4.9, a Semi-stencil algorithm is a valid option when high precision is required in large scientific problems.

Figures 4.10, 4.11, 4.12 and 4.13 show a detailed collection of hardware counters obtained by profiling on all four platforms. Performance statistics shown are: Memory traffic (GBytes), Million Updates Per Second (MUPS), L1/L2/L3 cache misses, GFlops and *FP/Cache* ratios. Each matrix column represents an algorithm implementation and each matrix row a specific stencil length, ℓ. Generally, most of these show a better performance when $\ell \geq 2$ (13-point). In terms of the *FP/Cache* ratio and cache misses, our proposed algorithm performs increasingly well with respect to the classical algorithm.

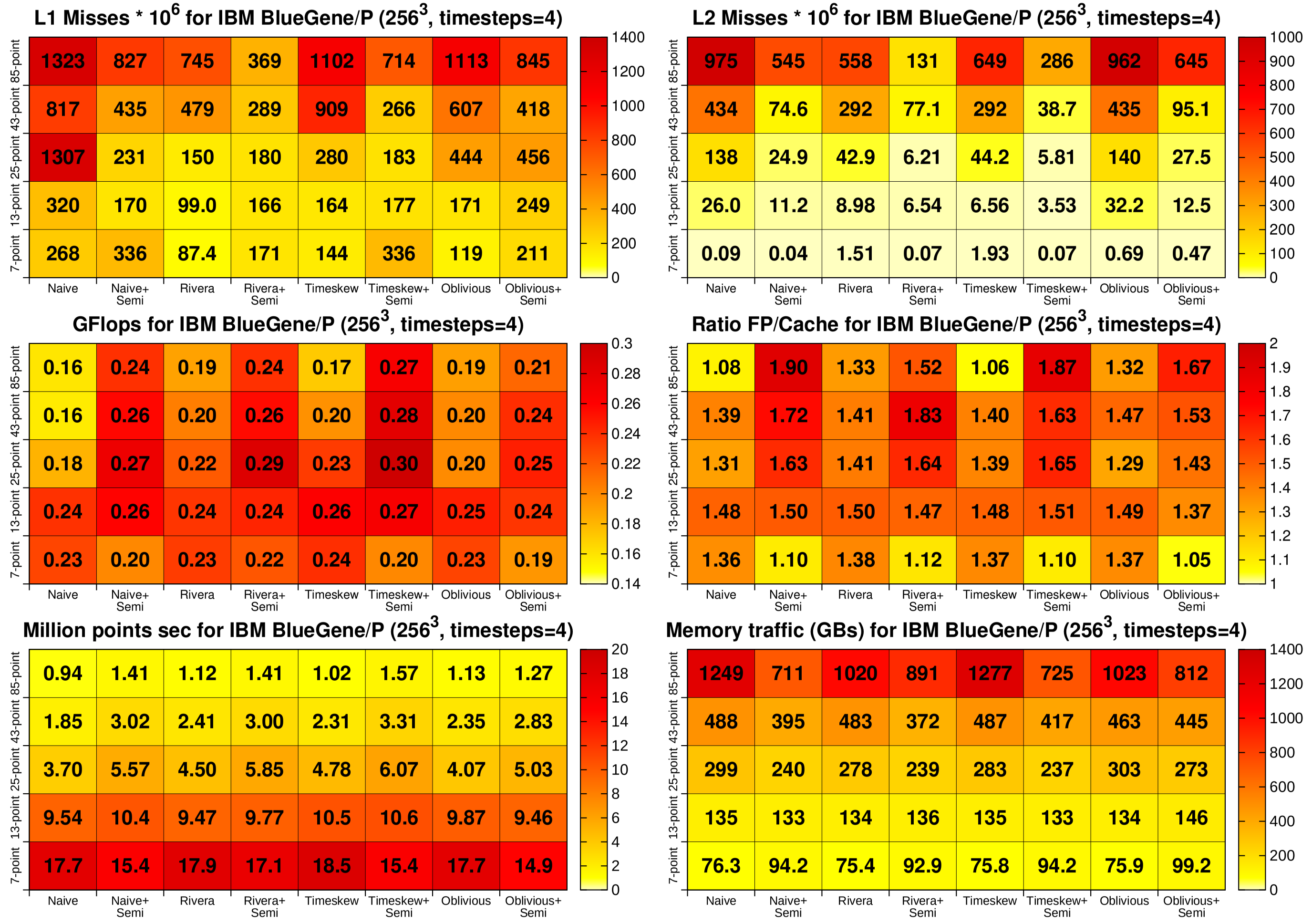
The stacked bar graphs in Figure 4.14 show the speed-up of each implementation and execution times in seconds. Data is gathered by platform, time-steps and stencil length. In these graphs, the baseline for each platform and set of parameters represents the Naive code using a classical algorithm (where the speed-up is 1). We see that, space and time-blocking techniques enhance some of the performance results, ranging from 1.05 to 1.3$\times$. However, using the same enhancements with Semi-stencil improves the performance even further, bestowing aggregated speed-ups of up to 1.6$\times$ in some cases. The BlueGene/P outperforms the others platforms in terms of speed-up especially for the Semi-stencil algorithm.

In order to show how important a blocking parameter is for space and time-blocking algorithms, Figure 4.15 and 4.16 collect all the execution runs sorted by blocking sizes. Results are shown for each platform by modifying the blocking parameters in the horizontal axis. As expected, the Rivera and Time-skewing algorithms show a variation of performance when TI, TJ and TK are changed. In all cases, the best blocking performance is obtained when TI is left uncut [43, 72]; the exception, strangely, is BG/P. This may be due to collateral cache-line conflicts. The Naive and Cache-Oblivious algorithms are plotted using their best configurations (tuning internal loop optimizations and CUTOFF parameter).

Algorithm	Intel Nehalem	IBM POWER6	IBM BlueGene/P	AMD Opteron
Naive	5.01 (1.00)	8.64 (1.00)	9.07 (1.00)	7.42 (1.00)
Rivera	4.82 (1.04)	8.74 (0.99)	7.46 (1.22)	8.63 (0.86)
Timeskew	4.43 (1.13)	8.55 (1.01)	7.02 (1.29)	11.04 (0.67)
Oblivous	**3.89 (1.29)**	9.05 (0.95)	8.25 (1.10)	5.95 (1.25)
Naive + *Semi*	4.27 (1.17)	6.53 (1.32)	6.03 (1.50)	5.64 (1.32)
Rivera + *Semi*	4.25 (1.18/*1.13*)	6.92 (1.25/*1.26*)	5.73 (1.58/*1.30*)	**4.90 (1.51/*1.76*)**
Timeskew + *Semi*	4.25 (1.18/*1.05*)	**6.50 (1.33/*1.32*)**	**5.53 (1.64/*1.27*)**	5.42 (1.37/*2.04*)
Oblivious + *Semi*	3.93 (1.27/*0.99*)	7.04 (1.23/*1.29*)	6.68 (1.36/*1.24*)	5.65 (1.31/*1.05*)

Table 4.6: Summary of execution times (in seconds) and speed-ups for Semi-stencil. Problem sizes are 512^3 except for BlueGene/P where 256^3 has been used due to memory limitations. A widely used stencil size ($\ell = 4$, 25-point) and a time-step of 2 have been chosen. Speed-ups with respect to the Naive implementation and without the Semi-stencil strategy are shown in parenthesis and italics respectively.

A summary of the performance results can be found in Table 4.6. This table shows the execution times and their speed-ups using a stencil size of 4. For three of the four platforms,



L1 Misses * 10^6 for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	1323	827	745	369	1102	714	1113	845
43-point	817	435	479	289	909	266	607	418
25-point	1307	231	150	180	280	183	444	456
13-point	320	170	99.0	166	164	177	171	249
7-point	268	336	87.4	171	144	336	119	211

L2 Misses * 10^6 for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	975	545	558	131	649	286	962	645
43-point	434	74.6	292	77.1	292	38.7	435	95.1
25-point	138	24.9	42.9	6.21	44.2	5.81	140	27.5
13-point	26.0	11.2	8.98	6.54	6.56	3.53	32.2	12.5
7-point	0.09	0.04	1.51	0.07	1.93	0.07	0.69	0.47

GFlops for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	0.16	0.24	0.19	0.24	0.17	0.27	0.19	0.21
43-point	0.16	0.26	0.20	0.26	0.20	0.28	0.20	0.24
25-point	0.18	0.27	0.22	0.29	0.23	0.30	0.20	0.25
13-point	0.24	0.26	0.24	0.24	0.26	0.27	0.25	0.24
7-point	0.23	0.20	0.23	0.22	0.24	0.20	0.23	0.19

Ratio FP/Cache for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	1.08	1.90	1.33	1.52	1.06	1.87	1.32	1.67
43-point	1.39	1.72	1.41	1.83	1.40	1.63	1.47	1.53
25-point	1.31	1.63	1.41	1.64	1.39	1.65	1.29	1.43
13-point	1.48	1.50	1.50	1.47	1.48	1.51	1.49	1.37
7-point	1.36	1.10	1.38	1.12	1.37	1.10	1.37	1.05

Million points sec for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	0.94	1.41	1.12	1.41	1.02	1.57	1.13	1.27
43-point	1.85	3.02	2.41	3.00	2.31	3.31	2.35	2.83
25-point	3.70	5.57	4.50	5.85	4.78	6.07	4.07	5.03
13-point	9.54	10.4	9.47	9.77	10.5	10.6	9.87	9.46
7-point	17.7	15.4	17.9	17.1	18.5	15.4	17.7	14.9

Memory traffic (GBs) for IBM BlueGene/P (256^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	1249	711	1020	891	1277	725	1023	812
43-point	488	395	483	372	487	417	463	445
25-point	299	240	278	239	283	237	303	273
13-point	135	133	134	136	135	133	134	146
7-point	76.3	94.2	75.4	92.9	75.8	94.2	75.9	99.2

Figure 4.10: Matrix of metric results for IBM Blue Gene/P. These matrices show data cache misses for L1 and L2, GFlops, *FP/Cache*, MUPS and memory traffic metrics for five stencil sizes with each different optimization method.

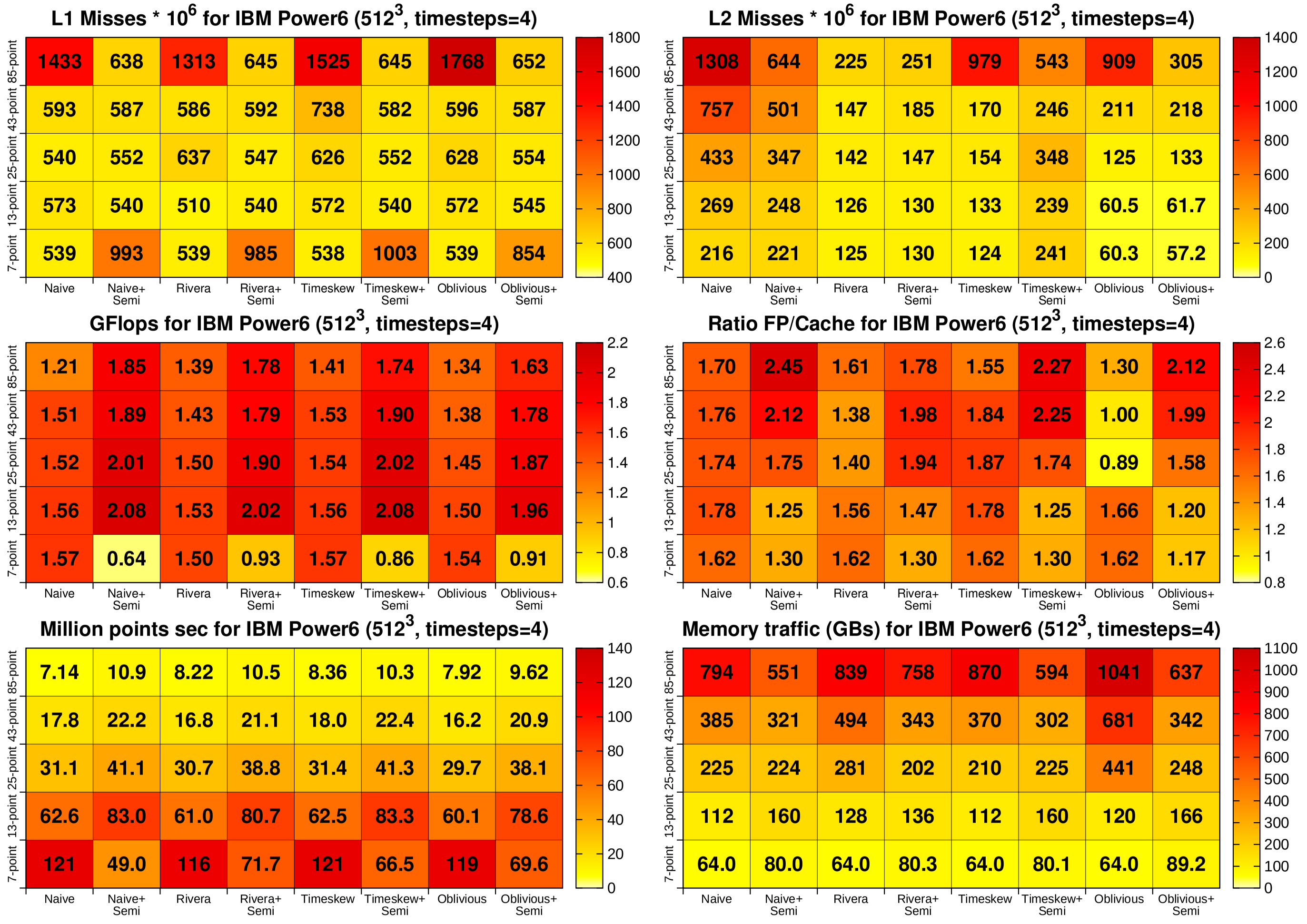


Figure 4.11: Matrix of metric results for POWER6. These matrices show data cache misses for L1 and L2, GFlops, *FP/Cache*, MUPS and memory traffic metrics for five stencil sizes with each different optimization method.

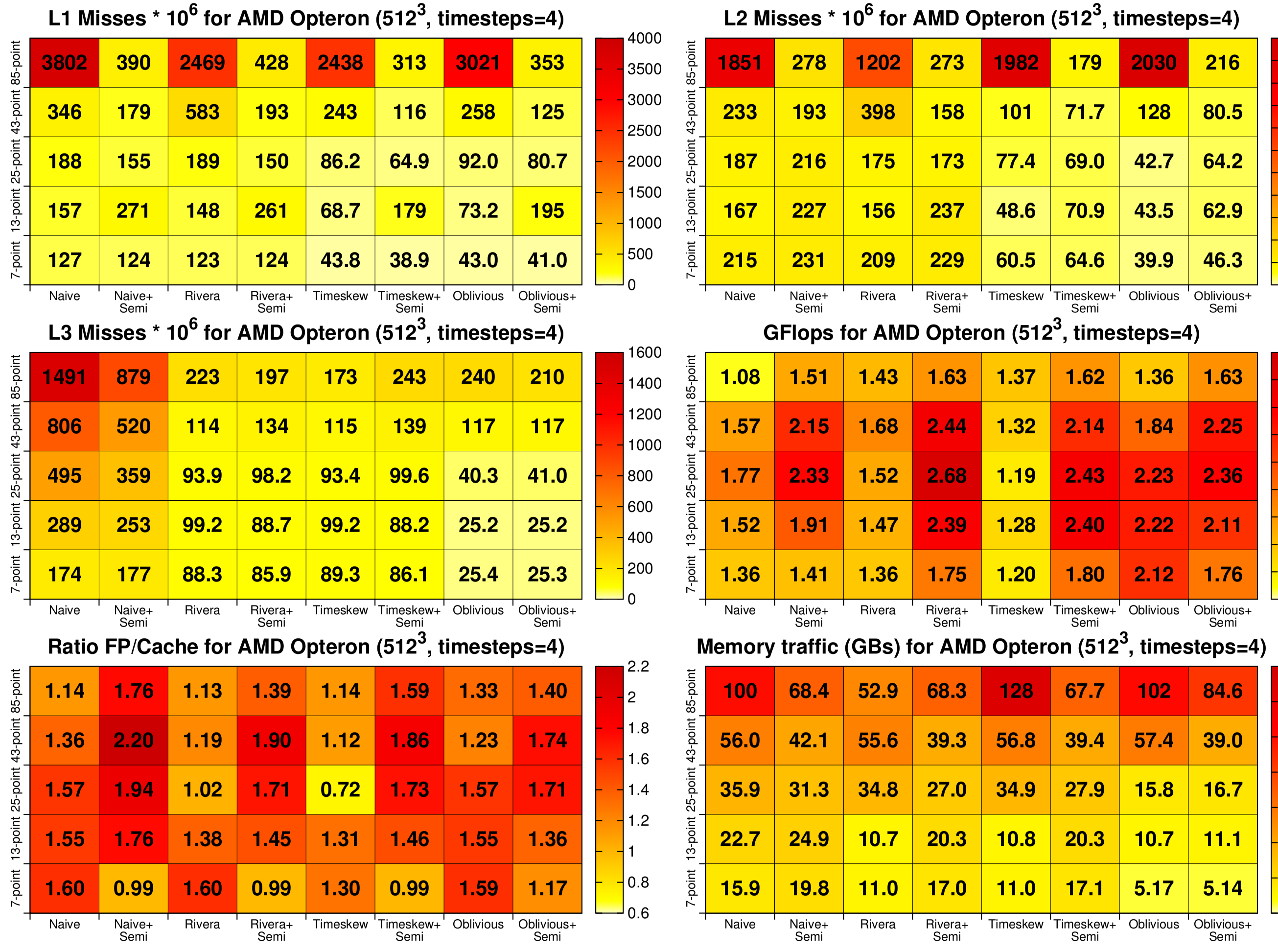
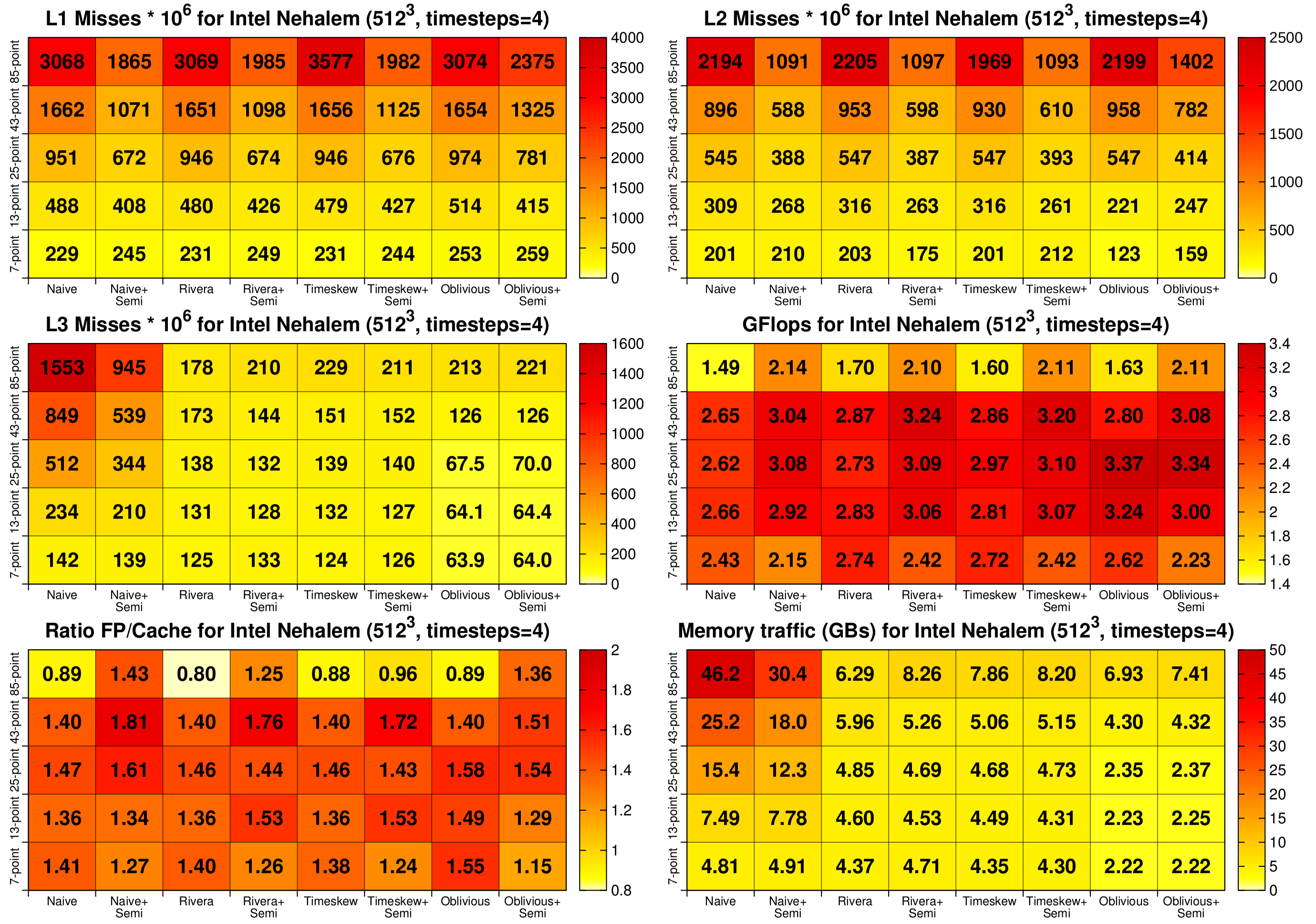


Figure 4.12: Matrix of metric results for AMD Opteron. These matrices show data cache misses for L1, L2 and L3, GFlops, *FP/Cache* and memory traffic metrics for five stencil sizes with each different optimization method.



L1 Misses * 10^6 for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	3068	1865	3069	1985	3577	1982	3074	2375
43-point	1662	1071	1651	1098	1656	1125	1654	1325
25-point	951	672	946	674	946	676	974	781
13-point	488	408	480	426	479	427	514	415
7-point	229	245	231	249	231	244	253	259

L2 Misses * 10^6 for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	2194	1091	2205	1097	1969	1093	2199	1402
43-point	896	588	953	598	930	610	958	782
25-point	545	388	547	387	547	393	547	414
13-point	309	268	316	263	316	261	221	247
7-point	201	210	203	175	201	212	123	159

L3 Misses * 10^6 for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	1553	945	178	210	229	211	213	221
43-point	849	539	173	144	151	152	126	126
25-point	512	344	138	132	139	140	67.5	70.0
13-point	234	210	131	128	132	127	64.1	64.4
7-point	142	139	125	133	124	126	63.9	64.0

GFlops for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	1.49	2.14	1.70	2.10	1.60	2.11	1.63	2.11
43-point	2.65	3.04	2.87	3.24	2.86	3.20	2.80	3.08
25-point	2.62	3.08	2.73	3.09	2.97	3.10	3.37	3.34
13-point	2.66	2.92	2.83	3.06	2.81	3.07	3.24	3.00
7-point	2.43	2.15	2.74	2.42	2.72	2.42	2.62	2.23

Ratio FP/Cache for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	0.89	1.43	0.80	1.25	0.88	0.96	0.89	1.36
43-point	1.40	1.81	1.40	1.76	1.40	1.72	1.40	1.51
25-point	1.47	1.61	1.46	1.44	1.46	1.43	1.58	1.54
13-point	1.36	1.34	1.36	1.53	1.36	1.53	1.49	1.29
7-point	1.41	1.27	1.40	1.26	1.38	1.24	1.55	1.15

Memory traffic (GBs) for Intel Nehalem (512^3, timesteps=4)

	Naive	Naive+ Semi	Rivera	Rivera+ Semi	Timeskew	Timeskew+ Semi	Oblivious	Oblivious+ Semi
85-point	46.2	30.4	6.29	8.26	7.86	8.20	6.93	7.41
43-point	25.2	18.0	5.96	5.26	5.06	5.15	4.30	4.32
25-point	15.4	12.3	4.85	4.69	4.68	4.73	2.35	2.37
13-point	7.49	7.78	4.60	4.53	4.49	4.31	2.23	2.25
7-point	4.81	4.91	4.37	4.71	4.35	4.30	2.22	2.22

Figure 4.13: Matrix of metric results for Intel Nehalem. These matrices show data cache misses for L1, L2 and L3, GFlops, *FP/Cache* and memory traffic metrics for five stencil sizes with each different optimization method.

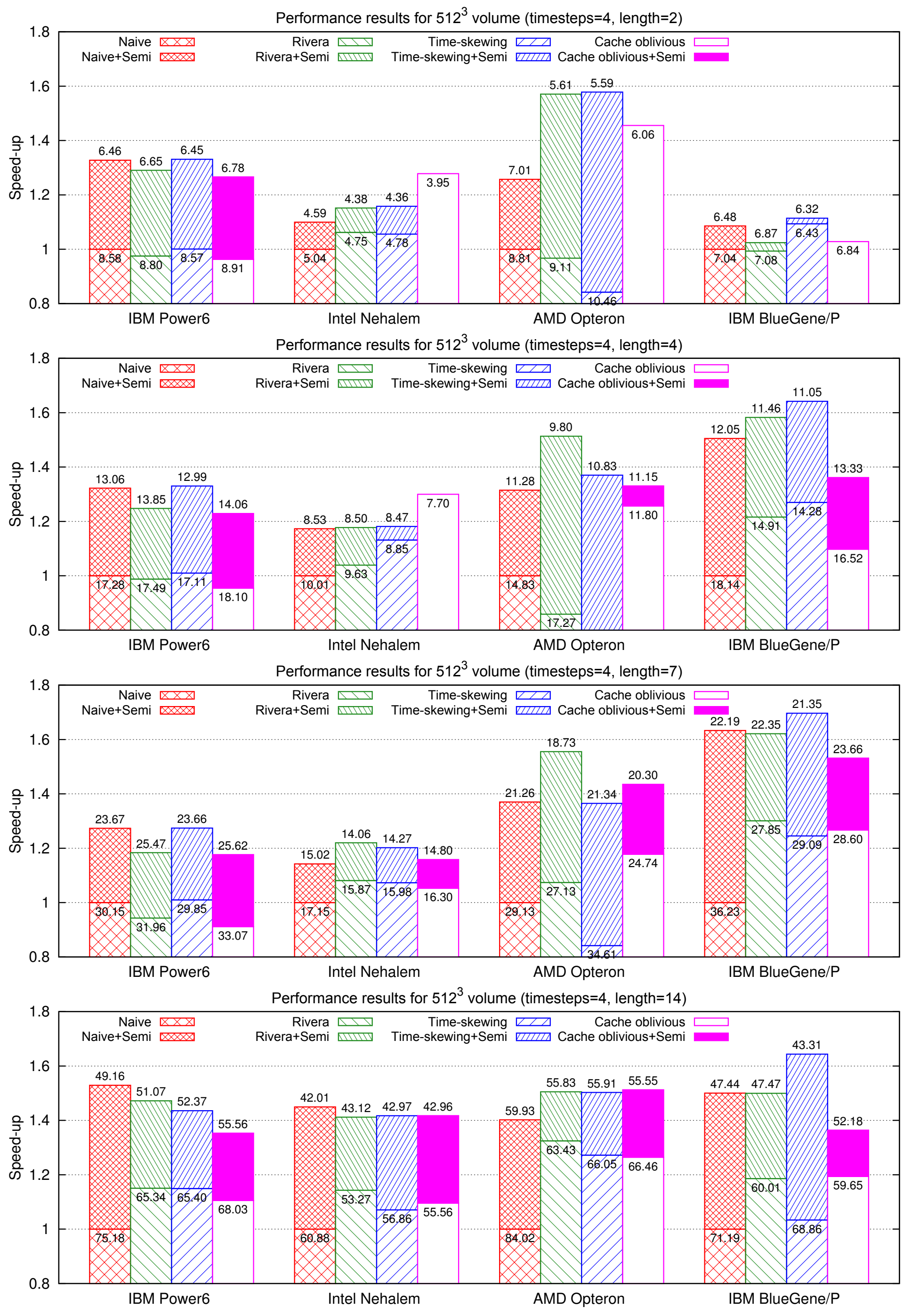
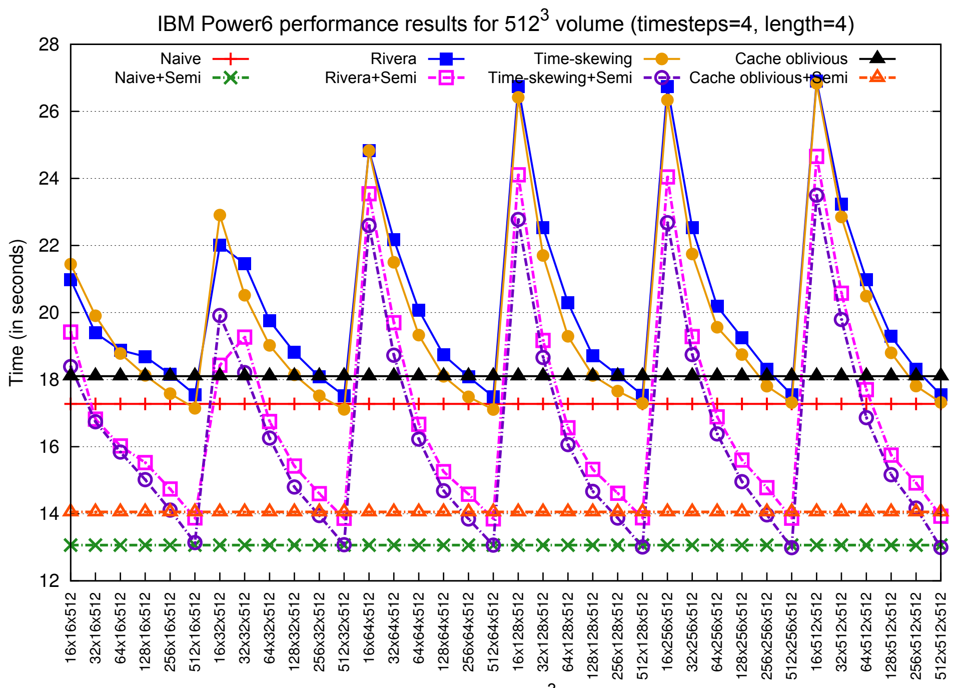


Figure 4.14: Speed-up results with respect to the baseline algorithm (Naive implementation). Numbers shown in bars represent the total time in seconds for each execution (lower is better).



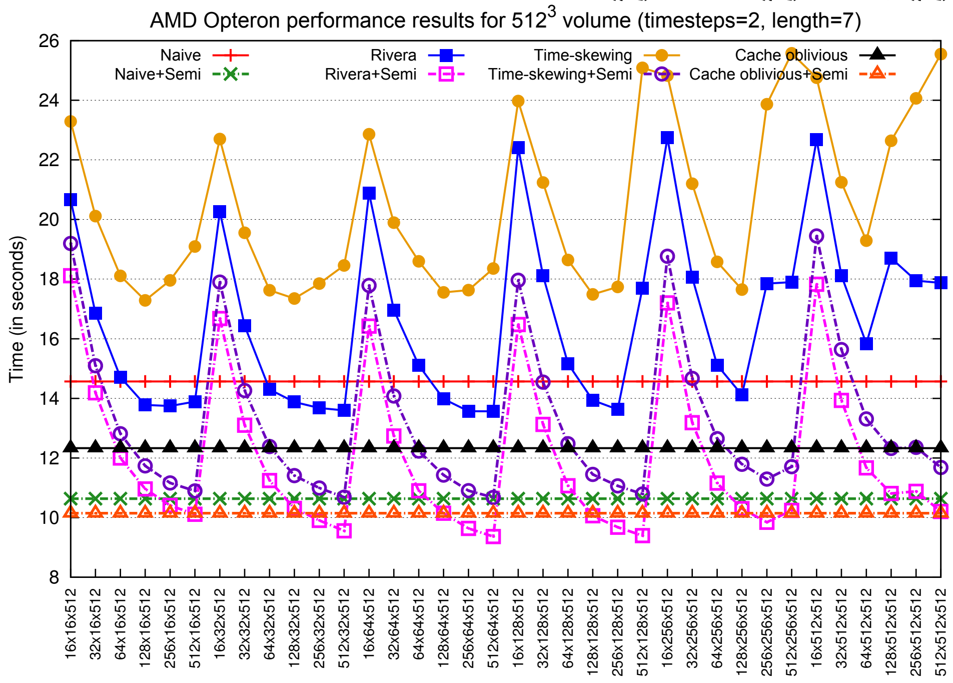


Figure 4.15: IBM POWER6 and AMD Opteron execution times for all the algorithmic combinations, where the blocking parameter has been changed to discover the optimum value. The algorithms which do not support blocking have been plotted using the best time obtained.

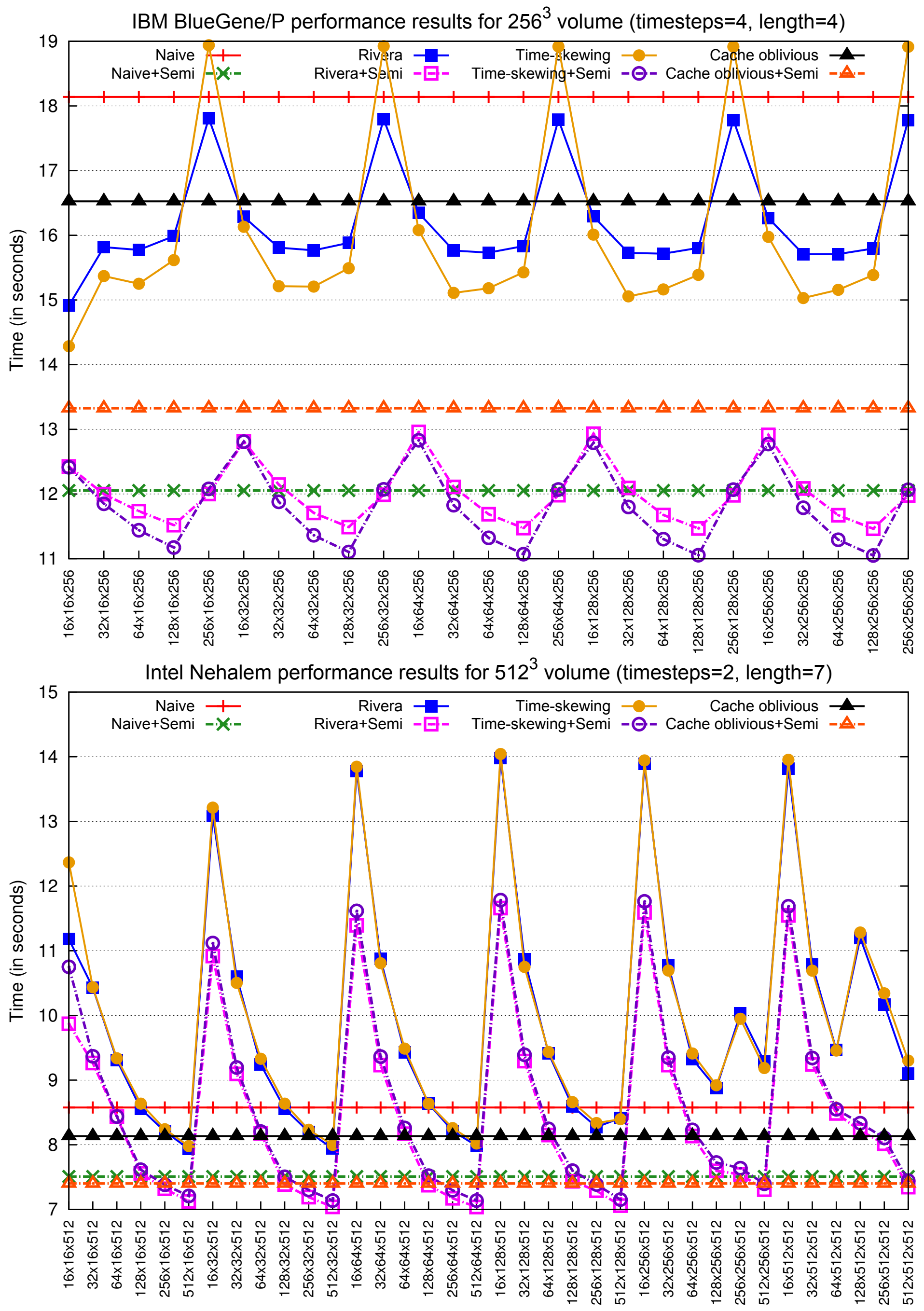


Figure 4.16: IBM BlueGene/P and Intel Nehalem execution times for all the algorithmic combinations, where the blocking parameter has been changed to discover the optimum value. The algorithms which do not support blocking have been plotted using the best time obtained.

the Semi-stencil versions are the best implementations. The speed-ups, with respect to the naive code, ranges from $\approx 1.20\times$ on the Intel Nehalem to $1.60\times$ for BG/P, which is the most favored architecture. On the Intel Nehalem, the classical Cache-oblivious version slightly outperforms the Semi-stencil version. It is likely that the `-fast` option on the Intel compiler is quite aggressive and optimizes stencil codes reasonably well. On the contrary, the AMD Opteron appears to have some kind of problem with Time-skewing, where the compiler does not appear able to generate optimized binaries for this code. The compiler used on this architecture was GCC v4.1. Finally, on both IBM platforms a substantial gain is observed. First, on BG/P a speed-up of 1.64 is obtained, basically because Semi-stencil helps to mitigate cache-line conflicts in the small L1/L2 caches (32kB and 1920B). Second, on the POWER6 architecture, space and time-blocking techniques do not have any substantial effect due to the large memory hierarchy (64kB, 4MB and 32MB) and the sophisticated hardware prefetching system. However, Semi-stencil is able to improve the execution time by 25% due to the lower number of loads issued into the memory system.

4.4.4 SMP Performance

In this subsection, the parallel aspect of the algorithms is evaluated. The multiprocessing version of each stencil algorithm version has been augmented using OpenMP pragmas. We have tested certain combinations of the algorithms on three different architectures: Intel Sandy Bridge, IBM POWER7 and the latest Intel Xeon Phi (MIC). Three specific architectures that have been chosen due to their quasi-massive parallel capabilities, able to run simultaneously with 8, 32 and 244 threads per chip respectively.

The parallel implementation of each stencil code has been designed following specific decomposition strategies. First, the Naive parallelization has been carried out cutting the least-stride dimension (NY) by the number of threads in order to deal with data locality. Each thread computes *thread blocks* of size $NZ \times NX \times TK_{SMP}$, where $TK_{SMP} = NY/Threads$. Second, in order not to affect the Rivera-base scheme performance, the serial search of the best TJ block size parameter has been respected as far as possible when decomposing the problem size. The block parameter is computed as: $TJ_{SMP} = TJ$ if $TJ * Threads \leq NX$, or $TJ_{SMP} = NX/Threads$ otherwise. Hence, each thread computes blocks of size $NZ \times TJ_{SMP} \times NY$. Finally, time blocking algorithms have been parallelized by applying thread synchronization methods to keep data dependencies between thread domain computations. Figure 4.17 shows the Time-skewing [42, 87] and Cache-oblivious [37] parallel strategies as discussed in Section 4.2.2.

Table 4.7 shows the parallel results on the POWER7 architecture for a large problem (512 $\times$ 1024 $\times$ 1024) in order to exploit the parallel capabilities (up to 32 threads). This table shows the execution time and scalability for each algorithm using the classical (top) and the Semi-stencil implementation (bottom) for varying numbers of OpenMP threads. Threads has been progressively increased until fully populate all 8 cores with 1, 2 and 4 SMT threads.

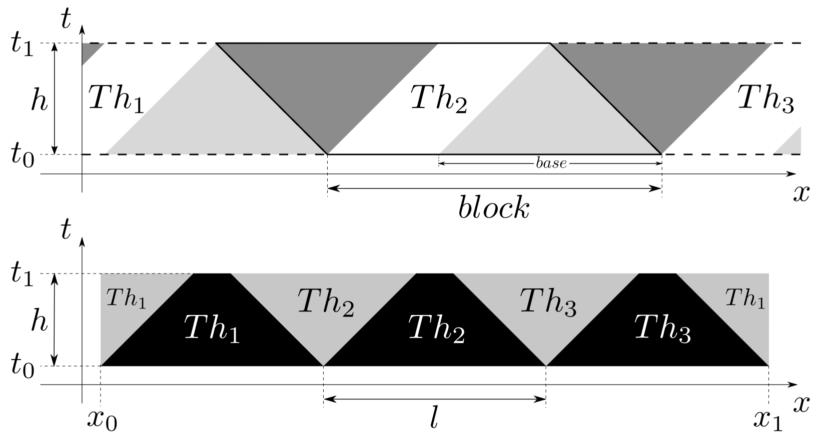


Figure 4.17: Thread decomposition strategies for time blocking algorithms. Top: in Time-skewing, each thread block (size $block$) is divided into three parallelepipeds and executed in the next order at the same time by all threads: firstly the light gray (size $base = 2{*}\ell{*}(t{-}1){+}1$), secondly the white and finally the dark gray area. Bottom: in Cache-oblivious, each thread block is divided into inverted and non-inverted trapezoids (size l) and executed in this order.

Reviewing the results, we observe clearly that Semi-stencil enhances the global performance in almost all cases, except, partially, in Cache-oblivious. In some cases, performance is nearly doubled (1.68 and 1.83×) when compared with the classical algorithm running the same number of threads and baseline algorithm (2 and 4 threads for Naive and Timeskew respectively). Besides, the Semi-stencil scalability results are mostly similar to the classical algorithm except on 8 and 16 threads runs, where a substantial degradation in performance is observed.

Thread level parallelism in IBM POWER7 for 512×1024×1024 (timesteps=2, length=4)

Algorithm	1 thread	2 threads	4 threads	8 threads	8×2 threads	8×4 threads
Naive	39.81	19.98 (1.99)	10.01 (3.97)	6.46 (6.15)	4.76 (8.35)	2.47 (16.08)
Rivera	33.48	16.81 (1.99)	8.73 (3.83)	8.21 (4.07)	5.80 (5.77)	2.77 (12.08)
Timeskew	39.50	19.84 (1.99)	9.96 (3.96)	6.59 (5.99)	4.64 (8.51)	3.36 (11.75)
Oblivious	23.86	11.92 (2.00)	5.96 (4.00)	**4.85 (4.91)**	**2.86 (8.33)**	2.04 (11.64)
N + *Semi*	23.45	11.83 (1.98)	5.95 (3.94)	5.87 (3.99)	3.37 (6.94)	2.36 (9.90)
R + *Semi*	22.48	11.30 (1.99)	5.66 (3.96)	4.87 (4.61)	4.82 (4.66)	2.13 (10.53)
T + *Semi*	21.47	**10.90 (1.97)**	**5.45 (3.94)**	5.54 (3.87)	3.83 (5.59)	2.92 (7.35)
O + *Semi*	22.80	11.46 (1.99)	5.71 (3.98)	5.57 (4.09)	3.47 (6.55)	**1.74 (13.06)**

Table 4.7: POWER7 SMP results among the different algorithms. Numbers shown represent the total time in seconds (lower is better). Scalability results with respect to the 1 thread execution are shown in parenthesis.

Furthermore, the POWER7 architecture scales well considering that despite having 32 threads, only 8 real cores exist per chip, achieving the impressive scalability of ≈16× for the Naive case and ≈13× for Semi-stencil case. This behaviour shows that the 4 way SMT feature maximizes the processor core throughput by offering an increase in efficiency.

This section is concluded by looking at two Intel-based platforms; a representative of the Xeon family and an early access to a Many Integrated Core (MIC) Knight Corner based system with 61 cores (SE10P).

For our experiments on the Sandy Bridge, the hyper-threading capability were turned off. Additionally, only one of the sockets was used since the micro-benchmark does not take advantage of the NUMA configuration (memory banks associated to sockets). Besides this, the testing on the Intel architectures followed the parallelism guidelines stated above. Regarding the software stack, the code was compiled and profiled with Intel tools, where the only outstanding compiler flags used were `-avx` and `-DAVX` macro in order to utilize Advanced Vector Extensions (AVX) instrinsics.

Thread level parallelism in Sandy Bridge for 640^3 (timesteps=4, length=4)

Algorithm	1 thread	2 threads	4 threads	6 threads	8 threads
Naive	6.77	3.59 (1.88)	2.48 (2.73)	2.45 (2.76)	2.47 (2.74)
Rivera	5.27	2.67 (1.97)	1.39 (3.79)	1.00 (5.27)	0.84 (6.27)
Timeskew	5.29	2.71 (1.95)	1.46 (3.62)	1.06 (4.99)	0.95 (5.57)
Oblivious	5.82	3.01 (1.93)	1.60 (3.64)	1.15 (5.06)	1.00 (5.82)
N + *Semi*	5.19	2.86 (1.81)	2.00 (2.89)	1.99 (2.60)	2.03 (2.56)
R + *Semi*	4.35	**2.21 (1.97)**	**1.18 (3.69)**	**0.90 (4.83)**	**0.82 (5.30)**
T + *Semi*	4.48	2.33 (1.92)	1.28 (3.50)	0.97 (4.62)	0.92 (4.87)
O + *Semi*	4.72	2.46 (1.92)	1.34 (3.52)	1.01 (4.67)	0.94 (5.02)

Table 4.8: Sandy Bridge (1 socket) SMP results for the different algorithms. Numbers shown represent the total time in seconds. Scalability results with respect to the 1 thread execution are shown in parenthesis.

The dataset sizes selected for the tests were 640^3 and $256\times4096\times256$ on Sandy Bridge and MIC respectively. These sizes are enough to overwhelm the memory hierarchy and also to bear a reasonable load-balance among OpenMP threads. Table 4.8 shows moderate scalability for Sandy Bridge (up to $6.27\times$) on Rivera, Rivera+*Semi-stencil*, Oblivious and Oblivious+*Semi-stencil*. Nevertheless, Naive results lead to the worst scalability results. Clearly, the current domain decomposition (cutting the least-stride dimension) is not the most appropriate approach for this case.

Also, the Semi-stencil algorithm reduces the execution time on 8 cores by only 3% on the worst case (Rivera) to 18% on the better case (Naive). Furthermore, the larger is the number of threads, less is the advantage of the Semi-stencil strategy over the algorithms. It is likely that memory channels saturate earlier, thus achieving the peak memory bandwidth and hampering the linear scalability for a large number of threads.

Finally, Table 4.9 shows MIC results for a 25-point stencil, where threads has been progressively augmented until populating its 61 cores with 1, 2 and 4 SMT threads. The best scalability for the 244 threads case is achieved by Rivera+*Semi-stencil* algorithm with $93.8\times$, Thread level parallelism in KNC for 256×4096×256 (timesteps=4, length=4)

Algorithm	1 thread	61 threads	61×2 threads	61×3 threads	61×4 threads
Naive	49.71	0.888 (55.9)	0.658 (75.5)	0.632 (78.6)	0.627 (79.2)
Rivera	45.36	0.813 (55.8)	0.595 (76.2)	0.515 (88.0)	0.594 (76.3)
Timeskew †	29.46	0.575 (51.2)	0.465 (63.3)	0.599 (49.2)	0.694 (42.4)
Oblivious	32.90	0.593 (55.5)	0.456 (72.1)	0.571 (57.6)	0.694 (47.4)
N + *Semi*	38.89	0.710 (54.7)	0.535 (72.7)	0.560 (69.4)	0.587 (66.2)
R + *Semi*	44.21	0.780 (56.6)	0.511 (86.5)	**0.457 (96.7)**	**0.471 (93.8)**
T + *Semi* †	21.13	**0.452 (46.7)**	**0.426 (49.6)**	0.512 (41.3)	0.599 (35.3)
O + *Semi*	34.16	0.601 (56.8)	0.472 (72.3)	0.503 (67.9)	0.576 (59.3)

Table 4.9: Knight Corner beta 0 SMP (balanced scheduling) results. Numbers represent the total time in seconds (scalability is shown in parenthesis). †Due to algorithmic constraints on Timeskew (thread decomposition is performed on least-stride dimension), tests were conducted using a 256×256×4096 dataset to enable parallelization with a large number of threads.

which corresponds to a 1.26× improvement over the Rivera baseline. Another worthy result is the combination of Time-skewing and Semi-stencil; this case offers the best performance in execution time (0.426 seconds with 122 threads) among all algorithms. However, it should be clarified that for this case, the dataset size was rearranged and the timestep was split in order to satisfy the parallelepipeds' size constraints (size $base$) for a decomposition with a large number of threads (see Figure 4.17 (top)).

4.5 Summary

In this chapter, a novel optimization technique for stencil-based computations is presented. This new algorithmic approach, called *Semi-stencil*, is especially well suited for scientific applications on cache-based architectures and in particular, for large stencil sizes.

The proposed technique is relevant because it deals with a number of well known problems associated with stencil computations. First, due to the lower number of reads that are issued it improves the floating-point operation to data cache access ratio (*FP/Cache*), which also entails a higher data re-utilization. Second, it modifies the memory access pattern, reducing the cache pressure and enhancing the spatial and temporal locality for the accessed data. In addition, due to its orthogonal property, the Semi-stencil outperforms other techniques on many architectures, either when implemented alone or combined with space and time-blocking algorithms.

On the SMP experiments, the Semi-stencil algorithm also improves the classical approach for most cases presented in this thesis. Nevertheless, the highest attainable scalability remains low for all three reviewed platforms. It achieves scalability efficiencies of only 41%, 66% and 38% for POWER7, Sandy Bridge and MIC respectively with the maximum number of threads. Next chapter discusses new parallel strategies focused on improving the scalability on modern multi- and many-core architectures.

Chapter 5

SMT, Multi-core and Auto-tuning Optimizations

The raise of the multi-core paradigm is already a fact. Computing industry has moved from high-frequency scaling toward chip multiprocessors (CMP) in order to break the limiting wall of performance and energy consumption [10]. In this approach, conventional cores are replicated and arranged in the same die, sharing memory resources as cache memories or prefetching engines. Nowadays, several mainstream platforms coexist with a variety of architectural features. Programmers have to face multi-core chips, simultaneous multithreading (SMT), SIMD capabilities, complex memory hierarchies with enhanced prefetching systems and NUMA arrangements due to multi-sockets implementations.

Moreover, the new trend towards many-core architecture is also a reality. Emerging platforms such as the Intel Xeon Phi architecture, including the MIC chip (Many Integrated Cores) with +60 cores and 4-way SMT capabilities, is a clear example. More novel many-cores architectures are going to appear. For instance, Intel has already announced the next version of the MIC chip in 2016, the Knights Landing built using up to 72 cores with 4-way SMT and out-of-order execution model. This tremendous parallelism (288 concurrent threads) poses optimization challenges to effectively and fully utilize the system's resources. As more of these massively parallel architectures appear in the market, specific parallel strategies should be explored to achieve suboptimal implementations of the computational kernels.

In Chapter 4, we have reviewed the current state-of-the-art in stencil optimizations at core level. Above all the highlighted algorithms in that chapter, spatial-blocking algorithms have provided higher performance portability and support across a wide breadth of numerical problems utilizing stencil computations in their structured meshes due to their simplicity of implementation. On the other hand, time-blocking algorithms (time-skewing and cache-oblivious), despite of their compelling performance, pose implementation difficulties when they are combined with I/O, boundary conditions or communications tasks in real-life applications. Time-blocking performs loop unrolling over time-step sweeps, therefore interfering in the execution of the remaining tasks which might be executed at each time-step. These issues make spatial-blocking a better candidate to be considered in general optimizations for stencil computations.

Nevertheless, spatial-blocking is very sensitive to the problem domain size and the pressure in the cache hierarchy in order to report remarkable improvements in performance. The larger the problem and the pressure in the cache hierarchy is, the better performance the spatial-blocking algorithm produces. Consequently, to effectively obtain a suboptimal implementation with spatial-blocking, a parametrical search of the tiling ($TI \times TJ \times TK$) must be performed for each problem size and architectural setup. At multi-core and many-core optimization level, where register pools, functional units, cache memories and prefetching engines are shared across computational threads, this sensitivity effect is magnified even more. As a result, the choice of the best tiling parameter is not only input-dependent but also architecture-dependent. With the broad disposal of computer architecture configurations, auto-tuning has emerged as a feasible approach to tune computational kernels [85], exploring all variants in less time than a human-driven strategy. Therefore, an auto-tuning framework at runtime level in combination with other approaches may allow to leverage the performance in existing and emerging architectures.

This chapter proposes a comprehensive set of general multi-core and many-core optimizations for stencil computations that aim to maximize overall performance by balancing the workload among the computational threads. For this purpose, we have devised a group of domain decomposition strategies at node-level that effectively reuse the shared resources of the cores across threads (cache memories and prefetching engines) considering all features in emerging platforms. We also present a simple and straightforward auto-tuning engine that explores and selects the best parametrical combinations for our implemented spatial-blocking algorithms in order to minimize the runtime. Combining both proposals, the domain decomposition policies and the auto-tuner framework, the chances to speed-up stencil kernels on multi-core platforms are promising.

5.1 State of the Art

The most remarkable study about domain decomposition and auto-tuning over stencil computations on multi-core platforms was developed by the CDR group in the Lawrence Berkeley National Laboratory [22, 23, 41]. Datta *et al.* proposed a set of multi-core strategies to optimize stencils at node-level: data allocation policies (NUMA aware and padding), a four-level domain decomposition strategy and low-level optimizations (prefetching and cache-bypass). Then, they combined all these optimization strategies as decision tree rules in a first component (a Perl code generator) of an auto-tuning framework that produced multithreaded C code. This approach allowed them to generate easily a vast combination of stencil codes encompassing all the optimization techniques. Finally, a second component of the auto-tuner performed an explicit search looking for local minima using heuristics for constraining the search space.

Figure 5.1 shows the four-level decomposition employed in their study. First, a node block of size $NX \times NY \times NZ$, that consists in the entire problem to solve, is partitioned into smaller core blocks of size $CX \times CY \times CZ$. In their work, X is the unit-stride whereas the Z is least-

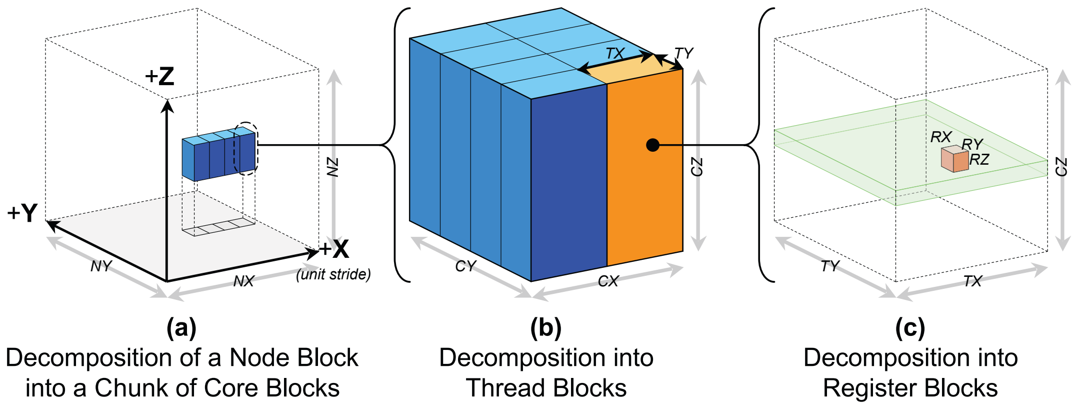

+Z
+Y
+X
(unit stride)
NY
NX
NZ
TX
TY
CY
CX
CZ
RX
RY
RZ
TY
TX
CZ
(a)
Decomposition of a Node Block into a Chunk of Core Blocks
(b)
Decomposition into Thread Blocks
(c)
Decomposition into Register Blocks

Figure 5.1: Datta and *et al.* proposed this four-level problem decomposition: In (a), a node block (the full grid) is broken into smaller chunks. All the core blocks in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A properly sized core block should avoid capacity misses in last level cache. A single thread block from the core block in (b) is then magnified in (c). A thread block should exploit common resources among threads. Finally, the magnified thread block in (c) is decomposed into register blocks, which exploit data level parallelism. This figure was taken from Kaushik Datta article [22].

stride dimension. Each core group is processed by threads belonging to the same core. This first decomposition is intended to avoid last level cache (LLC) capacity misses among different cores in the same chip by blocking the problem. In addition, in a second decomposition, contiguous core blocks can be also grouped together into chunks of size $ChunkSize$, being processed by the same core and its threads. This second scheme strives to explain shared cache locality, cache blocking, register blocking and NUMA-aware allocation. In a third step, core blocks are further decomposed into thread blocks of $TX \times TY \times CZ$ size (Figure 5.1.b). Thread blocks respect CZ size of core blocks, therefore when $TX = CX$ and $TY = CY$, only one thread per core block is assigned. This additional decomposition allows to exploit shared cache resources that may exist across threads. Chunks of core blocks are assigned to threads running in a core block ($CX/TX \times CY/TY$) in a round-robin fashion. Finally, a fourth decomposition is conducted through each thread block into register blocks of $RX \times RY \times RZ$ size, taking the advantage of the data level parallelism of SIMD instructions and the available registers.

As a summary, Datta *et al.* establish some interesting strategies for multi-core architectures in order to find their best stencil configuration. Their approach generates a massive variety of stencil codes at compile-time through a code generator tool of their auto-tuning framework considering all the different strategies that they propose. Then, using an iterative greedy search algorithm and heuristics to reduce the search space, they find a suboptimal configuration for a given platform. Nevertheless, this brute-force approach based on trial and error has considerable costs in terms of search and computation time. This execution flow must be performed prior the real scientific executions for each architectural setup and problem size. Therefore, if the number of spawned threads are changed or the problem size slightly modified, the second step of the auto-tuning environment must be performed once again to find the best decomposition. In addition, in their four-level decomposition, the considerations regarding the cooperation of the cache hierarchy and the prefetching effect across SMT threads are leaved aside. In the following sections, we propose specific strategies that intend to cover all compute levels (node, socket, core and SMT) in order to tackle efficiently the stencil scalability on modern architectures. We support this decomposition methodology with numerical results that strongly corroborate our initial thoughts.

5.2 Simultaneous Multithreading Awareness

Simultaneous Multithreading (SMT) is a Thread Level Parallelism (TLP) technique that increases the core throughput without the necessity of explicitly increasing the hardware resources. SMT enables multiple and independent threads to be running on the same hardware core sharing resources such as: register pools, functional units, memory (usually L1 and L2 caches) and their hardware prefetchers.

Mapping computational kernels to architectures with SMT features requires careful design and specific domain decomposition strategies to effectively exploit TLP capabilities. Otherwise, threads spawned and pinned to the same core can provoke collateral effects (e.g. *ping-pong* phenomena over data cache) that may harm the overall performance due to resource contention.

In memory-bound applications, such as stencil computations, data reuse is a key factor to improve the speed-up by reducing the memory footprint and increasing the Operational Intensity (OI) of the kernel. Therefore, when mapping stencil computations to SMT architectures the domain decomposition strategies must be rearranged to take advantage of data reuse between SMT threads. To this extent, SMT threads may reuse the neighbour points to compute the spatial operator ($\mathcal{X}^{t-1}_{i\pm1..\ell,j\pm1..\ell,k\pm1..\ell}$), thus minimizing the dataset to compute a subspace of the domain. As a consequence, compulsory and mainly capacity misses should be reduced.

In 3D stencil computations, we have devised three different ways of scheduling SMT threads in order to promote data cooperation across them:

- **Row distribution**: computational blocks of one single point are interleaved across threads through the unit-stride dimension (Z) (see Figure 5.2.Left).
- **Column distribution**: each thread traverses entire columns of size Z in an interleaving fashion computing blocks of Z consecutive points (see Figure 5.2.Center).
- **Plane distribution**: every Z-X plane of the domain is swept by only one thread, interleaving computational blocks through the least-stride dimension (see Figure 5.2.Right).

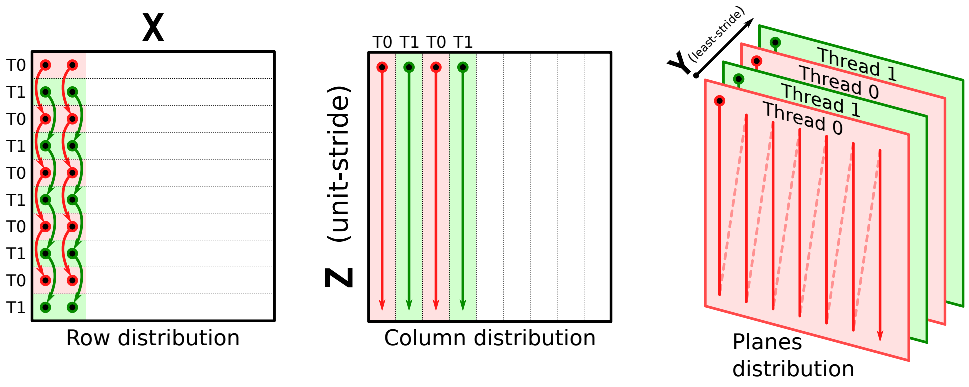


Figure 5.2: Data distributions for SMT threads. Planes depicted here lay over the Z-X axes. In this example a 2-way SMT core is shown. Data is traversed in a column-order fashion from left-top to right-bottom. The computational blocks are scheduled in a round-robin fashion (similar to the `schedule(static,1)` clause of OpenMP).

All the three traversing strategies for SMT threads promote data cooperation but only one of them is the most appropriate for a given architecture and stencil topology. Deciding the approach to be used may strongly depend on the prefetching capabilities. In a transient state of a stencil computation, where columns over X dimension do not produce conflict and capacity misses among them, one prefetching stream is triggered for each Z-X $\mathcal{X}^{t-1}_{i,j,k\pm1..\ell}$ plane accessed (see Section 6). Therefore, in a star-like stencil computation of order ℓ, up to $2*\ell+1$ streams are kept simultaneously in order to perform one single output plane.

Considering this scenario, at least four are the main reasons for scheduling SMT threads in a plane (Z-X) distribution fashion. First, despite of having constant-stride prefetchers, caches prefetch data in a more efficient way when unit-stride accesses are performed. Second, a higher ratio of warm-ups (depicted with circles in Figure 5.2) and look-ahead effects (see Section 3.6) can appear per plane in row and column distribution, interfering and disrupting the streaming effectiveness. Third, planes distribution promotes data reuse in the least-stride dimension, which in turn has a lower spatial locality and a higher probability of being evicted from cache. Fourth, architectures with concurrent streaming support are better exploited and shared across SMT threads in plane distribution scenario where longer unit-stride accesses are performed. The section of code in Algorithm 9 shows how to proceed to achieve a plane distribution among SMT threads. In this algorithm, the 3D stencil problem is distributed along the Y axis (`iniy` and `endy`), arranging together planes assigned to SMT threads that reside in the same physical core.

In order to corroborate these statements, a small test was conducted on Intel Xeon Phi (MIC) platform, which supports up to 4 threads in SMT mode. A particle dispersion model (ADR), which uses a 4th-order stencil in space (13-points), was run over 4 MIC cores, pinning 2 consecutive threads per core. Then, the explicit kernel time and the most relevant hardware metrics were gathered setting the `THRCORE` parameter to 1 (no affinity at all) and 2 (SMT

Algorithm 9 Snippet of C code that implements a plane distribution among SMT threads using OpenMP. `THRCORE` indicates the number of SMT threads running in the same core. Least-stride dimension (`iniy` and `endy`) is distributed across threads and an interleaved access is performed through `k+= THRCORE`.

```
#pragma omp parallel firstprivate(iniy,endy)
{
  int tid = omp_get_thread_num()%THRCORE;   // Thread id in core
  int dom = omp_get_max_threads()/THRCORE;  // Core id
  int nby = (endy - iniy) / dom;
  int rnb = (endy - iniy) % dom;
  if (rnb > dom) iniy = iniy + (++nby)*dom;    // Initial Y domain
  else           iniy = iniy + nby*dom + rnb;  // End Y domain
  endy = MIN(iniy+nby, endy);

  /* Update maingrid */
  for (k= iniy+tid; k< endy; k+= THRCORE) {
    for (j= inix; j< endx; j++) {
      for (i= iniz; i< endz; i++) {
```

affinity across 2 threads). Figures in 5.3 depict the workload distribution followed for this test.

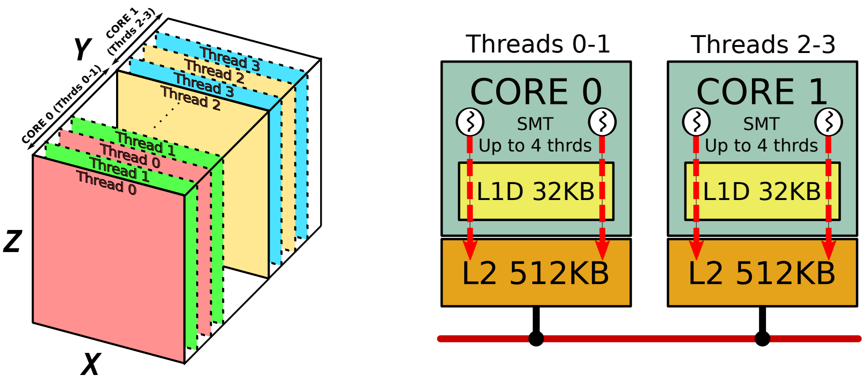


Figure 5.3: Left: Thread affinity strategy proposed for MIC architecture test. Right: MIC diagram of two cores. Two levels of cache are shared per core between SMT threads, L1 and L2 caches.

The results were compelling. All gathered metrics improved when SMT affinity was enabled across threads (see Table 5.1). L1 misses slightly decreased, but above all, L2 demand misses (explicit loads) suffered a reduction of 89% due to the better behavior of the L2 prefetcher, which tracked 79% more cache-lines. Finally, TLB misses were also significantly reduced by 92% because more data thread addresses belonged to the same 4 KB page-boundaries. As a result, the overall time was improved by almost 20%. The number of cores used for this test is not relevant due to hardware metrics are gathered per core and therefore equivalent for any number of cores.

Metric	L1 Misses	L2 Demand Misses	L2 HWP Misses	TLB Misses	Time (in sec)
No affinity	9504992	3492094	2753247	4411206	4.88
SMT affinity	8201066	361439	4933127	346578	3.96
Difference	-14%	-89%	+79%	-92%	**-19%**

Table 5.1: Ash Dispersion model: Intel MIC, 4 cores (2 threads each). Problem size 256x64x1024. Metrics were only measured for the explicit kernel (stencil computation). This hardware metrics correspond to a single core (not aggregated).

5.3 Multi-core and Many-core Improvements

High-level parallelism architectures (high degree of SMP-SMT) pose new challenges to maximize performance due to the complexity of work-load balancing when many threads are cooperating. For example, the optimal scalability can be tampered if the problem lacks of computational domain to fully utilize the resources. In addition, NUMA designs, the intricacy of cache levels and the shared resources across threads add additional complexity to this challenge. Assigning correctly the work-balance across threads can alleviate these issues. This section is devoted to devise strategies in order to reduce imbalance but, at the same time, taking into account the underlying architecture to effectively map resources, threads and computational blocks.

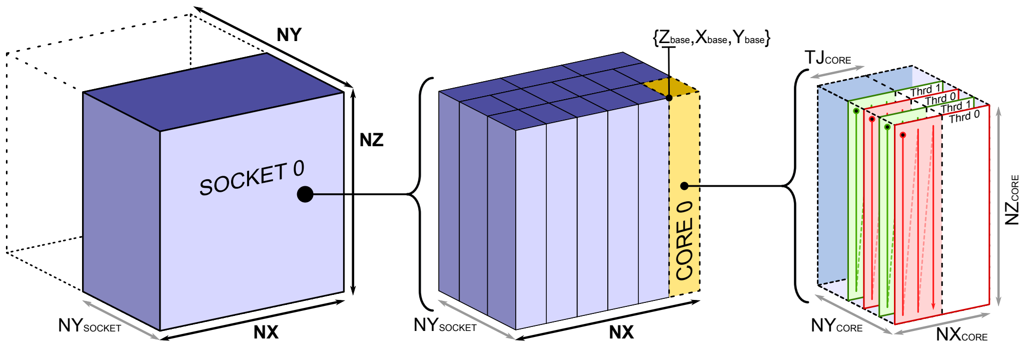


Figure 5.4: Intra-node domain decomposition. A three-level decomposition is performed with our schedulers. Left: first, the intra-node domain is statically split through the least-stride dimension (Y) across all sockets, thus avoiding NUMA effects. Center: each $Socket_{group}$ is decomposed across cores using one of the strategies (static, balanced or guided). This second decomposition strives to minimize imbalance, and prioritizes cutting along X and Y axes to avoid disruption of prefetching. Right: finally, each $Core_{group}$ is traversed by SMT_{group} threads in a interleaving fashion of Z-X planes using the selected TJ parameter from the auto-tuning step. The interleaving should exploit the common resources within the cores, and the TJ parameter should avoid capacity misses in the last level cache.

We propose simplified thread domain schedulers where decomposition is explored in a top-down approach (from socket to core and SMT-level). To that end, a three-level problem decomposition is devised (see Figure 5.4) categorizing threads within the following groups:

- $Socket_{group}$: to reduce slow data transferences across socket links due to NUMA, grid is only decomposed through the least-unit stride dimension (Y). Threads ($thrd_{id}$) are assigned to sockets as $Socket_{group} = MOD(N_{sockets} \times N_{cores} \times N_{SMT}, thrd_{id})$, being $N_{sockets}$, N_{cores} and N_{SMT} the number of sockets, cores per socket and SMT level concurrency respectively.
- $Core_{group}$: threads mapped to the same socket are classified into different cores as $Core_{group} = MOD(N_{cores} \times N_{SMT}, thrd_{id})$. Grid decomposition is either performed over any axis (Z, X or Y).
- SMT_{group}: finally, threads residing in the same core are assigned interlaced domains using the plane distribution strategy defined in Section 5.2.

$Socket_{group}$ distribution allows NUMA-aware allocation, whereas $Core_{group}$ ensures workload balancing across multiple core threads. Finally, SMT_{group} promotes data reuse and streaming concurrency among SMT threads into lower cache hierarchies such as L1 and L2.

Selecting a wise decomposition over $Core_{group}$ is not trivial. Schedulers must decide at runtime which is the best way to decompose the computational domain in order to reduce thread imbalance while sustaining the performance. We set some rules to select the best possible scenario. First, the unit-stride dimension (Z) is never selected for decomposing unless insufficient columns (NX) and planes (NY) are available for the remaining threads ($NX \times NY < nthrs$). This prevents the prefetching disruption. Second, there is a trade-off between $\mathcal{X}^{t-1}$ loads and the stored data of $\mathcal{X}^{t}$. This balance factor (β) is determined by the halo parameter (ℓ) with respect to the interior points as

$$\beta = \frac{NZ \times NX \times NY}{(NZ + 2 \times \ell) \times (NX + 2 \times \ell) \times (NY + 2 \times \ell)} \in (0, 1) \;, \tag{5.1}$$

where each local computed plane within a $Core_{group}$ ($NZ \times NX$) requires $2 \times \ell + 1$ extended planes from $\mathcal{X}^{t-1}$. This balance metric helps to decide which axis is a good candidate for $Core_{group}$ decomposition. The higher the β parameter for a given decomposition, the better ratio of loads and stores has.

Then, to support a wide variety of architectural setups in terms of sockets, cores and SMT configurations, three recursive schedulers differing in the $Core_{group}$ decomposition were developed. These schedulers are recursively called until all groups have been fully decomposed selecting the proper axis. The three work-balance schedulers are:

- Static: one axis is prioritized for cutting along (X or Y) without using the β parameter. However, other axis is selected whether insufficient elements (columns for X and planes for Y) are available for the remaining threads to assign. This strategy is similar to the `schedule(static)` of OpenMP, but adding another degree of freedom that decomposes in the opposite axis if required.

- Balanced: X and Y axes are indistinctly employed, selecting recursively the proper axis to cut that maximizes the β metric.
- Guided: instead of using the β metric to decide the decomposition axis, a previously computed TJ parameter (Rivera algorithm) is used to make the decision. Then, X axis is only recursively cut whether the new NX is still greater than TJ. Otherwise, either NY or NZ are selected depending on β. This strategy is especially designed when the decomposition is combined with auto-tuning techniques.

The result of these schedulers are a set of $\{(Z_{base}, NZ_{core}), (X_{base}, NY_{core}), (Y_{base}, NY_{core})\}$ tuples per thread ($dom[thrd_{id}]$) that bounds each thread to its computational domain. This tuple specifies the starting point and length for each dimension respectively. Every $Core_{group}$ domain must contain at least as many Z-X planes as SMT threads in order to fully populate all running threads. On top of that, spatial-blocking can be stacked on top of $Core_{group}$ blocks, including further data reuse for last level caches. Pseudo-codes for all schedulers are shown in Algorithms 10, 11 and 12.

Algorithm 10 Pseudo-code of the static domain decomposition scheduler for X and Y axes. $axis$ parameter must be set to either to X or Y. The scheduler tries to balance over other axes when not sufficient columns or planes are present.

```
1: procedure Static-Decomposition($Z_{base}$, $X_{base}$, $Y_{base}$, $NZ$, $NX$, $NY$,
                                  $N_{sockets}$, $N_{cores}$, $N_{SMT}$, $thrd_{id}$, $axis$, $dom$)
2:     $nthrs \leftarrow N_{sockets} \times N_{cores} \times N_{SMT}$            ▷ Threads to be assigned in this branch
3:     if $N_{sockets} > 1$ then                                  ▷ $Socket_{group}$ decomposition
4:         $N_{sockets} \leftarrow 1$
5:         for $i \leftarrow 0$ to $N_{sockets} - 1$ do                          ▷ Evenly cut in $NY$
6:             $thrd_{id}^{i} \leftarrow i \times N_{cores} \times N_{SMT}$
7:             $Y_{base}^{i} \leftarrow Y_{base} + \lfloor NY/N_{sockets} \rfloor \times i$
8:             $NY^{i} \leftarrow \lfloor NY/N_{sockets} \rfloor$
9:             Static-Decomposition$^{i}$($\ldots group^{i} args \ldots$)
10:        end for                                         ▷ Insufficient work for all threads
11:    else if $(axis = X \wedge NX < nthrs) \vee (axis = Y \wedge NY < nthrs)$ then
12:        $N_{cores}^{1} \leftarrow \lceil N_{cores}/2 \rceil$                              ▷ Odd or even group
13:        $N_{cores}^{2} \leftarrow \lfloor N_{cores}/2 \rfloor$                                   ▷ Even group
14:        $thrd_{id}^{1} \leftarrow thrd_{id}$                                   ▷ Set thread index
15:        $thrd_{id}^{2} \leftarrow thrd_{id} + N_{cores}^{1}$
16:        if $NX \times NY < nthrs \wedge (N_{cores}^{1} = 1 \vee N_{cores}^{2} = 1)$ then
17:            $Z_{base}^{1} \leftarrow Z_{base}$                                          ▷ Cut in $NZ$
18:            $NZ^{1} \leftarrow \lceil NZ/N_{cores} \rceil \times N_{cores}^{1}$
19:            $Z_{base}^{2} \leftarrow Z_{base} + NZ^{1}$
20:            $NZ^{2} \leftarrow \lfloor NZ/N_{cores} \rfloor \times N_{cores}^{2}$
21:        else if $axis = Y$ then
22:            $Y_{base}^{1} \leftarrow Y_{base}$                               ▷ Cut in $NY$ for $X$ axis
23:            $NY^{1} \leftarrow \lceil NY/N_{cores} \rceil \times N_{cores}^{1}$
24:            $Y_{base}^{2} \leftarrow Y_{base} + NY^{1}$
25:            $NY^{2} \leftarrow \lfloor NY/N_{cores} \rfloor \times N_{cores}^{2}$
26:        else
27:            $X_{base}^{1} \leftarrow X_{base}$                               ▷ Cut in $NX$ for $Y$ axis
28:            $NX^{1} \leftarrow \lceil NX/N_{cores} \rceil \times N_{cores}^{1}$
29:            $X_{base}^{2} \leftarrow X_{base} + NX^{1}$
30:            $NX^{2} \leftarrow \lfloor NX/N_{cores} \rfloor \times N_{cores}^{2}$
31:        end if
32:        Static-Decomposition$^{1}$($\ldots group^{1} args \ldots$)
33:        Static-Decomposition$^{2}$($\ldots group^{2} args \ldots$)
34:    else                          ▷ Assign $Core_{group}$ whether sufficient work is available
35:        for $i \leftarrow 0$ to $N_{cores} - 1$ do                            ▷ Evenly cut in $axis$
36:            $thrd_{id}^{i} \leftarrow thrd_{id} + i \times N_{SMT}$
37:            if $axis = X$ then
38:                $X_{base}^{i} \leftarrow X_{base} + \lfloor NX/N_{cores} \rfloor \times i$
39:                $NX^{i} \leftarrow \lfloor NX/N_{cores} \rfloor$
40:            else
41:                $Y_{base}^{i} \leftarrow Y_{base} + \lfloor NY/N_{cores} \rfloor \times i$
42:                $NY^{i} \leftarrow \lfloor NY/N_{cores} \rfloor$
43:            end if
44:            for $j \leftarrow thrd_{id}^{i}$ to $thrd_{id}^{i} + N_{SMT} - 1$ do               ▷ Assign domain area
45:                $dom[j] \leftarrow \{(Z_{base}^{i}, NZ^{i}), (X_{base}^{i}, NX^{i}), (Y_{base}^{i}, NY^{i})\}$
46:            end for
47:        end for
48:    end if
49: end procedure
```

Algorithm 11 Pseudo-code of the balanced domain decomposition scheduler. This recursive function might be initially called as Balanced-Decomposition$(0, 0, 0, NZ, NX, NY, 2, 8, 2, 0, dom)$ for a platform with 2 sockets, 8 core processor with 2-way SMT capabilities. Computational thread domains are defined with the $2 \times nthrs$ tuple dom, where first dimension is composed of the base address of the domain $(Z_{base}, X_{base}, Y_{base})$ and the local size of the domain $(NZ_{core}, NX_{core}, NY_{core})$ for $thrd_{id}$.

```
 1: procedure Balanced-Decomposition($Z_{base}$, $X_{base}$, $Y_{base}$, $NZ$, $NX$, $NY$,
                                   $N_{sockets}$, $N_{cores}$, $N_{SMT}$, $thrd_{id}$, $dom$)
 2:     $nthrs \leftarrow N_{sockets} \times N_{cores} \times N_{SMT}$          ▷ Threads to be assigned in this branch
 3:     if $nthrs = N_{SMT}$ then                                  ▷ Stop if $SMT_{group}$ is reached
 4:         for $i \leftarrow thrd_{id}$ to $thrd_{id} + nthrs - 1$ do                 ▷ Assign domain area
 5:             $dom[i] \leftarrow \{(Z_{base}, NZ), (X_{base}, NX), (Y_{base}, NY)\}$
 6:         end for
 7:     else if $N_{sockets} > 1$ then                             ▷ $Socket_{group}$ decomposition
 8:         $N_{sockets} \leftarrow 1$
 9:         for $i \leftarrow 0$ to $N_{sockets} - 1$ do                              ▷ Evenly cut in $NY$
10:             $thrd^{i}_{id} \leftarrow i \times N_{cores} \times N_{SMT}$
11:             $Y^{i}_{base} \leftarrow Y_{base} + \lfloor NY/N_{sockets} \rfloor \times i$
12:             $NY^{i} \leftarrow \lfloor NY/N_{sockets} \rfloor$
13:             Balanced-Decomposition$^{i}$(. . . $group^{i}args$ . . . )
14:         end for
15:     else if $N_{cores} > 1$ then                                 ▷ $Core_{group}$ decomposition
16:         $N^{1}_{cores} \leftarrow \lceil N_{cores}/2 \rceil$                                 ▷ Odd or even group
17:         $N^{2}_{cores} \leftarrow \lfloor N_{cores}/2 \rfloor$                                       ▷ Even group
18:         $thrd^{1}_{id} \leftarrow thrd_{id}$                                        ▷ Set thread index
19:         $thrd^{2}_{id} \leftarrow thrd_{id} + N^{1}_{cores}$
20:         if $NX \times NY < nthrs \wedge (N^{1}_{cores} = 1 \vee N^{2}_{cores} = 1)$ then
21:             $Z^{1}_{base} \leftarrow Z_{base}$                                            ▷ Cut in $NZ$
22:             $NZ^{1} \leftarrow \lceil NZ/N_{cores} \rceil \times N^{1}_{cores}$
23:             $Z^{2}_{base} \leftarrow Z_{base} + NZ^{1}$
24:             $NZ^{2} \leftarrow \lfloor NZ/N_{cores} \rfloor \times N^{2}_{cores}$
25:         else if $\beta(NZ, \lfloor NX/2 \rfloor, NY) > \beta(NZ, NX, \lfloor NY/2 \rfloor)$ then
26:             $X^{1}_{base} \leftarrow X_{base}$                                            ▷ Cut in $NX$
27:             $NX^{1} \leftarrow \lceil NX/N_{cores} \rceil \times N^{1}_{cores}$
28:             $X^{2}_{base} \leftarrow X_{base} + NX^{1}$
29:             $NX^{2} \leftarrow \lfloor NX/N_{cores} \rfloor \times N^{2}_{cores}$
30:         else
31:             $Y^{1}_{base} \leftarrow Y_{base}$                                            ▷ Cut in $NY$
32:             $NY^{1} \leftarrow \lceil NY/N_{cores} \rceil \times N^{1}_{cores}$
33:             $Y^{2}_{base} \leftarrow Y_{base} + NY^{1}$
34:             $NY^{2} \leftarrow \lfloor NY/N_{cores} \rfloor \times N^{2}_{cores}$
35:         end if
36:         Balanced-Decomposition$^{1}$(. . . $group^{1}args$ . . . )
37:         Balanced-Decomposition$^{2}$(. . . $group^{2}args$ . . . )
38:     end if
39: end procedure
```

Algorithm 12 Pseudo-code of the guided domain decomposition scheduler. An auto-tuning process must be priorly conducted in order to select a pseudo-optimal TJ parameter.

```
1: procedure Guided-Decomposition($Z_{base}$, $X_{base}$, $Y_{base}$, $NZ$, $NX$, $NY$,
                                  $N_{sockets}$, $N_{cores}$, $N_{SMT}$, $thrd_{id}$, $TJ$, $dom$)

2:     $nthrs \leftarrow N_{sockets} \times N_{cores} \times N_{SMT}$          ▷ Threads to be assigned in this branch

3:     if $nthrs = N_{SMT}$ then                              ▷ Stop if $SMT_{group}$ is reached
4:         for $i \leftarrow thrd_{id}$ to $thrd_{id} + nthrs - 1$ do           ▷ Assign domain area
5:             $dom[i] \leftarrow \{(Z_{base}, NZ), (X_{base}, NX), (Y_{base}, NY)\}$
6:         end for
7:     else if $N_{sockets} > 1$ then                          ▷ $Socket_{group}$ decomposition
8:         $N_{sockets} \leftarrow 1$
9:         for $i \leftarrow 0$ to $N_{sockets} - 1$ do                      ▷ Evenly cut in $NY$
10:            $thrd_{id}^{i} \leftarrow i \times N_{cores} \times N_{SMT}$
11:            $Y_{base}^{i} \leftarrow Y_{base} + \lfloor NY/N_{sockets} \rfloor \times i$
12:            $NY^{i} \leftarrow \lfloor NY/N_{sockets} \rfloor$
13:            Guided-Decomposition$^{i}$($\ldots group^{i} args \ldots$)
14:        end for
15:    else if $N_{cores} > 1$ then                             ▷ $Core_{group}$ decomposition
16:        $N_{cores}^{1} \leftarrow \lceil N_{cores}/2 \rceil$                            ▷ Odd or even group
17:        $N_{cores}^{2} \leftarrow \lfloor N_{cores}/2 \rfloor$                                ▷ Even group
18:        $thrd_{id}^{1} \leftarrow thrd_{id}$                                  ▷ Set thread index
19:        $thrd_{id}^{2} \leftarrow thrd_{id} + N_{cores}^{1}$
20:        if $NX \times NY < nthrs \wedge (N_{cores}^{1} = 1 \vee N_{cores}^{2} = 1)$ then
21:            $Z_{base}^{1} \leftarrow Z_{base}$                                   ▷ Cut in $NZ$
22:            $NZ^{1} \leftarrow \lceil NZ/N_{cores} \rceil \times N_{cores}^{1}$
23:            $Z_{base}^{2} \leftarrow Z_{base} + NZ^{1}$
24:            $NZ^{2} \leftarrow \lfloor NZ/N_{cores} \rfloor \times N_{cores}^{2}$
25:        else if $\beta(NZ, \lfloor NX/2 \rfloor, NY) > \beta(NZ, NX, \lfloor NY/2 \rfloor) \wedge$
26:                $(\lfloor NX/N_{cores} \rfloor \times N_{cores}^{2} >= TJ)$ then
27:            $X_{base}^{1} \leftarrow X_{base}$                                   ▷ Cut in $NX$
28:            $NX^{1} \leftarrow \lceil NX/N_{cores} \rceil \times N_{cores}^{1}$
29:            $X_{base}^{2} \leftarrow X_{base} + NX^{1}$
30:            $NX^{2} \leftarrow \lfloor NX/N_{cores} \rfloor \times N_{cores}^{2}$
31:        else
32:            $Y_{base}^{1} \leftarrow Y_{base}$                                   ▷ Cut in $NY$
33:            $NY^{1} \leftarrow \lceil NY/N_{cores} \rceil \times N_{cores}^{1}$
34:            $Y_{base}^{2} \leftarrow Y_{base} + NY^{1}$
35:            $NY^{2} \leftarrow \lfloor NY/N_{cores} \rfloor \times N_{cores}^{2}$
36:        end if
37:        Guided-Decomposition$^{1}$($\ldots group^{1} args \ldots$)
38:        Guided-Decomposition$^{2}$($\ldots group^{2} args \ldots$)
39:    end if
40: end procedure
```

5.4 Auto-tuning Improvements

Auto-tuning is an optimization methodology that strives for finding the combination of algorithmic parameters that bestows the best performance of a computational code for a given platform and specific problem configuration. To conduct this task, first, all possible algorithmic optimizations must be enumerated (e.g. loop unrolling). Then, an automatic code generator produces all equivalent implementations (code variants) of these enumerated algorithmic optimizations. Finally, a search must be performed by benchmarking the computational code variants with a set of possible parametrical configurations (e.g. degree of loop unrolling), choosing the fastest implementation. As a consequence, being D the algorithmic optimizations to be tested and N their possible parameters, a total of N^D combinations must be tried.

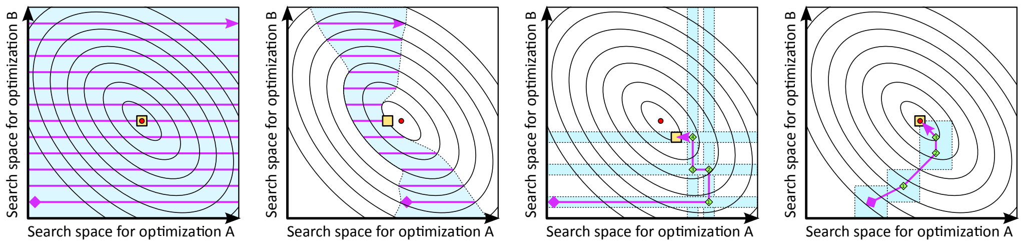

Search space for optimization B
Search space for optimization A
Search space for optimization B
Search space for optimization A
Search space for optimization B
Search space for optimization A
Search space for optimization B
Search space for optimization A

Figure 5.5: Auto-tuning strategies for exploring the optimization space. From left to right: exhaustive search, heuristically-pruned search, iterative hill-climbing (iterative greedy) and gradient descent search. The red circle represents the optimal parameter combination and the yellow rectangle the best option found for each strategy. The blue regions depict the parametrical space explored and the arrows the direction chosen for the exploration.

When the optimization space to explore is tractable (small N^D), an exhaustive search is usually employed, exploring the whole optimization space and guaranteeing the global minima. However, since the number of algorithmic optimizations and their parametrical space can easily grow in the order of hundreds, this search is no longer feasible in terms of time and resources. To alleviate the combinatorial explosion of the space size to traverse, a number of search strategies have emerged avoiding expensive exhaustive searches. Figure 5.5 shows the most representatives strategies used in auto-tuners for searching the best combination of parameters: exhaustive, heuristic, iterative hill-climbing and gradient descent search. In this figure, each axis plot represents an algorithmic optimization and their range of possible values, whereas the curves denote areas where performance stays constant. Although, performance curves are smooth, the optimization problems tackled by stencil auto-tuners lay on a discrete space where a multi-dimensional integer space must be traversed. The heuristic search tries to minimize the search time by applying heuristics that prune the full space area by deciding appropriate parameters for certain optimizations. As a drawback, the best performance might not be found (red dot in plots) given that the applied heuristics traverse a subspace (blue regions) that only allows to find a pseudo-optimal performance (yellow rectangle). On the other hand, the iterative hill-climbing search reduces the initial N^D space into a $k \times N \times D$ space. In order to achieve it, this strategy performs an iterative greedy search that seeks the local minima for an algorithmic optimization (green diamonds) while all other variables are kept fixed. Then, this process is repeated for the remaining algorithmic optimizations using the previous pseudo-optimal parameters. This search is conducted several times (k) until no further improvement is obtained (yellow rectangle). Finally, the gradient descent strategy diminishes the multi-dimensional search space significantly by testing the surrounding area of an initial seed and moving towards the greatest gradient direction (maximal slope) that minimizes the execution time. This operation is repeatedly performed until no descent gradient is found. The performance of this strategy is specially sensitive to the initial seed, not only in terms of search time, but also in pseudo-optimal results if local minima are present.

The auto-tuning search performed in our domain decomposition strategy has one main advantage, it only considers spatial-blocking algorithm as algorithmic optimization. This decision reduces significantly the algorithmic optimization combinations to only three parameters (TI, TJ and TK), and to a containable parametrical space ($TI = \{1 \ldots NZ\}$, $TJ = \{1 \ldots NX\}$ and $TK = \{1 \ldots NY\}$).

In addition, our auto-tuning strategy also strives to minimize the search time. First, it uses an heuristically-pruned search over TI and TK spatial-blocking parameters. In order to achieve it, we take advantage of previous research works [43, 72] stating that spatial-blocking algorithms benefit from leaving unit- and least-stride dimensions uncut ($TI = NZ$ and $TK = NY$). The main reason is that the prefetching effect is not disrupted and data reuse is promoted for least-stride dimension which is more expensive latency-wise (see Section 4.2). Second, we finally obtain a pseudo-optimal parameter by conducting a straightforward gradient descent search on TJ parameter. Algorithm 13 shows the pseudo-code of our gradient search which must be called by passing the maximum NX value ($dimx$) and the stencil kernel with spatial-blocking to be tuned ($kernel$). As an initial point for the gradient descent search, three possible seeds are tested: a lower bound (2), centered bound ($dimx/2$) and an upper bound ($dimx$). The seed that minimizes the execution time is used. Then, the traversing direction is chosen by benchmarking neighbour parameters at prefixed offsets. The neighbour parameter that minimizes the execution time is chosen as the next candidate for the following gradient iteration. The offset array must include enough distant parameters to avoid falling in a local minima that would prevent obtaining the optimal solution. The algorithm also includes dynamic programming to avoid benchmarking the parameter combinations more than once.

5.5 Experimental Results

In order to corroborate our statements and evaluate the decomposition strategies, we set a group of tests with different problem sizes. Their ranges are defined from relatively small to very large dimensions. The intent is to reproduce the behavior that appears in strong

Algorithm 13 Pseudo-code of the auto-tuner used in our tests for finding a pseudo-optimal TJ parameter. This auto-tuner performs a gradient descent search traversing the linear search space depending on the slope. Benchmark-Kernel routine performs the time benchmarking of $kernel$ routine by performing a number of fixed trials.

```
1: function Auto-tuner($dimx$, $kernel$)
                                                  ▷ Initial seeds for $TJ$ parameter
2:     $TJ_{upper} \leftarrow dimx$
3:     $TJ_{middle} \leftarrow \lfloor dimx/2 \rfloor$
4:     $TJ_{lower} \leftarrow 2$
                                                  ▷ Search best initial seed
5:     Benchmark-Kernel($TJ_{upper}$, $time[TJ_{upper}]$, $visit[TJ_{upper}]$, $kernel$)
6:     Benchmark-Kernel($TJ_{middle}$, $time[TJ_{middle}]$, $visit[TJ_{middle}]$, $kernel$)
7:     Benchmark-Kernel($TJ_{lower}$, $time[TJ_{lower}]$, $visit[TJ_{lower}]$, $kernel$)
                                                  ▷ Select best initial seed
8:     $TJ \leftarrow TJ_{lower}$
9:     if $time[TJ_{middle}] < time[TJ]$ then
10:        $TJ \leftarrow TJ_{middle}$
11:    end if
12:    if $time[TJ_{upper}] < time[TJ]$ then
13:        $TJ \leftarrow TJ_{upper}$
14:    end if
                                                  ▷ Offset for neighbor points to visit
15:    $OFFSET = \{-64, -32, -16, -8, -4, -2, +2, +4, +8, +16, +32, +64\}$
                                                  ▷ Search for a pseudo-optimal $TJ$ (local minima)
16:    $changed \leftarrow$ TRUE
17:    while $changed =$ TRUE $\wedge$ $iter < MAX_{iters}$ do
18:        for $i \leftarrow 0$ to $N_{neigh}$ do
19:            $changed \leftarrow$ FALSE
                                                  ▷ Neighbor to visit
20:            $neigh \leftarrow \max(\min(TJ + OFFSET[i], TJ_{upper}), TJ_{lower})$
21:            if $visit[neigh] =$ FALSE then
22:                Benchmark-Kernel($neigh$, $time[neigh]$, $visit[neigh]$, $kernel$)
23:            end if
24:            if $time[neigh] < time[TJ]$ then
25:                $TJ \leftarrow neigh$
26:                $changed \leftarrow$ TRUE
27:            end if
28:        end for
29:        $iter \leftarrow iter + 1$
30:    end while
31:    return $TJ$
32: end function
```

scalability executions where threads stagnate as we increase the computational nodes. To this extent, the least-stride dimension is varied while the remaining dimensions are kept fixed. In addition, all dimensions except the unit-stride have been defined as uneven in order to emphasize the imbalance across domains. Two modern platforms have been used, a dual-socket node with Intel Xeon Sandy Bridge E5-2670 processors and a many-core Intel Xeon Phi with 60 cores. Table 5.2 shows a summary of the testbed for the experimental results. The stencil used in this benchmark is an ADR kernel (See Appendix A.3) which has been fully SIMDized.

Parameters	Range of values
Intra-node decompositions	static (X and Y), balanced and guided
Problem sizes ($I \times J \times K$)	$64 \times 601 \times 25$, $64 \times 601 \times 75$,
	$64 \times 601 \times 151$, $64 \times 601 \times 301$,
	$64 \times 601 \times 601$, $64 \times 1201 \times 1201$
Stencil kernel	ADR kernel† in single-precision (13-point stencil)
Platforms	2 $\times$ Intel Xeon Sandy Bridge E5-2670 system
	and Intel Xeon Phi 5110P (KNC)

Table 5.2: List of parameters used for testing the decomposition strategies. †See Appendix A.3 for additional info.

The experimental results are exposed in three ways: maximum attainable performance, break-down of scalability and 2D decomposition view. Table 5.3 and 5.4 show a summary of the maximum attainable scalability for each platform, problem size and decomposition strategy. These results include the combination of the decomposition scheduler with the auto-tuner. For Intel Xeon Phi, it also includes the SMT affinity exposed in Section 5.2, where 4 SMT threads cooperate in a plane distribution policy. On the other hand, the Intel Sandy Bridge processor used does not have the SMT enabled. Nevertheless, we have taken advantage of the Semi-stencil algorithm which can leverage the performance by reducing cache pressure promoting data reuse. In addition, we have also included the β metric and the maximum imbalance in percentage across computational domains.

Size	Static X			Static Y			Balanced			Guided		
($J \times K$)	SCA	β	IMB	SCA	β	IMB	SCA	β	IMB	SCA	β	IMB
601×75	104.4	.66	10%	88.8	.51	.5%	113.4	.72	6%	90.0	.51	.5%
601×151	112.5	.67	10%	91.7	.56	19%	117.1	.76	3%	92.5	.56	19%
601×301	117.3	.68	10%	96.1	.56	20%	123.0	.81	3%	94.6	.56	20%
601×601	114.5	.69	10%	112.2	.69	10%	121.9	.85	2.3%	112.6	.69	10%

Table 5.3: Domain decomposition results for Intel Xeon Phi (MIC) architecture. Columns shown are scalability (SCA), β parameter and imbalance (IMB) for 240 threads (60 cores $\times$ 4 SMT). Only best result of each strategy is shown (Decomposition+Auto-tuning+SMT Affinity).

Results are quite compelling, specially for Intel Xeon Phi where the *balanced* strategy overcomes the remaining ones. In general, its imbalance is sustained at a very low ratio (2.3% to 6%), which allows a good equilibrium between the 60 cores. However, despite of having a higher imbalance in $64 \times 601 \times 75$ case compared to *static Y* and *guided* strategies (6% vs 0.5%), its performance is still higher thanks to the β metric, which remains only to 0.51 with *static Y* and *guided* and 0.72 with *balanced* scheduler. This metric reveals that at least 49% of the data involved in the stencil computation with *static Y* and *guided* are halo points, whereas halo represent 38% with *balanced*. This difference leads to a reduction of the memory traffic and the overall time. Figure 5.6 shows a 2D view (X-Y plane where Z is left uncut) of the domain distribution for this case. As it can be seen, *static Y* and *guided* schedulers produce very narrow domains that confer a low β metric. This is not the case of *balanced* decomposition, where, although having a higher imbalance compared to the remaining ones (up to 6%), its equilibrated domains ensure a higher β. The more cubical a domain is, the higher the β metric is. Actually, there is a trade-off between domain balance and β metric, where the preferred decomposition should include an imbalance $\approx$ 0% and a $\beta \approx 1$.

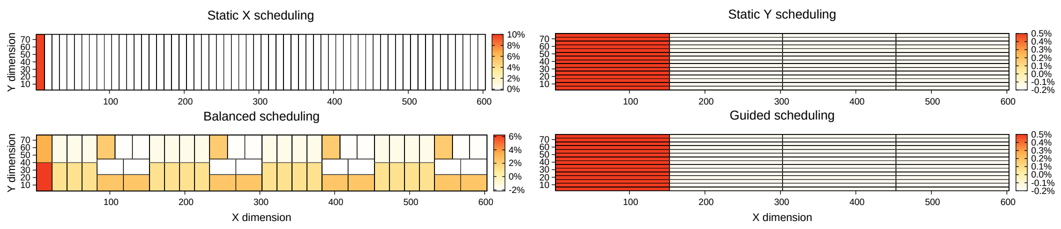


Figure 5.6: Imbalance of decomposition strategies for Intel Xeon Phi with a $64 \times 601 \times 75$ problem size. Distribution with 60 cores (60 domains) and 4-way SMT (240 threads).

Figure 5.7 shows the 2D distribution view for $64 \times 601 \times 601$ case with two opposite schedulers: *balanced* and *static Y*. Results are clarifying, the *balanced* strategy decomposes the computational domain striving for not only reducing the imbalance across domains, but also for a cubical distribution. The improvement of these two metrics leads clearly to a better performance.

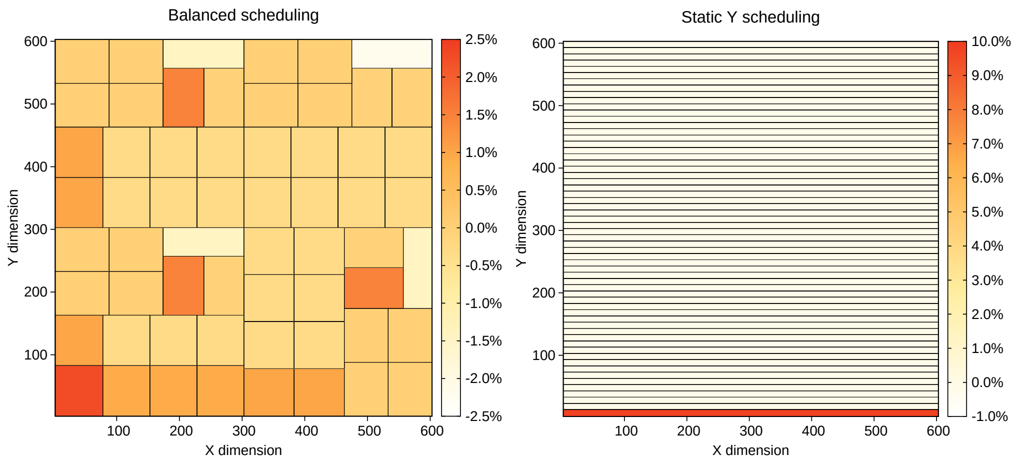


Figure 5.7: Imbalance of *balanced* and *static Y* schedulings for Intel Xeon Phi with a $64 \times 601 \times 601$ problem size. Distribution with 60 cores (60 domains) and 4-way SMT (240 threads).

Reviewing the Intel Sandy Bridge results in Table 5.4, a totally different scenario is observed. Although the *balanced* scheduler behaves slightly better, there is not a clear winner for large sizes. Indeed, the low thread level parallelism of our Intel Sandy Bridge processor does not pose a real challenge for *balanced* scheduling. As a consequence, all schedulers produce fair scalability results ($\approx 15\times$) with good β and imbalance metrics. For instance, Figure 5.8 depicts the 2D distribution view for a large case with *balanced* and *guided* distributions. In both cases, imbalance remains low (1.0% and 1.3%) and β close to 1 (0.92 and 0.91). However, as the problem size is decreased, two strategies prevail, *static X* and *balanced*. Actually, *static X* performs satisfactorily well due to the large and fixed X dimension used in all test cases. On the other hand, *balanced* strategy should guarantee an outstanding decomposition in terms of performance regardless the decomposition strategy at inter-node level.

Size	Static X			Static Y			Balanced			Guided		
($J \times K$)	SCA	β	IMB	SCA	β	IMB	SCA	β	IMB	SCA	β	IMB
601×25	15.2	.69	5.2%	11.9	.31	36%	15.5	.69	5.2%	11.9	.31	36%
601×75	14.2	.85	2.5%	12.2	.47	15%	14.2	.85	2.5%	13.4	.66	6.8%
601×151	14.5	.85	1.8%	12.6	.67	5.9%	14.8	.85	1.8%	14.7	.85	1.8%
601×601	15.1	.88	1.3%	14.5	.84	1.5%	15.2	.89	1.0%	14.9	.88	1.3%
1201×1201	15.4	.91	.7%	14.8	.89	1.2%	15.4	.92	.5%	15.0	.91	.5%

Table 5.4: Domain decomposition results for Intel Sandy Bridge platform. Columns are scalability (SCA), β parameter and imbalance (IMB) for 16 threads (2 sockets $\times$ 8 cores). Only the best result of each strategy is shown (Decomposition+Auto-tuning+Semi-stencil).

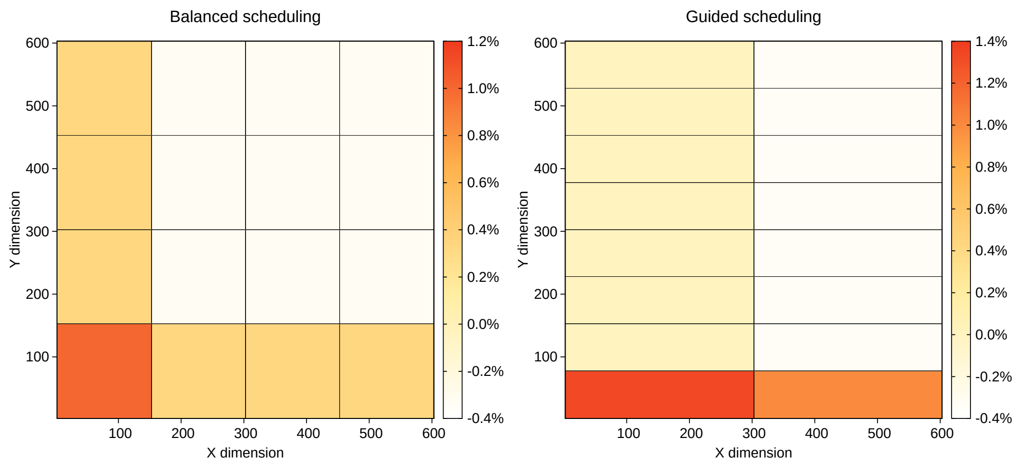


Figure 5.8: Imbalance of *balanced* and *guided* schedulings for Intel Sandy Bridge-EP platform with a $64 \times 601 \times 601$ problem size. Distribution for dual-socket node with 2×8 cores (16 domains).

Finally, we expose the scalability results for each particular layer of the schedulers: domain decomposition and auto-tuning, including SMT affinity for Intel Xeon Phi and Semi-stencil for Intel Sandy Bridge. Results are broken down to show separately the benefit of each optimization technique. The intent is to show not only the key fact of having a wise decomposition strategy, but also its combination with an auto-tuning and SMT affinity approaches in order to leverage scalability. Figure 5.9 plots the scalability break-down for Intel Xeon Phi on two problem sizes, small and medium. The number of threads per core is progressively increased until the processor is fully populated. As threads are increased, the decomposition strategies alone are not able to attain a sustainable scalability. In fact, in most cases, the performance drops when more than 2 threads are pinned per core (>120 threads). Nevertheless, when the auto-tuning step is used, this trend disappears and the best scalability results scale up to +90×. The search of a tuned TJ parameter for the spatial-blocking algorithm is crucial to reduce pressure and cacheline conflicts in shared L1 and L2 caches. In addition, the SMT affinity approach (on 120 threads executions upwards) can produce further improvements by reducing memory footprint and prefetcher contention. For instance, on $64 \times 601 \times 300$ case, the SMT affinity yields an improvement from 100× to 123× with the *balanced* scheduler.

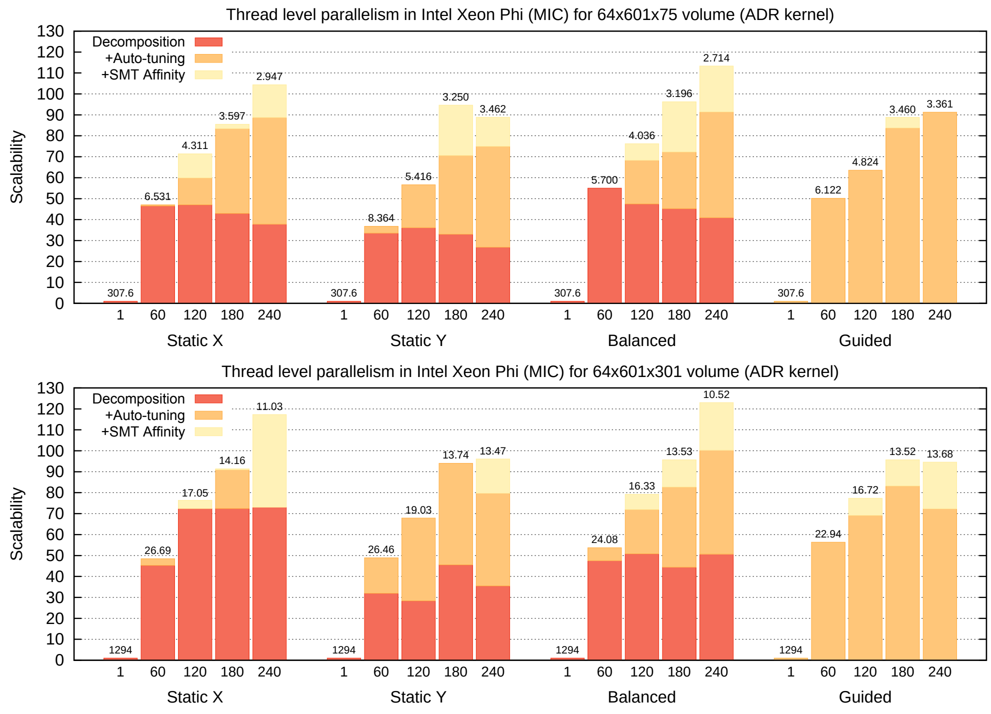


Figure 5.9: Scalability break-down for Intel Xeon Phi (MIC). The numbers shown on top of each bar represent the execution time (in milliseconds) for the best configuration.

The Intel Sandy Bridge results in Figure 5.10 show a different scenario when compared to Intel Xeon Phi. First, all decomposition strategies alone yield a reasonable scalability as threads are increased until both processors are fully populated (≈13.8×). The private L1 and L2 caches, only shared by one thread, and their aggressive prefetchers cope successfully with the scalability of the ADR kernel. Second, the auto-tuning of TJ parameter leads to a certain improvement, but not as remarkable as on Intel Xeon Phi. Indeed, the tremendous shared L3 cache of 20 MB per processor diminishes the spatial-blocking effect. Additionally, this performance loss is particularly exacerbated in *static X* and *balanced* schedulings, where an implicit tiling is already applied due to the specific decomposition along the X axis. As a final optimization, the Semi-stencil algorithm was added on top of the classical ADR kernel. This novel algorithm could partially replace the lack of SMT feature by improving data reuse in private L1 and L2 caches. For example, Semi-stencil led to a nearly perfect scalability on $64 \times 601 \times 601$ case enhancing the results from 14 to 15.2$\times$.

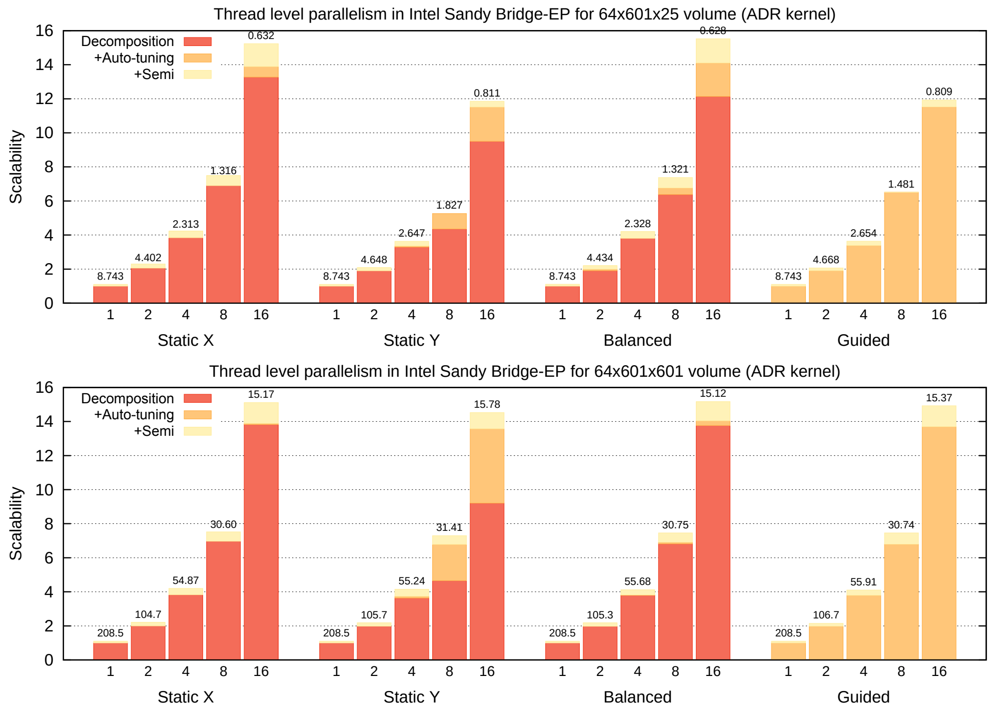


Figure 5.10: Scalability break-down for Intel Sandy Bridge. The numbers shown on top of each bar represent the execution time (in milliseconds) for the best configuration.

5.6 Summary

This chapter has proposed a set of intra-node optimizations for stencil computations pursuing the CMP trend, where many-core processors with complex hierarchical caches coexist in multi-socket platforms. The intra-node strategies discussed cover the following areas: SMT affinity, multi-core decomposition and auto-tuning. Through the combination of these three strategies, we have achieved better scalability results.

So far, auto-tuning has shown to be very convenient to select pseudo-optimal parameters for specific algorithms. Nevertheless, the search can be cumbersome when the parametrical space to explore is huge due to the complex combination of optimizations. In order to ease the search of pseudo-optimal parameters, performance model can help giving hints almost instantly. Next chapter is devoted to this topic by proposing a multi-level cache performance model for stencils that remains flexible and accurate.

Chapter 6

Performance Modeling of Stencil Computations

As shown in Chapter 2, the stencil computations are the core of many Scientific Computing applications, of which moving data back and forth from memory to CPU is a major concern. Therefore, the development and analysis of source code modifications that can effectively use the memory hierarchy of modern architectures is crucial. This is not a simple task, actually it is lengthy and tedious. The large number of stencil optimization combinations, which might consume days of computing time, makes the process lengthy. Besides, the process is tedious due to the slightly different versions of code to be implemented and assessed.

As an alternative, models that predict performance can be built without any actual stencil computation execution, thus reducing the above mentioned cumbersome optimization task. This performance models help expose bottlenecks and predict suitable tuning parameters in order to boost stencil performance on any given platform. Previous works have proposed cache misses and execution time models for specific stencil optimizations. However, most of them have been designed for low-order stencils (7-point) and simplified cache-hierarchies, leaving aside considerations of modern HPC architectures. Nowadays, the following two considerations need to be accurately modeled. First, the complex architectures that sport multi- or many-core processors with shared multi-level caches featuring one or several prefetching engines. Second, the algorithmic optimizations, such as spatial blocking or Semi-stencil, which have complex behaviors that follow the intricacy of the above described architectures. The challenge is to cover all these features to effectively model stencil computation performance.

In this chapter, we propose a flexible and accurate model for a wide range of stencil sizes that captures the behavior of 3D stencil computations using the platform parameters. The model has been tested in a group of representative hardware architectures, using realistic dataset sizes. The accuracy results obtained show that the proposed model is reliable and can help to speed up the stencil computation optimization process. To that end, this model has been designed to be extended adding new stencil optimization techniques, thus providing essentially a framework that can cover most of the state-of-the-art in stencil optimizations.

6.1 Performance Modeling Overview

The main objective of performance models is to improve stencil-based applications performance. High performance is usually achieved by algorithmic changes that lead to suboptimal implementations, where many executions of these different implementations are required. Notice that these code modifications must not taint the numerical soundness of the numerical algorithm. To this extent, we define a *test case* as a set of characteristics that include platform and stencil parameters, such as: stencil size, dataset size, algorithm implementation or architecture configuration (see Figure 6.1). Since every one of those parameters may change among executions, the number of total test cases can be extremely large. Therefore, it is not feasible to find a possible suboptimal stencil scheme by manual means. The automatization of this task is a must.

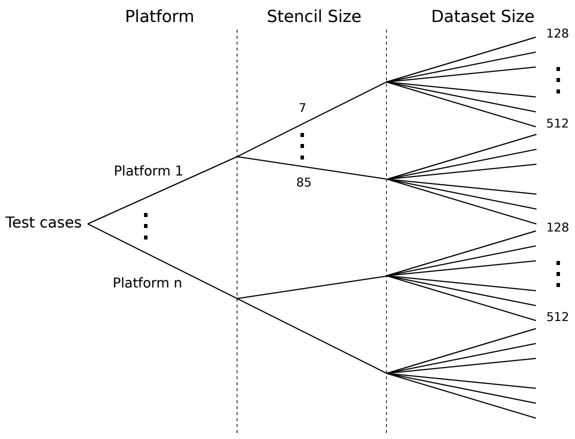


Figure 6.1: *Test cases* tree, this one with a basic configuration based on three main factors: platform, stencil size and dataset size.

We identify three ways of improving stencil performance: by brute force, *i.e.* trying every possible test case as in auto-tuning, by modeling the algorithm behavior and by a hybrid approach based on the two previous ideas. The manual trial-and-error approach turns the process of optimizing codes lengthy and tedious. The large number of stencil optimization combinations, which might consume days of computing time, makes the process lengthy. Furthermore, the process is tedious due to the slightly different versions of code that must be implemented and assessed. To alleviate the cumbersome optimization process from user supervision, several auto-tuning frameworks [16, 22, 41] have been developed. These frameworks automatize the search by using heuristics to guide in the pruning of the parameter subspace.

As an alternative, models that predict performance can be built without the requirement of any actual stencil computation execution. We consider that the model-based approach has three advantages with respect to auto-tuning. First, once the model is validated, there is no need for real executions, which can reduce hours or days of testing time. Second, there is no need of developing a framework to command the trial-and-error experiments. And third, which is the most important, the model flexibility allows the extension without a substantial effort in implementation and experimental executions.

In addition, performance modeling is highly time-cost effective compared to other modeling approaches based on regression analysis. In regression-based analysis, users are required to conduct extensive and costly experiments to obtain the input data for regression. A wide range of hardware performance counters are gathered and machine learning algorithms are used to determine correlations between architectural events and compiler optimizations. The more complex the model is, the more data is required to estimate the correlation coefficients. Moreover, regression models do not provide neither cache miss predictors nor hints about algorithmic parameter candidates (e.g. spatial blocking). Nevertheless, regression analysis can be partially useful whether it is intended to give indications of possible performance bottlenecks and it is combined with knowledge-based systems.

Unfortunately, modeling has also shortcomings. The performance characterization of a kernel code is not trivial and relies heavily on the ability to capture the algorithm's behavior in an accurate fashion, independently of the platform and the execution environment. In order to do so, the estimation of memory latencies is critical in memory-bound kernels. This is why predicting accurately 3C (*compulsory*, *conflict* and *capacity*) misses play an important role to characterize effectively the kernel performance. Another drawback is the limited coverage of the experimental space regarding very fine grain test cases. As a solution, the hybrid approach can be used. In this approach, models are integrated into auto-tuning frameworks for compile and run-time optimizations; making guided decisions about the best algorithmic parameters and suggesting code modifications. In the *test case* tree of this hybrid approach, the optimization search of main branches may be covered by modeling, whereas the parameter tuning of tree leaves may be performed by the auto-tuning mechanism.

6.2 State of the Art

The performance modeling topic on stencil computations has been fairly studied in the recent years. The pioneers Kamil and Datta [21, 43] proposed straightforward cost models to capture the performance of a set of stencil optimizations. Such models were strongly based on the Stanza Triad micro-benchmark (STriad), derived from the STREAM Triad benchmark [54]. Using their approach, a 7-point naive stencil computation was modeled by taking into account three types of memory access costs in a flat memory hierarchy: *first*, *intermediate* and *stream*. In this model, the access cost of the first non-streamed cache line is represented by C_{first}, given the overhead of performing a STriad operation with a short length. On the other hand, the cost of streamed cache lines, C_{stream}, is computed by performing a STriad with a large vector length in order to trigger the stream access. Nevertheless, due to the *warm-up* phase of the prefetching engines, stream accesses are only activated after a number of misses (denoted as k) with a cost of $C_{intermediate}$.

Therefore, considering a domain of L elements (being $L = N^3$ and N the axis domain size) and W elements per cache line, where $\lceil (L/W) \rceil$ cache lines must be fetched to compute the entire domain, the total cost $C_{stencil}$ of a low-order stencil can be summarized as:

$$C_{stencil} = C_{first} + k * C_{intermediate} + (\lceil (L/W) \rceil - k - 1) * C_{stream} \ . \tag{6.1}$$

In addition, Kamil and Datta also developed a cache blocking model for Rivera tiling [72], where an N^3 problem is traversed using $I \times J \times N$ blocks, being I and J the most unit-stride dimensions of the cut. This model was built on the top of the Equation 6.1. For this purpose, they devised a simple approach by setting lower and upper cost bounds of memory traffic for $I \times J \times N$ blocks in a 7-point stencil computation. The lower bound assumes high cache reuse with only compulsory misses ($2C_{stencil}$, one plane to read and one plane to write), while the upper bound considers no cache reuse at all leading to conflict and capacity misses ($4C_{stencil}$, three planes to read and one plane to write).

Therefore, given an N^3 grid problem, the number of non-streamed (T_{first}), partially streamed ($T_{intermediate}$) and streamed ($T_{streamed}$) cache lines fetched are evaluated differently due to the disruption of the prefetching effect as: $T_{first} = \frac{N^3}{I}$ if $I \neq N$, or $\frac{N^3}{IJ}$ if $I = N \neq J$, or $\frac{N^2}{IJ}$ if $I = J = N$, $T_{intermediate} = T_{first} * (k-1)$ and $T_{stream} = T_{total} - T_{intermediate} - T_{first}$, where $T_{total} = \frac{\lceil (I/W) \rceil N^3}{I}$. Consequently, the cost of sweeping a 3D domain in a blocked fashion is approximated as:

$$C_{stencil} = C_{first} T_{first} + C_{intermediate} T_{intermediate} + C_{stream} T_{stream} \ . \tag{6.2}$$

Lately, Datta *et al.* [21] extended their model to time-skewing stencil computations predicting partially the entire stencil running time. In order to do so, they distinguished mainly five cases of cache misses: three preferred cases with compulsory misses, and other two cases with capacity misses. Finally, conflict misses where incorporated to the model by using a cumulative Gaussian distribution that matched the data from a simple microbenchmark.

Kamil, Datta *et al.* set the basis for modeling stencil computations with the approach of streamed and non-streamed data for Rivera blocking. However, there are at least two issues which were not considered in their works. First, they were taking into account only low-order 3D stencil computations (7-point or 27-point in a compact fashion). Nevertheless, in many scientific computing applications, high-order stencils are required, from 25-point star-like stencils [9] upwards [12, 69]. Second, the memory model is approached as an unified homogeneous hierarchy, leaving aside multi-level cache peculiarities of modern architectures. Furthermore, the number of available registers in the ALU or the different cost between load and store operations are not considered in their model. We do believe that all these aspects must be taken into account in order to lay down a flexible, accurate and reliable model suitable for any kind of stencil computation and architecture. However, as far as we know, no stencil computation model with such characteristics can be found in the literature, which motived us to develop the current work.

Regression analysis has also shown some appeal for modeling stencil computations. Rahman *et al.* [71] developed a set of formulas via regression analysis to model the overall performance in 7 and 27-point Jacobi and Gauss-Seidel computations. Their intent was not to predict absolute execution time but to extract meaningful insights that might help developers to improve effectively their codes. The time-skewing technique has been also modelized by Strzodka *et al.* [78]. They proposed a performance model for their cache accurate time skewing (CATS) algorithm, where the system and the cache bandwidths were estimated using regression analysis. The CATS performance model considered only two levels of memory hierarchy, and therefore it could be inaccurate on some HPC architectures. Their aim was to find out which hardware improvements were required in single-core architectures to match the performance of future multi-core systems. The main disadvantage of regression analysis techniques is that they are based on an extensive collection of empirical data which must be gathered for each configuration (stencil type, problem size and architecture). This exhaustive number of executions makes regression analysis an infeasible technique to obtain immediate results.

Likewise, performance modeling has been successfully deployed into other numerical areas such as sparse matrix vector multiplications [62] and generic performance models for bandwidth-limited loop kernels [40, 81, 82].

6.3 Multi-Level Cache Performance Model

In this section, we present the basis for a flexible and accurate model suitable for a wide range of stencil computations and architectures. As mentioned before, the intricacy of the multi-level cache hierarchy has to be considered to predict accurately the behavior of stencil computations in modern architectures. This work is actually composed of two main parts, an initial research where the basis of the model were settled, and a second research stage where some concepts of the initial model were clearly improved and extended to fulfill the coverage of the complex architectures.

It is a widely accepted fact that stencil computations are usually dominated by the data transfer latency [21, 29]. Therefore, this performance model considers stencil computations as memory-bound, where the cost of computing the floating-point operations is assumed negligible due to the overlap with considerable memory transfers. This assumption is especially true for large domain problems where, apart from compulsory misses, capacity and conflict misses arise commonly leading to a low OI [25]. Likewise, interferences between data and instructions at cache level are not taken into account.

This section proceeds as follows: firstly, we review the base model. Secondly, the number of read and write misses for stencil computations are approximated. And finally, some peculiarities, like prefetching and other algorithmic optimizations, such blocking or Semi-stencil, are added to the model.

Algorithm 14 The classical stencil algorithm pseudo-code. II, JJ, KK are the dimensions of the data set including ghost points. ℓ denotes the neighbors used for the central point contribution. $C_{Z1\ldots Z\ell}$, $C_{X1\ldots X\ell}$, $C_{Y1\ldots Y\ell}$ are the spatial discretization coefficients for each direction and C_0 for the self-contribution. Notice that the coefficients are considered symmetric and constant for each axis.

```
1: for $t = 0$ to $timesteps$ do                                   ▷ Iterate in time
2:     for $k = \ell$ to $KK - \ell$ do                                    ▷ Y axis
3:         for $j = \ell$ to $JJ - \ell$ do                                ▷ X axis
4:             for $i = \ell$ to $II - \ell$ do                            ▷ Z axis
5:                 $\mathcal{X}^{t}_{i,j,k} = C_0 * \mathcal{X}^{t-1}_{i,j,k}$
                       $+ C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k})$
                       $+ C_{X1} * (\mathcal{X}^{t-1}_{i,j-1,k} + \mathcal{X}^{t-1}_{i,j+1,k}) + \ldots + C_{X\ell} * (\mathcal{X}^{t-1}_{i,j-\ell,k} + \mathcal{X}^{t-1}_{i,j+\ell,k})$
                       $+ C_{Y1} * (\mathcal{X}^{t-1}_{i,j,k-1} + \mathcal{X}^{t-1}_{i,j,k+1}) + \ldots + C_{Y\ell} * (\mathcal{X}^{t-1}_{i,j,k-\ell} + \mathcal{X}^{t-1}_{i,j,k+\ell})$
6:             end for
7:         end for
8:     end for
9: end for
```

6.3.1 Base Model

Considering a problem size of $I \times J \times K$ points of order ℓ, where I is the unit-stride (Z axis) and J and K the least-stride dimensions (X and Y axes), an amount of $P_{read} = 2 \times \ell + 1$ and $P_{write} = 1$ Z-X planes of $\mathcal{X}^{t-1}$ are required to compute a single $\mathcal{X}^{t}$ plane (see Algorithm 14). Thus, the total data to be held in memory to compute one k iteration of the sweep is

$$S_{total} = P_{read} \times S_{read} + P_{write} \times S_{write}, \tag{6.3}$$

being $S_{read} = II \times JJ$ and $S_{write} = I \times J$ their size in words. Notice that II, JJ and KK are the domain problem dimensions including the ghost points ($2 \times \ell$).

Likewise, the whole execution time (T_{total}) on an architecture with n levels of cache is estimated based on the aggregated cost of transferring data on three memory hierarchy groups: *first* (T_{L1}), *intermediate* (T_{L2} to T_{Ln}) and *last* (T_{Memory}),

$$T_{total} = T_{L1} + \cdots + T_{Li} + \cdots + T_{Ln} + T_{Memory} \ . \tag{6.4}$$

Each transferring cost depends on their hits and misses and is computed differently for each group. But, in general, the transferring cost ($T_{Li} = Hits^{data}_{Li} \times T^{data}_{Li}$) is based on the latency of bringing as much data (word or cacheline) as required ($Hits^{data}_{Li} = Misses^{data}_{Li-1} - Misses^{data}_{Li}$) from the cache level to the CPU ($T^{data}_{Li} = data/Bw^{read}_{Li}$) in order to compute the stencil. Finally, the amount of misses issued at each cache level is estimated as

$$Misses_{Li} = \lceil II/W \rceil \times JJ \times KK \times nplanes_{Li} \ , \tag{6.5}$$

where $W = cacheline/word$ is the number of words per cacheline, and $nplanes_{Li}$ is the number of $II \times JJ$ planes read from the next cache level $(Li+1)$ for each k iteration due to possible compulsory, conflict or capacity misses.

However, the transferring costs of *first* and *last* memory hierarchy groups are computed in a slightly different way. For example, when the CPU issues a $word$ load or store instruction, the data is brought from the closer cache level $(L1)$. If the data is not present, a miss is flagged and passed to the next level of the hierarchy. Recall that a stencil computation requires several values in the inner loop to compute a single point: grid points $(\mathcal{X}^{t-1})$, weights $(C_{Z,X,Y,0})$ and indices (i,j,k) to access grid points. In the CPU register bank, grid points and weights are kept in Floating-Point Registers (FPR), while indices use General-Purpose Registers (GPR). Leaving aside compiler optimizations, grid points must be fetched in every sweep of the loop, whereas weights and indices might be partially reused. However, depending on the order of the stencil (ℓ), the dimension of the problem (dim) and the available registers $(FPR_{free}$ and $GPR_{free})$, data reuse becomes burdensome, leading to register spilling and a higher CPU-$L1$ traffic.

Two different ways were devised to estimate the data brought directly from the *first* level $(Hits_{L1}^{word})$; an initial method based on register pressure prediction, and a more accurate method based on static analysis of the compiler generated object code:

Register pressure predictor: After a deep analysis of the assembly stencil code on our testbed architectures, a couple of assertions could be pointed out. First, the required $\ell \times dim + 1$ weights were reused along the sweep if enough FPR_{free} were available. Second, the compiler kept one index register for each grid point, except for those points along the unit-stride dimension (Z axis) that shared the same index $(2 \times \ell \times (dim-1)+1)$. Likewise, index registers were also reused whether enough GPR_{free} resources existed. Using this assertions, we proceeded to estimate the register pressure $(Reg_{\{grid,weight,index\}})$ and the hits $(Hits_{L1}^{word})$ to compute the whole domain $(I \times J \times K)$ as,

$$\begin{aligned}
Reg_{grid} &= 2 \times \ell \times dim + 1 \\
Reg_{index} &= \max((2 \times \ell \times (dim-1)+1) - GPR_{free}, 0) \\
Reg_{weight} &= \max((\ell \times dim + 1) - FPR_{free}, 0) \\
Hits_{L1}^{word} &= (Reg_{grid} + Reg_{weight} + Reg_{index}) \times I \times J \times K - Misses_{L1}^{word}
\end{aligned} \tag{6.6}$$

where FPR_{free} and GPR_{free} were speculated using the specifications of the architecture and the register management policy of the compiler. Nevertheless, this method poses many difficulties to predict accurately the $L1$ accesses due to the extra information about the register usage, making this strategy an infeasible option.

Static code analysis: In this method, the register pressure was directly estimated by counting the number of loads issued to the memory hierarchy through static analysis of the compiler generated object. This value remains constant for a single iteration of the inner loop and can be deterministically computed with binary analysis tools that determine the number

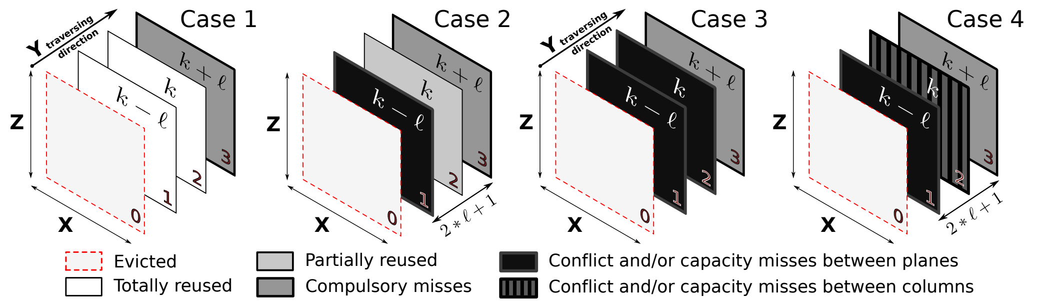


Figure 6.2: Cases considered for cache misses during the k sweep of the stencil computation. Each plane color represents the misses generated when accessing the plane. A light color means none or few misses and a dark color implies a high ratio of misses.

of loads issued in a section of code. The main advantage of this method is that it collects precise information and must be conducted only once per binary code, being an option not prone to errors.

Finally, the number of accesses to the *last* level can be computed as $Hits_{Memory}^{cacheline} = Misses_{Ln}^{cacheline}$ given that data is always allocated in the *last* hierarchy level. Therefore, any miss issued by the previous cache hierarchy level (Ln) will generate necessarily a hit in the main memory.

6.3.2 Cache Miss Cases and Rules

The correct estimation of $nplanes_{Li}$ is crucial for the model accuracy. To do so, four miss cases (C_1, C_2, C_3 and C_4, ordered from lower to higher penalty) and four rules (R_1, R_2, R_3 and R_4) are devised. Each of these rules triggers the transition from one miss case scenario (see Figure 6.2) to the next one. In this model, the rules are linked together, and therefore triggered in sequential order, thus exposing different levels of miss penalty.

Rule 1 (R_1): The best possible scenario (lower bound) is likely to happen when all the required Z-X planes (S_{total}) to compute one k iteration fit loosely (R_{col} factor) into the cache level ($size_{Li}$). This yields to only compulsory misses and to the following rule, $R_1 : ((size_{Li}/w) \times R_{col} \geq S_{total})$.

Rule 2 (R_2): Conversely to R_1, when all the required planes do not fit loosely in cache, except the k-central plane with a higher temporal reuse (less chance to be evicted from cache), conflict misses are produced among planes. This scenario is likely to happen when the following rule is true, $R_2 : ((size_{Li}/w) > S_{total})$.

Rule 3 (R_3): On a third possible scenario, it is assumed that despite the whole data set does not fit in cache (S_{total}), the k-central plane does not overwhelm a significant part of the cache (R_{col} factor). Therefore, the possibility of temporal reuse is reduced compared to R_2 but not canceled completely. This scenario can occur when, $R_3 : ((size_{Li}/w) \times R_{col} > S_{read})$.

Rule 4 (R_4): The worst scenario (upper bound) appears when neither the planes nor the columns of the k-central plane fit loosely in the cache level. Then, capacity and conflict misses arise frequently, resulting as well in fetching the k-central plane at each j iteration of the loop. This scenario gives the following rule, $R_4 : ((size_{Li}/w) \times R_{col} < P_{read} \times II)$.

w is the word size (in single or double precision), and R_{col} is a factor proportional to the required data by the k-central plane with respect to the whole dataset ($P_{read}/2P_{read} - 1$). Putting all the ingredients together, the computation of $nplanes_{Li}$ is yielded by the following conditional equations:

$$nplanes_{Li}(II, JJ) = \begin{cases} C_1 : 1, & \text{if } R_1 \\ C_1 \sqcup C_2 : (1, P_{read} - 1], & \text{if } \neg R_1 \wedge R_2 \\ C_2 \sqcup C_3 : (P_{read} - 1, P_{read}], & \text{if } \neg R_2 \wedge R_3 \\ C_3 \sqcup C_4 : (P_{read}, 2P_{read} - 1], & \text{if } \neg R_3 \wedge \neg R_4 \\ C_4 : 2P_{read} - 1, & \text{if } R_4 \text{ ,} \end{cases} \tag{6.7}$$

which only depends on II and JJ parameters for a given architecture and a stencil order (ℓ). Figure 6.3 shows an example of how $nplanes_{Li}$ evolves with respect to $II \times JJ$ parameter.

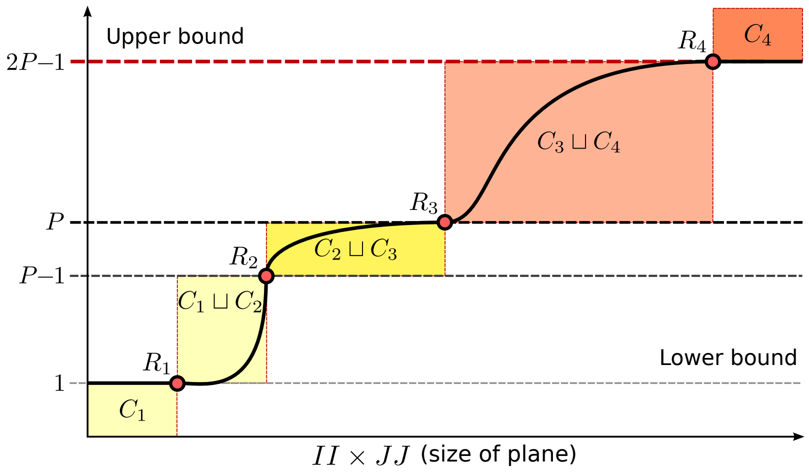


Figure 6.3: The different rules R_1, R_2, R_3 and R_4 bound the size of the problem (abscissa: $II \times JJ$) with the miss case penalties (ordinate: 1, $P_{read} - 1$, P_{read} and $2P_{read} - 1$).

Large discontinuities can appear in transitions of Equation 6.7 ($C_1 \sqcup C_2$, $C_2 \sqcup C_3$ and $C_3 \sqcup C_4$). This effect can be partially smoothed by using interpolation methods. Apart from the discrete transitioning, three types of interpolations have been added in our model: linear, exponential and logarithmic. An interpolation function ($f(x, x_0, x_1, y_0, y_1)$) requires five input parameters, the X-axis bounds (x_0 and x_1), the Y-axis bounds (y_0 and y_1) and the point in the X-axis (x) to be mapped into the Y-axis (y). In our problem domain, the X-axis represents the $II \times JJ$ parameters, whereas the Y-axis is the unknown $nplanes_{Li}$. For instance, for $C_1 \sqcup C_2$ transition, isolating II from R_1 and R_2 rules, II_{min} (x_0) and II_{max} (x_1) are respectively obtained, bounding the interpolation. By using their respective rules, and isolating the required variable for X-axis, the same procedure is also applied to the remaining transitions of Equation 6.7. In this way, an easy methodology is presented to avoid unrealistic discontinuities for the model.

6.3.3 Cache Interference Phenomena: II×JJ Effect

As stated before, three types of cache misses (3C) can be distinguished: compulsory (*cold-start*), capacity and conflict (*interference*) misses. Compulsory and capacity misses are relatively easily predicted and estimated [79]. Contrarily, conflict misses are hard to evaluate because it must be known where data are mapped in cache and when it will be referenced. In addition, conflict misses disrupt data reuse, spatial or temporal. For instance, a high frequency of cache interferences can lead to the rare *ping-pong* phenomena, where two or more memory references fall at the same cache location, therefore competing for cache-lines. Cache associativity can alleviate this issue to a certain extent by increasing the cache locations for the same address.

The cache miss model presented in Subsection 6.3.2 sets the upper bound for each of the four cases in terms of number of planes read for each plane written ($nplanes_{Li}$), thus establishing a discrete model. Nevertheless, this discrete scenario is unlikely to happen for cases C_2, C_3 and mainly C_4, due to their dependency on capacity and especially on conflict misses. There are two factors that clearly affect conflict misses: the reuse distance for a given datum [79] and the intersection of two data sets [40], giving consequently a continuum scenario. The former depends on temporal locality; the more data is loaded, the higher the probability that a given datum may be flushed from cache before its reuse. On the other hand, the latter depends on two parameters: the array base address and its leading dimensions.

In stencil computations the Z-X plane ($II \times JJ$ size) and the order of the stencil ($P_{read} = 2 \times \ell + 1$) are the critical parameters that exacerbate conflict misses. The conflict misses are related to the probability of interference, $P(i)$, and the column reuse of the central k-plane. $P(i)$ is proportional to the words of the columns to be reused ($II \times (P_{read} - 1)$) after reading the first central column with respect to the whole size of the central k-plane to be held in cache ($II \times JJ$),

$$P(i) = \frac{II \times JJ - II \times (P_{read} - 1)}{II \times JJ} = 1 - \frac{P_{read} - 1}{JJ} \in [0, 1] \ , \tag{6.8}$$

which yields to a logarithmic function depending on P_{read}, II and JJ parameters. A zero value means no conflict misses at all, whereas a probability of one means disruption of temporal reuse (high ratio of interferences) for columns of the central k-plane. Then, the $P(i)$

probability can be added to Equation 6.7 as

$$nplanes_{Li'} = nplanes_{Li} \times P(i) \; , \tag{6.9}$$

tailoring the boundary of read misses to their right value depending on the conflict misses issued. Thus, the larger the data used to compute one output plane ($I \times J$), the higher the probability of having capacity and conflict misses. Figure 6.4 shows the accuracy difference between the model with and without cache interference effect.

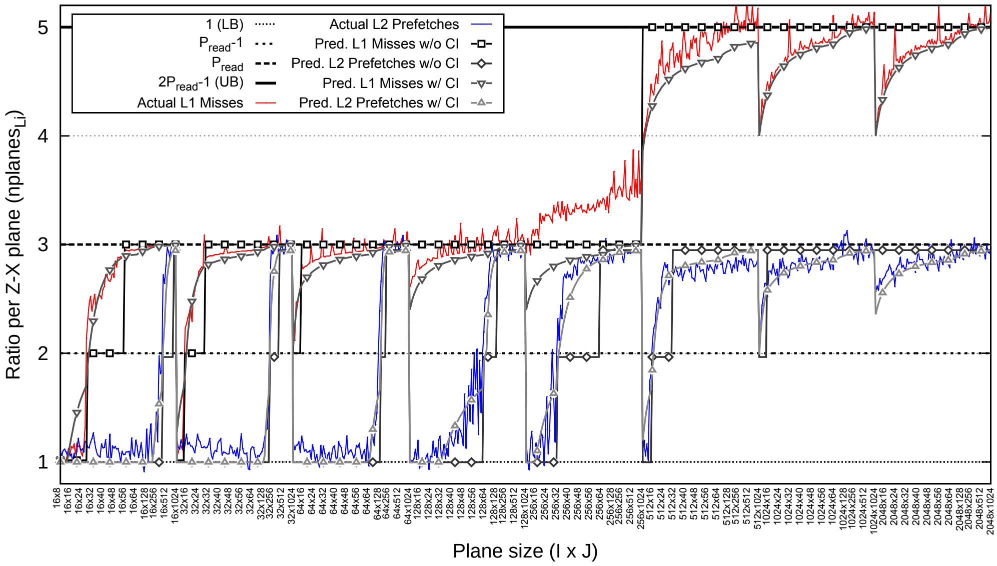


Figure 6.4: Cache interference (CI) effect as a function of problem size. These results are for a Naive 7-point stencil ($\ell = 1$) on Intel Xeon Phi. The corresponding read misses bounds for $nplanes_{Li}$ are shown as 1 (Lower Bound), 2 ($P_{read} - 1$), 3 (P_{read}) and 5 ($2P_{read} - 1$, Upper Bound). Whilst Equation 6.9 is not applied, a discrete model is obtained (straight lines with squares and diamonds). Conversely, its use leads to a continuum model (inverted and non-inverted triangles).

6.3.4 Additional Time Overheads

During the execution of HPC stencil codes, some additional overheads may arise. In this subsection, we briefly explain how these overheads are weighed in our stencil performance model. The overheads are categorized into three groups: parallelism, memory interferences and computational bottlenecks.

- Intra-node parallelism (OpenMP and Posix threads): small overheads may appear due to the thread initialization and synchronization tasks whether data is disjoint among threads. This overhead usually has a clear impact only on small dataset problems. In order to characterize its effect on the stencil model, a small (order of milliseconds) and constant ϵ (T_{OMP}) is included.

- Memory contention: TLB misses, ECC memories (error checking & corruption) and cache coherence policies between cores (*e.g.* MESI protocol) affect noticeably the memory performance. Nevertheless, all these effects are already taken into account in the memory characterization through our STREAM2 tool (see Section 6.4 for further details).

- Computational bottlenecks: stencil computations are mainly considered memory bound instead of compute bound (low OI) [25, 86]. Therefore, for the sake of simplicity, the tampering effect of floating-point operations is expected to be negligible, and thus not considered.

6.4 From Single-core to Multi-core and Many-core

Current HPC platforms are suboptimal for scientific codes unless they take fully advantage of simultaneous threads running on multi- and many-cores chips. Some clear examples of such architectures are Intel Xeon family, IBM POWER7 or GPGPUs. All of them with tens of cores and their ability to run in SMT mode. As a consequence, the parallel nature of the current stencil computation deployments leads us to extend accordingly our model. To that end, the parallel memory management is a main concern, and this section is fully devoted to sort it out.

In order to characterize the memory management of multi-core architectures, the bandwidth measurement is critical. The bandwidth metrics are captured for different configurations using a bandwidth profiler such as STREAM2 benchmark [54]. To validate the results, our STREAM2 version [24] has been significantly extended by adding new features such as vectorization (SSE, AVX and Xeon Phi ISAs), aligned and unaligned memory access, non-temporal writes (through Intel pragmas), prefetching and non-prefetching bandwidths, thread-level execution (OpenMP) and hardware counters instrumentation (PAPI).

The process to obtain bandwidth measurements is straightforward. First, the thread number is set through the OMP_NUM_THREADS environment variable. Then, each thread is pinned to a specific core of the platform (*e.g.* using *numactl* or KMP_AFFINITY variable in Xeon Phi architecture). Finally, the results obtained for DOT (16 bytes/read) and FILL (8 bytes/write) kernels are respectively used as read and write bandwidths for the different cache hierarchies of the model. Figure 6.5 shows an example of the bandwidths used for a particular case in the Intel Xeon Phi platform. The importance of mimicking the environment conditions is crucial, in particular the execution time accuracy of the model is very sensitive to the real execution conditions. This means that the characterization of the memory bandwidth must be similarly performed in terms of: number of threads, threads per core, memory access alignment, temporal or non-temporal writes and SISD or SIMD instruction set.

Additionally, there are some memory resources that might be shared among different threads running in the same core or die. In order to model the behaviour in such cases,

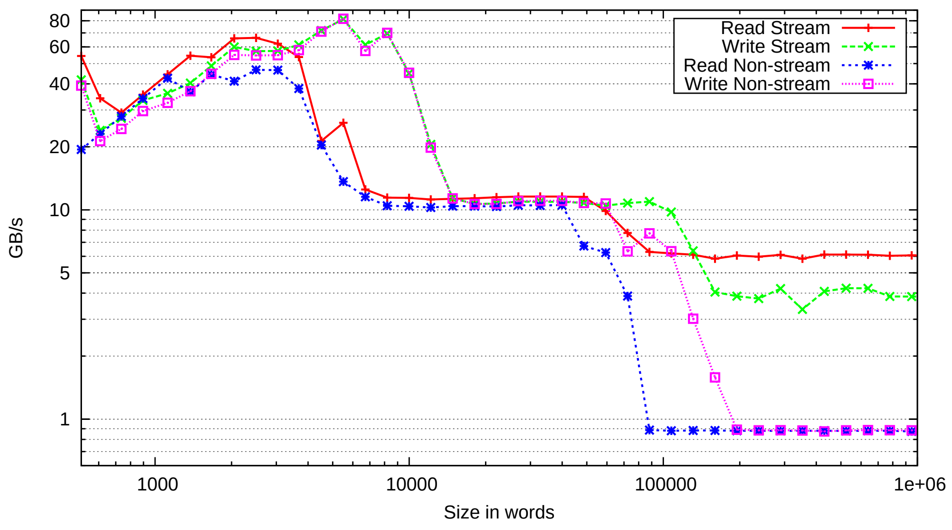


Figure 6.5: STREAM2 results for Intel Xeon Phi architecture (4 threads, 2 per core). Each plateaux represents the sustainable bandwidth of a cache level.

the memory resources are equally split among all threads. This is, if we have a cache size ($size_{Li}$ in rules $R_{1,2,3,4}$) of N KBytes, then each thread would turn out to have a cache size of $size_{Li} = N/nthreads_{core}$.

6.5 Modeling the Prefetching Effect

Prefetching is a common feature in modern architectures. It enables a reduction of cache misses, thus increasing the hit ratio and minimizing the memory latency. Data prefetching, also known as streaming in the literature, consists of fetching data eagerly from main memory into cache before it is explicitly requested. As the processor-memory performance gap has widened, the significance of prefetching has noticeably increased. Prefetching comes in two flavours, hardware and software driven. Furthermore, most modern architectures support aggressive prefetching of streams at different levels of the cache hierarchy simultaneously, which can can have a drastic effect in the application performance. Hence, it is necessary to include prefetching capabilities to the model so that the stencil behaviour can be predicted in current architectures.

This section is devoted to discuss and introduce the necessary extensions into the model to effectively capture the prefetching behavior.

6.5.1 Hardware Prefetching

The modeling of the hardware prefetching mechanism is complex, in particular to figure out whether a stream triggers any misses. We devised two different methods for this purpose: a simple approach based on categorizing the Z-X planes that are prefetched and non-prefetched, and a second method based on an efficiency metric.

Prefetched and non-prefetched planes: Misses of the model are divided into two groups, prefetched and non-prefetched, depending on the concurrent streams that the prefetching engine supports. First, the Z-X planes that can be prefetched ($nplanes^{S}_{Li}$) and not prefetched ($nplanes^{NS}_{Li}$) at the cache level i are computed as

$$\begin{aligned} nplanes^{NS}_{Li} &= \max(nplanes_{Li} - pref_{Li}, 0) & nplanes^{S}_{Li} &= nplanes_{Li} - nplanes^{NS}_{Li} \\ Misses^{NS}_{Li} &= \lceil II/W \rceil \times JJ \times KK \times nplanes^{NS}_{Li} & Misses^{S}_{Li} &= \lceil II/W \rceil \times JJ \times KK \times nplanes^{S}_{Li} \end{aligned} \tag{6.10}$$

where $pref_{Li}$ refers to the number of stream channels supported by the current hierarchy level i. Then, prefetched and non-prefetched cache line misses are evaluated using their $nplanes_{Li}$. Next, hits are estimated within each group using the general rule ($Hits_{Li} = Misses_{Li-1} - Misses_{Li}$) but using their specific cache misses as

$$\begin{aligned} Hits^{NS}_{Li} &= Misses^{NS}_{Li-1} - Misses^{NS}_{Li} & Hits^{S}_{Li} &= Misses^{S}_{Li-1} - Misses^{S}_{Li} \\ T^{NS}_{Li} &= cacheline/Bw^{read-NS}_{Li} & T^{S}_{Li} &= cacheline/Bw^{read-P}_{Li} \\ T_{Li} &= Hits^{NS}_{Li} \times T^{NS}_{Li} + Hits^{S}_{Li} \times T^{S}_{Li} \end{aligned} \tag{6.11}$$

aggregating the partial transferring costs, which have been computed with their corresponding bandwidths (T^{NS}_{Li} and T^{S}_{Li}), to obtain the final T_{Li}. Recall, that the actual bandwidths for prefetched and non-prefetched streams are computed through the STREAM2 benchmark (see Section 6.4).

Prefetching effectiveness: Recent works [52, 56] have characterized the impact of prefetching mechanism on scientific application performance. They establish a new metric called *prefetching effectiveness*, which computes the fraction of data accesses to the next memory level that are initiated by the hardware prefetcher. Therefore, for a given data cache level (*DC*), its prefetching effectiveness is computed as

$$DC_{effectiveness} = DC\_Req\_PF / DC\_Req\_All \in [0, 1] \ , \tag{6.12}$$

where DC_Req_PF refers to the number of cache-lines requests initiated by the prefetching engine, and DC_Req_All represents the total number of cache-lines transferred to the DC level (including demanding and non-demanding loads). This approach has been adopted in our model as the alternative way to accurately capture the prefetching behaviour.

In order to be able to characterize the *prefetching effectiveness* in our testbed platform, the *Prefetechers* micro-benchmark was used (see Section 3.6). This benchmark traverses a chunk of memory simultaneously by different threads and changes the number of stream accesses in a round-robin fashion. Then, to compute their effectiveness, a set of hardware performance counters were gathered through PAPI. For instance, on Intel Xeon Phi architecture, two native events were instrumented to compute the *prefetching effectiveness*: HWP_L2MISS and L2_DATA_READ_MISS_MEM_FILL. Figure 6.6 shows the results obtained for this platform over the L2 hardware prefetcher.

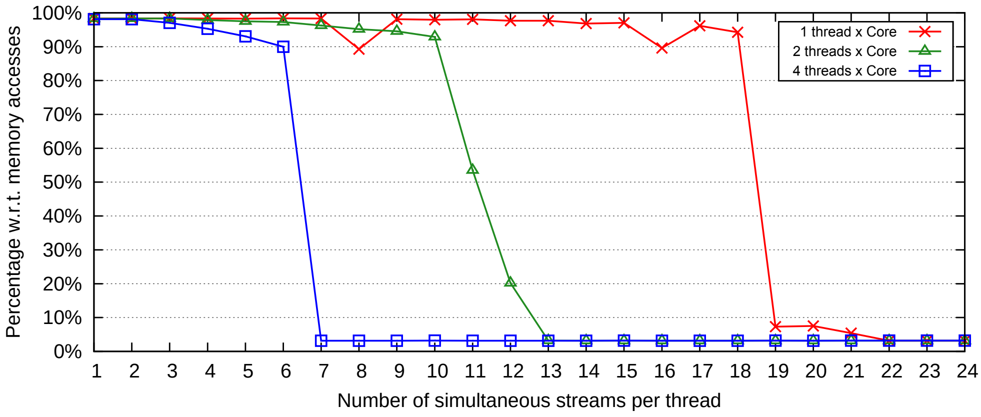


Figure 6.6: L2 prefetching efficiency for Intel Xeon Phi architecture. The efficiency has been computed using one core and varying the SMT configuration from 1 to 4 threads.

The *prefetching effectiveness* ($DC_{effectiveness}$) is then used to compute the total number of cache-line misses that are fetched using streaming bandwidths ($nplanes^S_{Li}$) and those that are fetched using a regular bandwidth ($nplanes^{NS}_{Li}$):

$$\begin{aligned} nplanes^S_{Li} &= nplanes_{Li} \times DC_{effectiveness}, \\ nplanes^{NS}_{Li} &= nplanes_{Li} \times (1 - DC_{effectiveness}) \ . \end{aligned} \tag{6.13}$$

Similarly to the memory resources, prefetching engines might be shared among threads running on the same core. In such scenarios, the *prefetching effectiveness* is computed with our prefetching tool varying the number of threads per core (for instance, 2 and 4 threads results can be observed in Figure 6.6). In fact, these results are insightful and help to understand when the core performance might be degraded due to excessive simultaneous streams, thus adversely affecting the parallel scaling of stencil computations.

6.5.2 Software Prefetching

Software prefetching is a technique where compilers, and also programmers, explicitly insert prefetching operations similar to load instructions into the code. Predicting the performance of software prefetching is challenging. Compilers use proprietary heuristics in order to decide where (code location), which (data array) and how long in advance (look-ahead in bytes) data should be prefetched. Furthermore, programmers can even harden this task by adding special hints in the code to help the compiler make some of these decisions [59]. As software prefeching produces regular loads in the cache hierarchy, it also prevents hardware prefetcher to be triggered when it performs properly [34]. Thus, the failure or success of software prefetching affects collaterally the hardware prefeching behaviour.

As a result of all above commented issues, software prefetching has not been taken into account in the present work. The software prefetching can be disabled in Intel compilers by using the pragma `noprefetch` or the `-opt-prefetch=0` flag during the compilation.

6.6 Optimizations

The state-of-the-art in stencil computation is constantly being extended with the publication of several optimization techniques in recent years. Under specific circumstances, some of those techniques improve the execution performance. For instance, space blocking is a tiling strategy widely used in multi-level cache hierarchy architectures. It promotes data reuse by traversing the entire domain into small blocks of size $TI \times TJ$ which must fit into the cache [43, 72]. In consequence, space blocking is especially useful when the dataset structure does not fit into the memory hierarchy. This traversal order reduces capacity and conflict misses in least-stride dimensions increasing data locality and overall performance. Notice that a search of the best block size parameter ($TI \times TJ$) must be performed for each problem size and architecture.

A second example of stencil optimization is the Semi-stencil algorithm [25]. This algorithm changes the way in which the spatial operator is calculated and how data is accessed in the most inner loop. Actually, the inner loop involves two phases called *forward* and *backward* where several grid points are updated simultaneously. By doing so, the dataset requirements in the inner loop are reduced, while keeping the same number of floating-point operations. Thereby, increasing data reuse, and thus the OI. Conversely to read operations, the number of writes are slightly increased because the additional point updates. Due to this issue, this algorithm only improves performance in medium-large stencil orders ($\ell > 2$).

These two stencil optimizations have been included into our model. The motivation of modeling them is two-fold. First, to reveal insights of where and why an algorithm may perform inadequately for a given architecture and environment. Second, to analytically guide the search for good algorithmic parameter candidates without the necessity of obtaining them empirically (brute force).

6.6.1 Spatial Blocking

Space blocking is implemented in our model by including similar general ideas as [21], but adapting them in order to suit the advantages of our cost model. Basically, the problem domain is traversed in $TI \times TJ \times TK$ blocks. Then, first the blocks on each direction are computed as $NBI = I/TI$, $NBJ = J/TJ$, and $NBK = K/TK$. Therefore, the total number of tiling iterations to perform are $NB = NBI \times NBJ \times NBK$. Blocking may be performed as well in the unit-stride dimension. Given that data is brought to cache in multiples of the

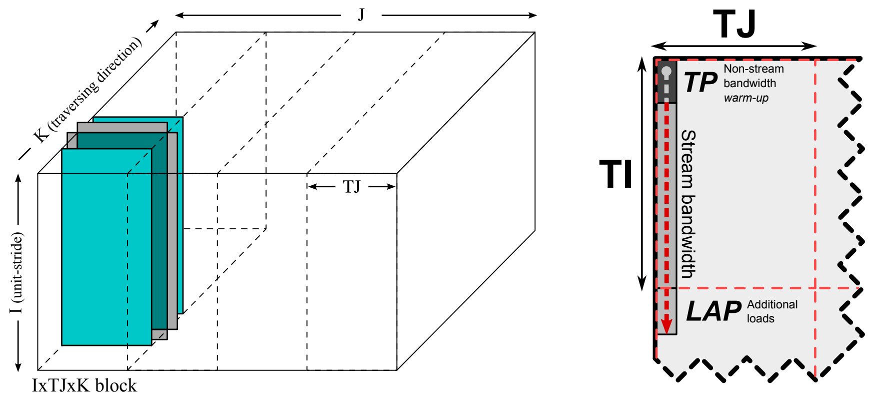


Figure 6.7: Modeling of spatial blocking optimization. Left: spatial-blocking layout with $I \times TJ \times K$ tiles. Right: detail of prefetching effect when $TI \neq I$, disrupting memory access and increasing data transfers.

cache-line, additional transfer overhead may arise when TI size is not multiple of cache-line. This is considered in the model by reassigning I, J, K and their extended dimensions as follows:

$$\begin{aligned} I &= \lceil TI/W \rceil \times W, & J &= TJ, & K &= TK, \\ II &= \lceil (TI + 2 \times \ell)/W \rceil \times W, & JJ &= TJ + 2 \times \ell, & KK &= TK + 2 \times \ell\,. \end{aligned} \tag{6.14}$$

The new II and JJ parameters are then used for rules $R_{1,2,3,4}$ to estimate $nplanes_{Li}$ based on the blocking size. Finally, Equation 6.5 shall be rewritten as

$$Misses_{Li}^{[S,NS]} = \lceil II/W \rceil \times JJ \times KK \times nplanes_{Li}^{[S,NS]} \times NB\,, \tag{6.15}$$

where NB factor is considered to adjust streamed $(^{S}_{Li})$ and non-streamed $(^{NS}_{Li})$ misses depending on the total number of blocking iterations.

Architectures with prefetching features may present performance degradation when $TI \neq I$ [42]. Blocking on the unit-stride dimension may tamper streaming performance due to the interference caused to the memory access pattern detection of the prefetching engine (see Figure 6.7). The triggering of the prefetching engine involves a *warm-up* phase, where a number of cache-lines must be previously read (TP). Additionally, prefetching engines keep a look-ahead distance (LAP) of how many cache-lines to prefetch in advance. Disrupting a regular memory access will produce LAP additional fetches to the next cache level if the prefetching engine was triggered. Considering all these penalties, the cache misses are updated with:

$$\begin{aligned} Misses_{Li}^{NS} &\overset{+}{=} TP \times JJ \times KK \times nplanes_{Li}^{NS} \times NB, && \text{if } II/W \geq TP, \\ Misses_{Li}^{S} &\overset{+}{=} LAP \times JJ \times KK \times nplanes_{Li}^{S} \times NB, && \text{if } II/W \geq TP\,. \end{aligned} \tag{6.16}$$

TP and LAP parameters can be obtained from processor manufacturer's manuals or empirically through our prefetching benchmark. To deduce such parameters, the *Prefetching* benchmark was modified to traverse arrays in a blocked fashion whilst TI parameter was slowly increased through consecutive executions. Then, the prefetching hardware counter was monitored in order to flag at what precise point ($TP = \lceil TI/W \rceil$) the prefetching metric soared significantly. Likewise, LAP parameter was estimated by counting the extra prefetching loads (apart from the TP) that were issued.

6.6.2 Semi-stencil Algorithm

Adapting the model for the Semi-stencil algorithm is equally straightforward. Indeed, this can be achieved by setting P_{read} and P_{write} parameters correctly. By default, in a partial Semi-stencil implementation (*forward* and *backward* phases on X and Y axes), $\ell + 1$ Z-X planes from $\mathcal{X}^t$ and one $\mathcal{X}^{t+1}$ plane (k-central plane update) are read for each k iteration. As output, two planes are written back as partial ($\mathcal{X}^{t+1}_{i,j,k+\ell}$) and final ($\mathcal{X}^{t+1}_{i,j,k}$) results. However, these values can slightly increase when no room is left for the k-central columns; thus yielding

$$\begin{aligned} P_{read} &= \ell + 2, \quad P_{write} = 2, \quad \text{if } \neg R_4 \\ P_{read} &= \ell + 3, \quad P_{write} = 3, \quad \text{if } R_4 \end{aligned} \tag{6.17}$$

as the new data requirements to compute one output plane. This adaptability reveals the model resilience, where an absolutely different stencil algorithm can be modeled by simply tuning a couple of parameters.

6.7 Experimental Results

In this section, the experimental results are presented to validate the robustness of the model. To that end, the predicted results (execution times and cache misses) are compared with actual (execution times and hardware or software counted cache misses), computing also their relative errors. In order to achieve that, we have followed a methodology that consists in the following five stages:

Stage 1. Test-cases Parameters: due to the sheer parameter combinations to be analysed, some representatives test cases are selected within the experimental space. The test cases space is a combination of stencil size, dataset dimension, algorithm implementation and parameters for each specific optimization algorithm.

Stage 2. Real Execution Performance: the test cases selected in the previous stage are executed and the results are stored for further use. Two sets of real data have been collected for the validation step: the real execution times and hardware or software counters (cache misses and prefetching metrics). In order to gather these metrics, the tools presented in Section 3.4 have been used.

Stage 3. Model Parameters and Architectural Characterization: each platform information is gathered, such as cacheline size, cache hierarchy size, prefetching capabilities and bandwidth (streamed and non-streamed). Some of them, for instance memory bandwidth and prefetching effectiveness, are obtained through characterization benchmarks such as STREAM2 and *Prefetching* (see Sections 6.4 and 6.5). The rest of the information is obtained reviewing the manufacturer's datasheets. Afterwards, the consolidated information is added to the model through configuration files.

Stage 4 & 5. Model Performance and Validation: finally, the predicted performance is obtained by using the model for the test cases selected in Stage 1 and are checked against actual executions from Stage 2. As shown in Figure 6.8, this methodology has a feedback mechanism, where model parameters can be adjusted and experiments repeated if the relative error computed in Stage 5 is out of acceptable correctness range. It may occur that some characteristics are not clearly specified by the manufacturer, then a process of refinement must be performed. This refinement process is conducted executing Stages 3-4-5 as many times as it is needed.

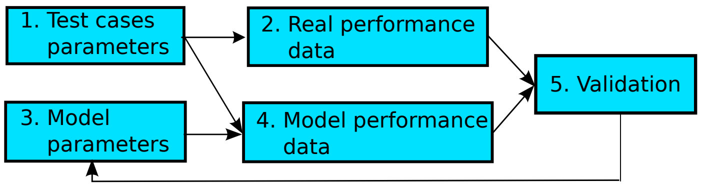


Figure 6.8: Work-flow of the model methodology with feedback mechanism.

Due to the enhancing process conducted during the research of the stencil model, the experimental results are presented in two separated sections. The first section shows the initial stage results for the preliminary model on three different platforms. The second section exposes the results of the advanced model on a complex many-core architecture. Table 6.1 summarizes the features enabled for each experimental results group.

All experimental results in this section were validated using the StencilProbe [43], a synthetic benchmark that we have extended. The new StencilProbe features [25] include: different stencil orders (ℓ), thread support (OpenMP), SIMD code, instrumentation and new optimization techniques (*e.g.* spatial blocking and Semi-stencil). This benchmark implements the stencil scheme shown in Algorithm 3, where star-like stencils with symmetric and constant coefficients are computed using 1st order in time and different orders in space (see Table 3.6).

6.7.1 Preliminary Model Results

The stencil sizes of the preliminary experiments carried out are 13, 25, 43 and 85-point. For the sake of simplicity, we have collapsed the 3D dataset dimension to cubic (N^3). The dataset size ranges from 128^3 to 512^3 points, in strides of 16. Thus, in total, 288 test case results have

Features	Preliminary Model	Advanced Model
Base Model	Multi-Level cache	Multi-Level cache
L1 read estimation	Register predictor	Static analysis
Cases & Rules	4 Cases & Bounds	4 Cases/Rules & Bounds
Interpolation Methods	✗	Linear, exponential and logarithmic
Cache interference	✗	✓
Multiprocessing	Single-core	SMT and Multi-core
Overheads	✗	T_{OMP}
Prefetching method	Planes policy	Prefetching effectiveness
Optimizations	✗	Blocking & Semi-stencil
Architectures tested	Intel Nehalem, AMD Opteron, IBM BG/P	Intel Xeon Phi (MIC)

Table 6.1: Features used for preliminary and advanced model results.

been analysed. Table 6.2 details the list of parameters used for the preliminary model on each platform.

Parameters		Intel Nehalem	IBM BlueGene/P	AMD Opteron
Register availability[†]	GPR_{free}	10	26	10
	FPR_{free}	12	26	12
Block size	$cacheline$	64 bytes	32/128 bytes	64 bytes
Prefetching channels	$pref_{L1}$	2	0	2
	$pref_{L2}$	2	7	0
	$pref_{L3}$	0	0	0
Cache capacity	$size_{L1}$	32 kB	32 kB	64 kB
	$size_{L2}$	256 kB	1920 bytes	512 kB
	$size_{L3}{}^{*}$	8 MB	8 MB	2 MB
Measured Bandwidth[‡] (GB/s)	Bw^{read}_{L1}	49.4 (23.4)	6.2 (3.2)	29.1 (14.6)
	Bw^{read}_{L2}	29.4 (12.5)	2.1 (1.3)	13.9 (4.1)
	Bw^{read}_{L3}	21.1 (8.1)	2.0 (0.6)	7.6 (2.2)
	Bw^{read}_{Memory}	8.2 (3.5)	2.0 (0.6)	3.3 (1.3)
	Bw^{write}_{L1}	49.4 (24.6)	3.3 (3.3)	29.9 (8.6)
	Bw^{write}_{Memory}	7.9 (3.9)	3.3 (1.9)	4.9 (0.8)

Table 6.2: Parameters used in the preliminary model. [†]The available registers have been estimated doing assembly register analysis of a naive case. [‡]Streaming and non-streaming bandwidths (shown in parenthesis). *L3 cache is shared among the cores.

In order to visualize and analyse the amount of data at hand, firstly we review the results for a couple of interesting cases, and secondly we summarize the whole results in Table 6.3. In Figure 6.9.Top, we can observe a clear pattern for an AMD Opteron case with a high-order stencil. The predicted curve of L2 cache misses diverges from actual (hardware counters) and simulated (Valgrind) curves. Even so, the relative error with respect to the predicted results stays in an acceptable 10 to 20% for test cases bigger than 256^3. On the other hand, simulated and actual curves follow a similar path, demonstrating that Valgrind cache simulator can be a reliable tool for those metrics that can not be gathered using hardware counters. Figure 6.9.Bottom depicts a different scenario, the measured and the predicted execution times curves follow a closer path, where relative error remains under 12% for medium and large problems. The main reason for this different behavior is that on high-order stencils (85-point) the effect of L3 cache and memory latency overwhelms the remaining cache hierarchy (L1 and L2), thus being the dominant factor of the total execution time. In addition, the execution time of large problem sizes with high-order stencils is slightly perturbed by the AMD Opteron prefetching mechanism, with only 2 streamers in the L1 cache, thus diminishing its effect and easing the prediction by the preliminary model.

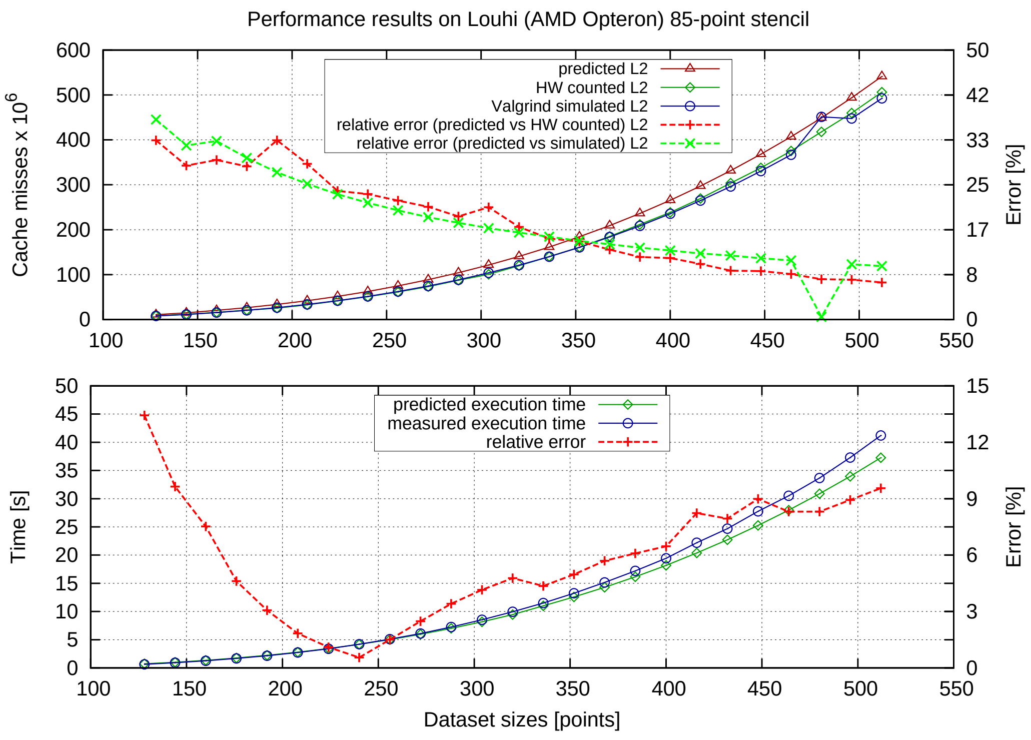


Figure 6.9: AMD Opteron results for the preliminary model with a 85-point naive stencil. On the top figure, L2 cache misses are shown for the model (predicted), Valgrind (simulated) and hardware counters results (actual). On the bottom, the execution times are presented for predicted and measured metrics.

Figure 6.10.Top shows the L1 and L2 results for a 25-point stencil case on Intel Nehalem architecture. In general, the relative error remains under 5% and the cache metrics follow the simulated curves except for L1 misses when problem sizes are between 200^3 and 400^3, where the relative error soars considerably. However, in terms of execution time prediction, the Figure 6.10.Bottom shows that the relative error stays under 15% for the most relevant part of the graph due to the low effect of the L1 misprediction in the global execution time.

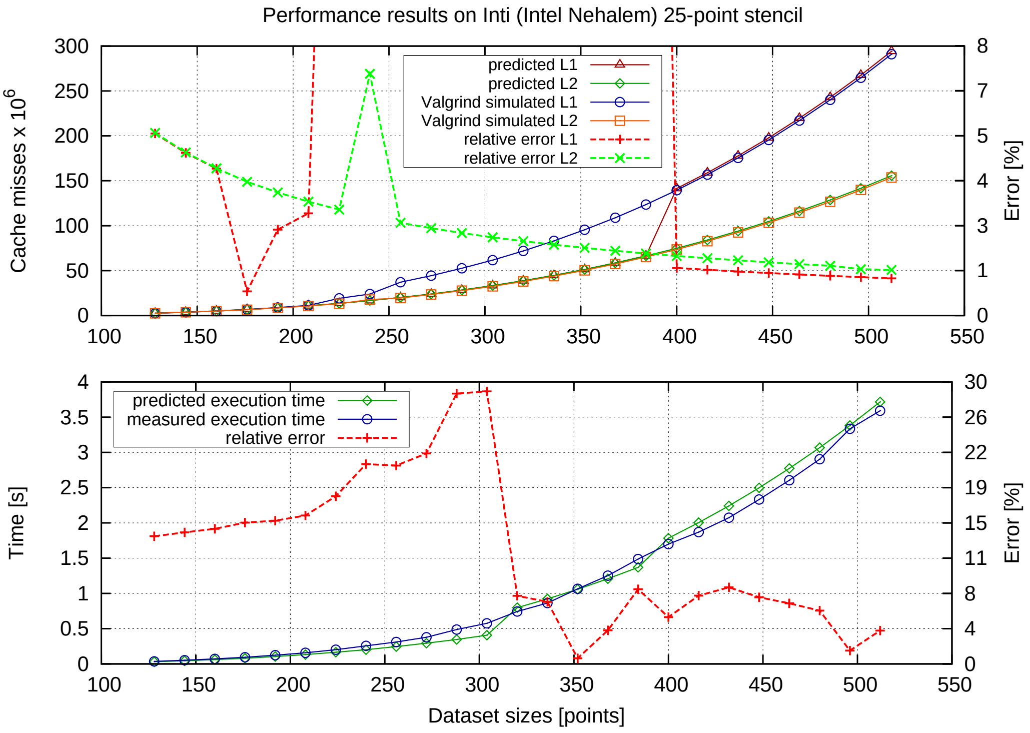


Figure 6.10: Intel Nehalem results for the preliminary model with a 25-point naive stencil.

	AMD Opteron				IBM BlueGene/P				Intel Nehalem			
Size	13	25	43	85	13	25	43	85	13	25	43	85
Max	37.2	37.0	18.2	13.4	17.2	24.5	27.9	16.8	46.2	28.9	38.2	20.0
Min	3.5	1.8	1.8	0.5	2.4	0.6	8.7	1.0	0.4	0.6	0.6	2.4
Avg.	13.0	11.6	8.5	5.8	11.1	17.1	21.1	10.8	9.9	11.9	22.1	12.5
Stdev.	16.4	18.7	3.7	2.7	6.8	7.5	10.8	10.1	0.5	7.1	4.9	0.5

Table 6.3: Summary of preliminary model results. Statistics for relative errors (%) between predicted and measured execution times.

Reviewing the summarized results in Table 6.3, we can conclude that in general, considering both accuracy and stability, the model performs reasonable well for the AMD architecture, where the average relative error is 9.7% for all stencil sizes. On the other hand, the model is less effective on the Intel and especially on the IBM architectures where their maximum relative errors are higher. In addition, it is important to remark that in a wide range of problem sizes with low and medium stencil orders (13 and 25-point) the relative error is higher.

The main cause of this inaccuracy is the misprediction of the prefetching method followed by the preliminary model. This behavior reveals a serious problem of this approach to capture accurately any kind of stencil topology and problem size. In contrast, this situation is not clearly exposed on high order stencils and very large problems due to the diminishing effect of the prefetching mechanism on the overall execution time.

6.7.2 Advanced Model Results

This subsection estimates through experimental results how accurate is the advanced model when exposed to: widespread problem sizes, hardware prefetching, thread parallelism and code optimizations techniques.

A large number of different problem sizes were explored so as to validate the advanced model accuracy for a wide parametrical space. Recall that the two first dimensions (on Z and X axes) are the critical parameters that increase the cache miss ratio ($nplanes_{Li}$) for a given stencil order (ℓ) and architecture. Therefore, the last dimension K was set to a fixed number, and the I and J dimensions were widely varied covering a large spectrum of grid sizes. All the experiments were conducted using double-precision, and the domain decomposition across threads was performed by cutting in the least-stride dimension (Y axis) with static scheduling. This scheduling almost avoids the collateral effects between threads reducing the cacheline invalidation because Read For Ownership (RFO) and the data duplicated among core's caches. Table 6.4 summarizes the different parameters used.

Parameters	Range of values
Naive sizes ($I \times J \times K$)	$8 \times 8 \times 128 \ldots 2048 \times 1024 \times 128$
Rivera sizes ($I \times J \times K$)	$512 \times 2048 \times 128$
Stencil sizes (ℓ)	1, 2, 4 and 7 (7, 13, 25 and 43-point respectively)
Algorithms	{Naive, Rivera} $\times$ {Classical, *Semi-stencil*}
Block sizes (TI and TJ)	$\{8, 16, 24, 32, 64, 128, 256, 512, 1024, 1536, 2048\}$

Table 6.4: List of parameters used for the model and the StencilProbe benchmark.

We have used a leading hardware architecture in our advanced experiments, the popular Intel Xeon Phi 5100 series (SE10X model), also known as MIC. This architecture shows an outstanding appeal for this research due to its support for all of the new hardware features that our extended model intend to cover (hardware prefetching and a high level of parallelism with SMT/SMP).

Hardware counters were gathered for all the experiments so as to validate the model results against actual executions. Table 6.5 shows the hardware performance counters instrumented. The stencil code generated by StencilProbe is vectorized, and therefore only vector reads were fetched (VPU_DATA_READ) during executions. Additionally, the L2 prefetcher in Xeon Phi can also prefetch reads for a miss in a write-back operation (L2_WRITE_HIT) when it has the opportunity. Then, in order to fairly compare the prefetched read misses of the model with actual metrics, the L2 prefetches (HWP_L2MISS) were normalized. This normalization was performed by subtracting reads owing to a miss in a write operation scaled by the *prefetching efficiency*. Likewise, some writes were considered prefetched (L2_WRITE_HIT $\times DC_{effectiveness}$) and others not (L2_WRITE_HIT $\times(1-DC_{effectiveness})$) due to contention of the L2 prefetching engine. Finally, the remaining miss counters (VPU_DATA_READ_MISS and L2_DATA_READ_MISS_MEM_FILL) only consider demanding reads, initiated by explicit reads, and therefore were directly used as non-prefetched read misses. It is important to mention that, in our preliminary model [24], several complex formulas were derived to estimate the number of reads issued to the first cache level ($Hits_{L1}^{word}$) through a register pressure predictor. This estimation is not straightforward and lacked accuracy. However, we realized that this parameter kept constant per loop iteration and could be precisely estimated by performing static analysis of the inner stencil loop only once (counting the numbers of reads in the object file).

Description	Intel Xeon Phi Events	Time Cost Formulas
Cycles	CPU_CLK_UNHALTED	$T_{L1} =$ (L1 Hits - L1 Misses) $\times Bw_{L1}^{cline}$
L1 Hits	VPU_DATA_READ	$T_{L2} = Bw_{L2}^{cline} \times$ (L1 Misses - L2 Misses -
L1 Misses	VPU_DATA_READ_MISS	(L2 Prefetches - L2 Writes $\times$ Pref Eff))
L2 Misses	L2_DATA_READ_MISS_	$T_{Mem} =$ L2 Misses $\times Bw_{Mem}^{NS} + Bw_{Mem}^{S} \times$
	MEM_FILL	(L2 Prefetches - L2 Writes $\times$ Pref Eff)
L2 Prefetches	HWP_L2MISS	$T_{Writes} =$ L2 Writes $\times$ Pref Eff $\times Bw_{Write}^{S}$
L2 Writes	L2_WRITE_HIT	+ L2 Writes $\times$ (1 - Pref Eff) $\times Bw_{Write}^{NS}$

Table 6.5: Hardware counters and the formulas used to compute the projected time.

An aim of this subsection is to prove that stencil computations can be accurately modeled on SMT architectures. Therefore, all possible SMT combinations for a single core were sampled. Our tests were conducted using 4 threads varying their pinning to cores. The Intel KMP_AFFINITY environment variable was accordingly set to bind threads to the desired cores. The SMT configurations tried for each test were: 1 core in full usage (4 threads per core), 2 cores in half usage (2 threads per core) and 4 cores in fourth usage (1 thread per core).

Due to the sheer number of combinations sampled, only the most representative and interesting results are shown. Results have been categorized as a function of core occupancy (1, 2 and 4 threads per core) in order to explicitly visualize the effect of resource contention on the actual metrics and test the predicted results.

Figure 6.11 shows the actual and the predicted misses as a factor of $nplanes_{Li}$ with the advanced model version (prefetched and non-prefetched for L2) on all three SMT configurations using a Naive stencil order of $\ell = 4$. In this case 680 different problem sizes (X axis in figures) were tested per configuration. Recall that software prefetching was disabled, and therefore, L1 or L2 cache levels do not exhibit collateral effects due to compiler-assisted prefetch. This figure is very insightful because the empirical results clearly corroborate our

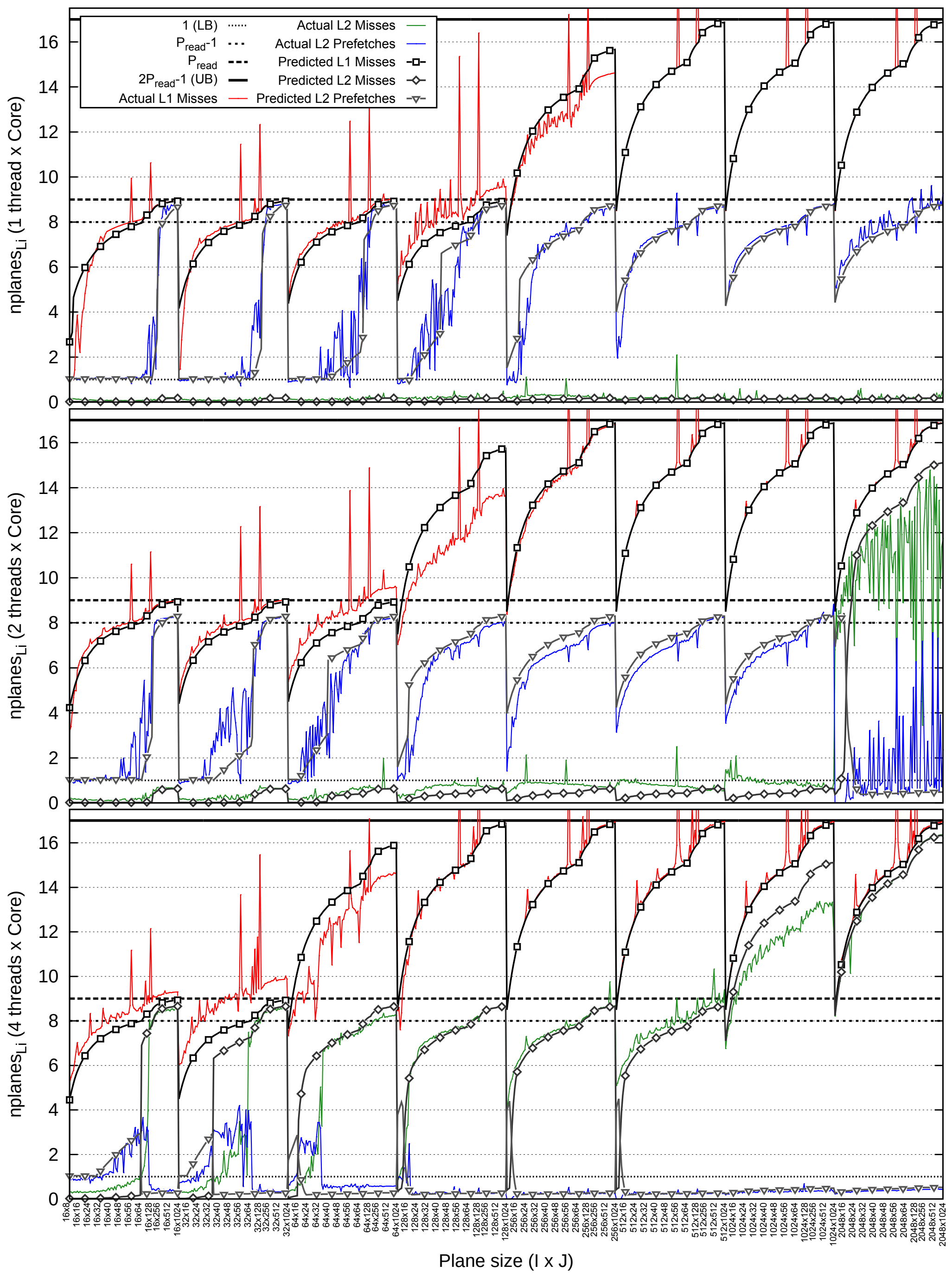


Figure 6.11: Actual and predicted for prefetched (inverted triangles) and non-prefetched (squares and diamonds) cache-lines for the three SMT configurations. These results are for the Naive implementation of a medium-high order ($\ell = 4$) stencil.

thoughts regarding the different bounds applied in the stencil model. Indeed, in a $\ell = 4$ stencil the read miss bounds for the model are: 1, 8 ($P_{read} - 1$), 9 (P_{read}) and 17 ($2P_{read} - 1$) per each $I \times J$ plane computed. Actual L1 and L2 misses tend to these bounds when a specific problem size is reached, never reaching beyond the upper bound ($2P_{read} - 1$), which is showed as a solid coarse horizontal line in all plots. Cache levels with prefetched and non-prefetched misses are a special case due to their direct relation with $DC_{effectiveness}$ ratio, and therefore they might be under the lower bound (1). Additionally, as the threads per core are increased, the inflection points (transitions) between bounds ($C_1 \sqcup C_2$, $C_2 \sqcup C_3$ and $C_3 \sqcup C_4$) are triggered earlier in terms of plane size ($I \times J$). The larger the number of threads running concurrently on the same core, the more contention and struggle for shared resources occurs. Likewise, some spikes appear on account of *ping-pong* effect, where different planes and columns addresses fall in the same cache set. This effect is also exacerbated as more threads are pinned to the same core. However, this effect is not captured by our model because it would require a multi-level set-associative cache model, which is not covered yet in our model.

Comparing the empirical (hardware counters) versus the analytical results (model), it can be observed that the advanced model predicts accurately the number of misses on both levels of the cache hierarchy, including those reads that are prefetched. However, some slight mispredictions appear in specific sizes when the transition between miss cases is triggered. Deciding a discrete point ($I \times J$) for transitions is difficult, and it might depend on other parameters apart from those considered in this work. Nevertheless, we think that our rules ($R_{1,2,3,4}$) have approximated these transitions fairly well. It is also important to mention the prediction of the L2 prefetching engine, especially in the late executions for 2 threads and in the early ones for 4 threads per core configurations. As hardware metrics show, in these cases, the prefetching effect starts disrupting the results due to contention. Nonetheless, in general the predicted results follow properly the trend of both type of misses as a result of the $DC_{effectiveness}$ parameter.

The model accuracy is verified in Figure 6.12, which shows a summary of three types of execution times: actual, projected and predicted. The actual times were obtained using the CPU clock cycles metric (CPU_CLK_UNHALTED). On the other hand, the projected times were computed with the aggregated time of T_{L1}, T_{L2}, T_{Mem} and T_{Write} by using actual hardware counters of reads, writes and misses with their respective bandwidth parameters (STREAM2 characterization). Finally, the predicted times follow the same idea than the projected but using the estimations of our model instead of the instrumented ones. The purpose of the projected time is the verification of the aggregated equation and the calibration of the bandwidth parameters at each cache level. Therefore, the projected time plays an important role ensuring that predicted times are a faithful representation of an actual execution.

Comparing the execution times shown in Figure 6.12, we observe that the predicted relative error (right axis) is very low in most of the cases. However, as the results reveal, some predictions have a high error (2 threads per core). Reviewing the cache miss predictions

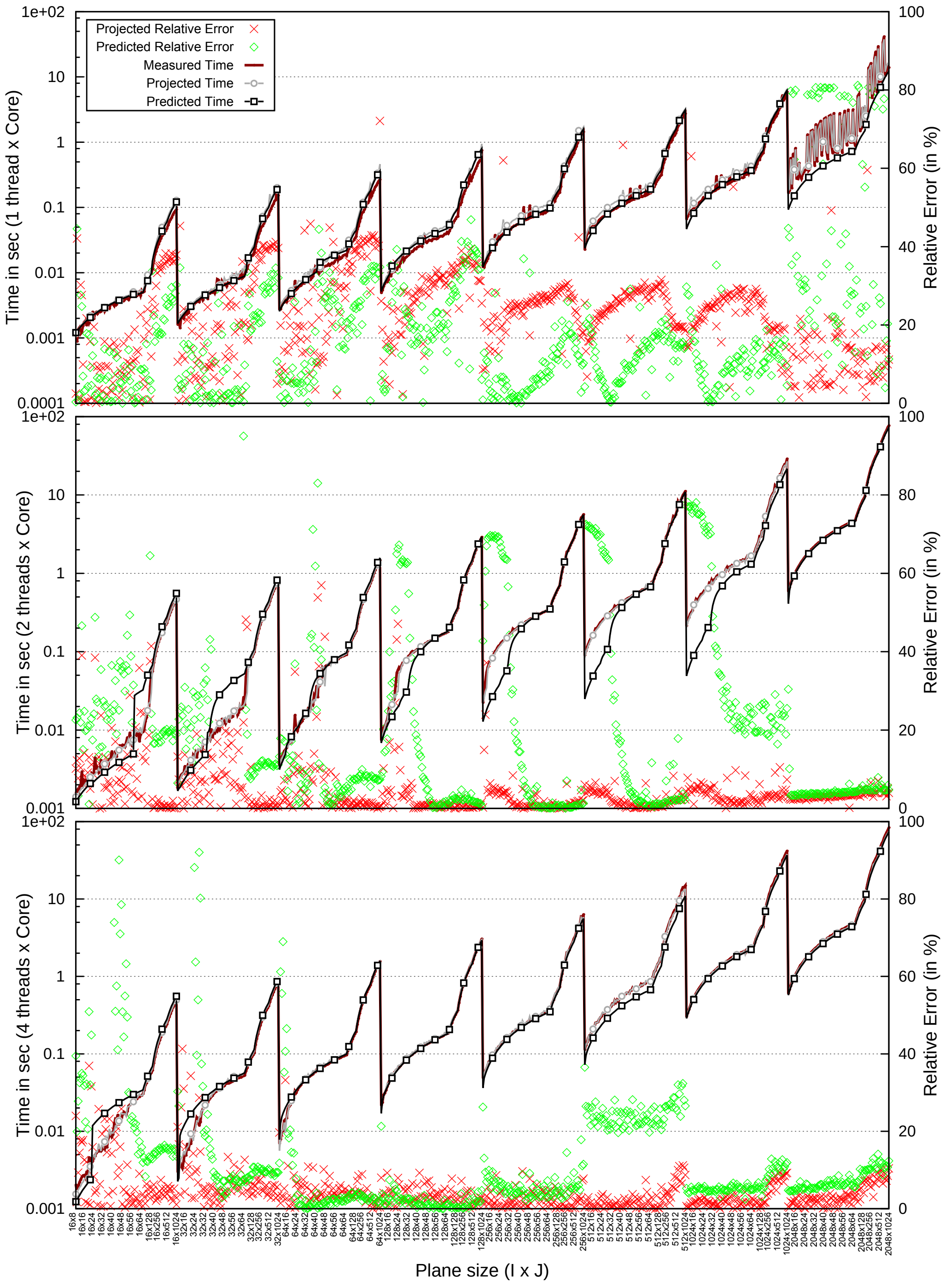


Figure 6.12: Left axis: actual (solid line), projected (circles) and predicted (squares) execution times for the three SMT configurations. Right axis: relative errors compared with actual times. These results are for a high order ($\ell = 7$) Naive stencil.

(not shown here), this error is owing to a late deactivation of the L2 prefetching engine, misleading the aggregated predicted time. Once the prefetching efficiency is correctly predicted again, the relative error drops considerably under 10%. Equally, some actual executions also present peaks due to the *ping-pong* effect. Projected times clearly follow this instabilities because their mirroring on cache misses. On the contrary, our model can not mimic such situations, and therefore, the relative error increases considerably on those cases.

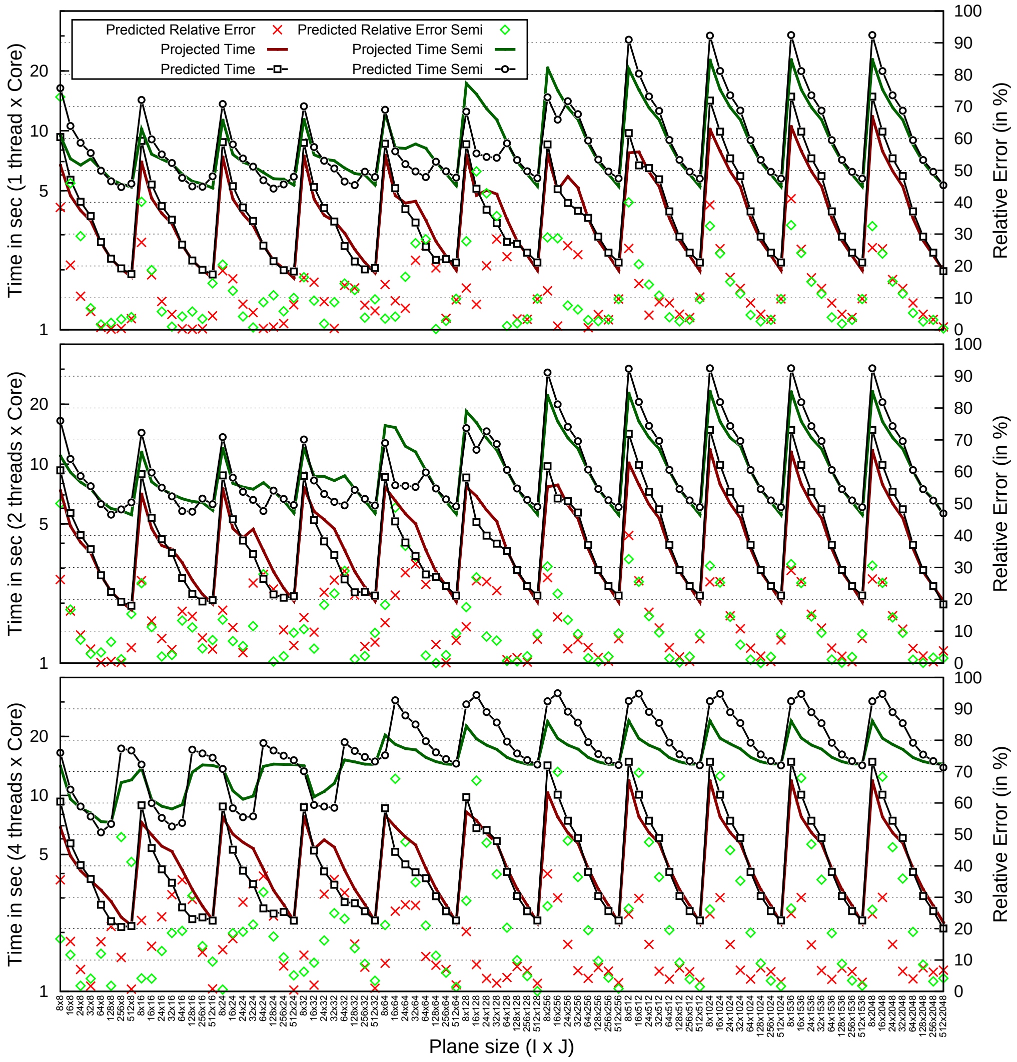


Figure 6.13: Left axis: projected (solid line) and predicted (squares and circles) execution times for spatial blocking results. Right axis: relative errors compared with projected times. Results shown are for Naive ($\ell = 1$) and for Semi-stencil ($\ell = 4$).

Results considering stencil optimizations such as Semi-stencil and spatial blocking are shown in Figure 6.13. In this test, 88 different tiling sizes were compared. The TP and LAP parameters used for the model were set to 3 and 5 cache-lines respectively. These values were obtained empirically using the *Prefetching* benchmark as explained in Section 6.6. As shown in Figure 6.13, the model clearly estimates the different valleys (local minima) that appear when searching for the best tiling parameters due to the disruption of prefetched data and the increase of cache-line misses. The model is even able to suggest some good parameter candidates. For instance, taking a look to the *Naive+Blocking* results, the model predicts successfully the best tiling parameter for 1 and 2 threads per core configurations (512×16 and 512×8 respectively). This is not the case when running 4 threads per core. However, in this latter case, the actual best parameter is given as third candidate (512×8). On the other hand, reviewing the *Semi+Blocking* results, despite of some mispredictions, especially for 4 threads per core, most of the local minima areas are well predicted.

Additionally, the model can reveal other insightful hints regarding the efficiency in SMT executions. It can help to decide the best SMT configuration to be conducted in terms of core efficiency. Let τ^{SMT_i} be the execution time for a SMT_i configuration of n different combinations. We define the *core efficiency* as

$$Core^{SMT_i}_{efficiency} = \frac{\min(\tau^{SMT_1}, \ldots, \tau^{SMT_n})}{\tau^{SMT_i}} \in [0,1] \,, \qquad (6.18)$$

where a *core efficiency* of 1 represents the best performance-wise SMT configuration for a set of specific stencil parameters (ℓ, $I \times J$ plane size, spatial blocking, Semi-stencil, etc.) and a given architecture. Therefore, the desirable decision would be to run the stencil code using the SMT_i configuration that maximizes the *core efficiency*. Normalizing our experiments for all three SMT combinations on a Naive stencil ($\ell = 4$) the Figure 6.14 is obtained. Note that depending on the problem size, the best SMT configuration ranges from 4 threads for small sizes to 2 threads for medium sizes and just only 1 thread per core for very large problems. The factor leading to this behavior is the contention of shared resources, especially the prefetching engine.

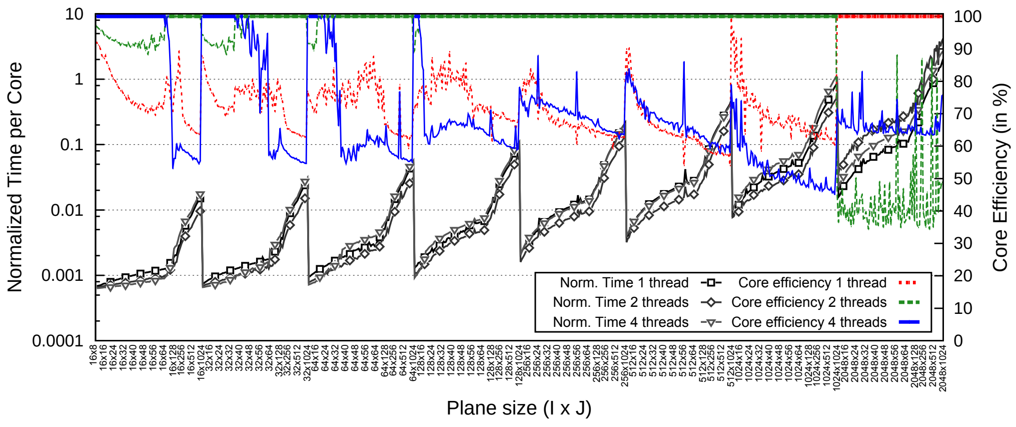


Figure 6.14: Core efficiency for all three SMT combinations using a naive stencil ($\ell = 4$).

6.8 Summary

This chapter presents a thorough methodology to evaluate and predict stencil codes performance on complex HPC architectures. The aim of this research was to develop a performance model with minimal architectural parameter dependency (flexible) and at the same time able to report accurate results (reliable). In this regard, we have obtained fairly good prediction results, where the average error for most relevant cases floats between 5-15%. All these results factored in cache's associativities, TLB page size or complex prefetching engine specifications without explicitly modeling them. However, the final goal of this research is not only to devise an accurate and flexible model, but also to freely experiment with platform parameters, which in turn, would be helpful to forecast the performance of current and future platforms as well.

Chapter 7

Case Studies

Many relevant problems arising in Geoscience and Computational Fluid Dynamics (CFD) can be solved numerically using Finite Difference (FD) methods on structured computational meshes. Application areas include, seismic wave propagation, weather prediction or atmospheric transport. FD numerical schemes on structured meshes allow peak performances of at least $\approx 20 - 30\%$, about 3 times larger than analogous FE (Finite Element) methods.

In this chapter, several of the techniques presented in this thesis are implemented alone or combined together into two real-life applications used nowadays. The challenge is to assess and demonstrate that the reviewed optimization techniques can be successfully deployed as feasible solutions to improve the performance of real cases for the industry.

First, we review an optimization of the RTM model, a method used in Oil & Gas industry, using the Semi-stencil algorithm on the IBM Cell B.E. architecture. We take the code developed at the Kaleidoscope Project [13], a partnership of Repsol and BSC to develop HPC tools for seismic imaging, as the infrastructure to optimize. Next, we present the WARIS framework, a brand-new multi-purpose framework aimed at solving scientific computing problems using FD; and in which the work of this thesis has partially collaborated. The initial design requirements were: development of a portable framework (*i.e.* able to run on any hardware platform) suited for accelerated-based architectures, with reusable software components, easily extendible, and able to solve the physical problems on structured meshes explicitly, implicitly or semi-implicitly. Finally, we present an application example using WARIS framework, the WARIS-Transport module, entirely developed in this thesis. This module is a porting of the FALL3D code [2], a multi-scale parallel Eulerian transport model that simulates the volcanic ash dispersal. The FALL3D model has a worldwide community of users, such as Volcanic Ash Advisory Centers (VAACs), and several applications, including short-term dispersion forecasts of hazardous substances, long-term hazard assessments, air quality evaluations or climate studies.

7.1 Oil & Gas Industry

In the search for new reserves, oil companies are turning to complex geological structures so far left unexplored due to their inherent difficulty to be prospected. The most prominent examples are reserves located under salt domes, like in the offshore of the U.S. Gulf of Mexico.

These reserves are estimated to hold more than three thousands sub-salt oil pools, accounting for 37 billion barrels of undiscovered recoverable oil, and 191 trillion cubic feet of gas reserves [49]. Also, the discovery of a Brazilian deep-water sub-salt exploration area was already announced, which may contain as many as 33 billion oil barrels [64].

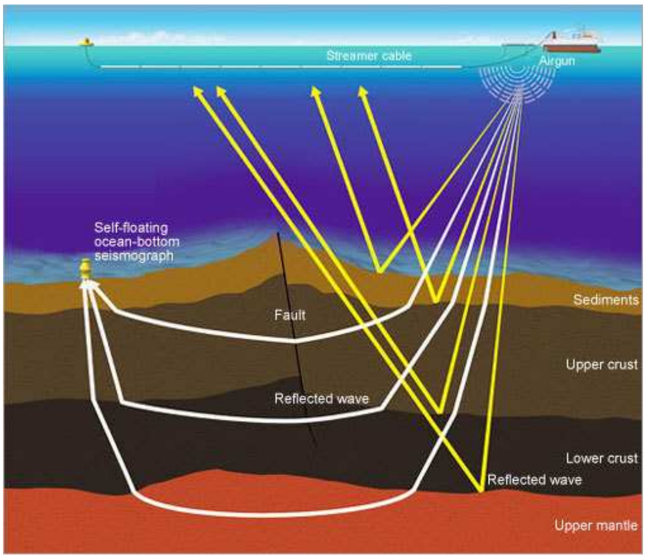


Figure 7.1: Off-shore oil and gas prospection. Ships are equipped with airguns and geophones to convey acoustic analysis.

Oil discovery is a theoretical, algorithmic and computational challenge. Prospection is primarily performed with acoustic depth-imaging techniques: an intense acoustic signal is directed into the ground, and receivers record the signal's echoes. In off-shore campaigns (see Figure 7.1), a prospecting ship is used to gather this data. These ships are equipped with an array of airguns (sources) that produce acoustic waves whose echoes are recorded by geophones (receivers). Receivers are arranged in a grid next to the airgun array. Then using the recorded information of the receivers, wave field reconstruction methods are used to solve equations which govern the propagation of acoustic waves through the Earth. These methods aim to determine densities and shapes of the subsurface structures, thus finding and assessing possible areas with oil & gas reservoirs. Due to the multi-million dollar cost of drilling operations, oil companies trust in wave field reconstructions methods for making decisions about when and where to conduct drilling campaigns.

Two imaging methods dominate the arena: One-way Wave Equation Migration (WEM) and Reverse-Time Migration (RTM) [58]. WEM is very popular thanks to its lower computational cost and its acceptable degree of accuracy in traditional scenarios. RTM is more accurate, but its computational cost, which is at least one order of magnitude higher than WEM, hinders its adoption. Nevertheless, in scenarios with reserves located beneath salt where large velocity contrasts or steeply dipping formations arise, RTM overcomes WEM's

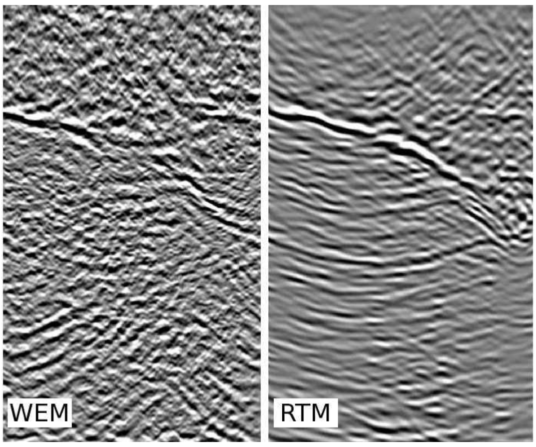


Figure 7.2: Comparison between WEM and RTM results. RTM provides higher quality images, with better signal-to-noise ratio and clearer structure delineation.

image quality. In fact, RTM can handle events that propagate along both directions of the depth axis, whereas WEM cannot. As a consequence, WEM performs poorly with structures, like salt flanks, that are illuminated by overturned reflections. Figure 7.2 presents a visual comparison of the results provided by both methods.

7.1.1 RTM Overview

In this section, the RTM method is briefly presented, unveiling the algorithm and identifying its main computational kernels that are our optimization targets.

The purpose of RTM is to generate the image of geological mediums, for instance multi-mile-deep volumes of subsea geology. The RTM inputs consists of: an initial version of the medium to be studied, a wavelet (provided by airguns), and a set of recorded acoustic wave pressure traces (provided by geophones).

RTM simulates mathematically the propagation of sound in the given medium. In this simulation, first the medium is excited by introducing the wavelet from airguns (namely shot), expressed as a function of frequency and time. Then, the wave propagation is simulated forward by using an acoustic wave equation. Next, this operation is repeated in a backward fashion, but starting from the data recorded by the receivers (geophones) and propagating the wave field back in time. Finally, when both acoustic fields representing the *forward* and *backward propagations* are available, a cross-correlation is performed in order to generate the output image of the subsurface.

Algorithm 15 shows the pseudo-code of the RTM method. The forward and backward propagation steps have been separated for the sake of clarity. Both propagation codes consist of four steps: the computation of the wave equation, the injection of sources, the absorbing boundary conditions and the I/O operations. The first step (lines 4 and 20) contains the PDE solver in charge of solving explicitly the acoustic wave propagation, which usually de-

Algorithm 15 The RTM pseudo-code. Forward-Propagation, input: medium (u, v), shots (source wavelets) and α; output: forward wavelet. Backward-Propagation, input: medium (u, v), receiver' traces, forward wavelet and α, output: correlated image.

```
 1: procedure Forward-Propagation
 2:     for $t = 0$ to $t_{end}$ do
 3:         for all grid points do                    ▷ Stencil
 4:             Wave-Propagator
 5:         end for
 6:         for all sources locations do              ▷ Source
 7:             Add source wavelet
 8:         end for
 9:         for all absorbing points do               ▷ ABC
10:             Apply absorption conditions
11:         end for
12:         if $t$ is multiple of $\alpha$ then       ▷ I/O
13:             Save forward wave field
14:         end if
15:     end for
16: end procedure
17: procedure Backward-Propagation
18:     for $t = 0$ to $t_{end}$ do
19:         for all grid points do                    ▷ Stencil
20:             Wave-Propagator
21:         end for
22:         for all receivers location do             ▷ Source
23:             Add receivers data
24:         end for
25:         for all absorbing area points do          ▷ ABC
26:             Apply absorption conditions
27:         end for
28:         if $t$ is multiple of $\alpha$ then       ▷ I/O
29:             Recall wave field saved at $t$
30:             for all grid points do
31:                 Correlate wave fields
32:             end for
33:         end if
34:     end for
35: end procedure
```

mands the most part of the execution time. The second step (lines 7 and 23) introduces the source wavelet and the receivers' traces respectively. Step three (lines 10 and 26) applies the absorbing boundary conditions in order to contain the physical phenomena in a finite computational domain. Finally, step four (lines 13 and 29) saves the wave field in the forward propagation to perform later the correlation image with respect to the backward wave field. This step is only performed once every α time-steps, thus storing the RTM output (correlated image) between the two wave fields saved at the same time iteration. The choice of α will determine the accuracy of the final image. Unlike step one, the remaining three steps should not require significant execution time.

The acoustic wave propagation equation (see Appendix A.2) is represented by PDEs. This equation is solved in the RTM by FD using an explicit method and therefore a stencil operator. Assuming an isotropic, non-elastic medium, where density is not variable, an 8th-order cross-shape discretization in space ($\mathcal{X}^{t-1}_{i\pm1..4,j\pm1..4,k\pm1..4}$) and a 2nd-order backward difference discretization in time ($\mathcal{X}^{t}$, $\mathcal{X}^{t-1}$ and $\mathcal{X}^{t-2}$) are employed. Thus, the PDE solver involves a 3D 25-point stencil computation in a co-located mesh.

Algorithm 16 details the pseudo-code of the PDE solver that can be found as computation step in both kernels propagators. The loop structure traverses, in a naive fashion, the computational domain (a 3D structured mesh) that represents the subsea area to explore. The explicit solver is composed of two parts. Line 5 computes the stencil operator (spatial integrator) for each point, whereas lines 6 and 7 perform the time integration of the PDE solver. Due to the isotropic medium, the stencil can be computed as a Laplacian operator rather than as

Algorithm 16 Pseudo-code of the PDE solver for the 3D wave equation in the RTM. u^t is the space domain for time-step t, v is the velocity field of the media u, and Z, X, Y are the dimensions of the dataset (ordered from unit to least-stride) including ghost points. $C_{Z1 \ldots Z4}$, $C_{X1 \ldots X4}$, $C_{Y1 \ldots Y4}$ and C_0 are the spatial discretization coefficients for each dimension and Δt is the temporal discretization parameter. Integrating the equation requires maintaining the wave field of at least two earlier time-steps (u^{t-1} and u^{t-2}).

```
1: procedure Wave-Propagator($u^t, u^{t-1}, u^{t-2}, v, Z, X, Y, \Delta t, C_0, C_{Z1 \ldots Z4}, C_{X1 \ldots X4}, C_{Y1 \ldots Y4}$)
2:     for $k = 4$ to $Y - 4$ do
3:         for $j = 4$ to $X - 4$ do
4:             for $i = 4$ to $Z - 4$ do                                  ▷ Spatial integration (Stencil)
5:                 $u^t_{i,j,k} = C_0 * u^{t-1}_{i,j,k}$
                       $+ C_{Z1} * (u^{t-1}_{i-1,j,k} + u^{t-1}_{i+1,j,k}) + \ldots + C_{Z4} * (u^{t-1}_{i-4,j,k} + u^{t-1}_{i+4,j,k})$
                       $+ C_{X1} * (u^{t-1}_{i,j-1,k} + u^{t-1}_{i,j+1,k}) + \ldots + C_{X4} * (u^{t-1}_{i,j-4,k} + u^{t-1}_{i,j+4,k})$
                       $+ C_{Y1} * (u^{t-1}_{i,j,k-1} + u^{t-1}_{i,j,k+1}) + \ldots + C_{Y4} * (u^{t-1}_{i,j,k-4} + u^{t-1}_{i,j,k+4})$
                                                                           ▷ Temporal integration
6:                 $u^t_{i,j,k} = v^2_{i,j,k} * u^t_{i,j,k}$
7:                 $u^t_{i,j,k} = \Delta t^2 * u^t_{i,j,k} + 2 * u^{t-1}_{i,j,k} - u^{t-2}_{i,j,k}$
8:             end for
9:         end for
10:    end for
11: end procedure
```

a gradient followed by a divergence.

Step	Forward	Backward
Propagation	91.2%	70.2%
Source (shot, receivers)	1.3%	2.7%
Boundary conditions	7.5%	6.2%
Cross-correlation	N/A	20.9%

Table 7.1: Execution time breakdown of the RTM workload. A synthetic benchmark was used for this results using a naive RTM implementation.

We have benchmarked a sequential, unoptimized implementation of the RTM code in order to determine the relative contribution to execution time of the different workload portions. The results are reported in Table 7.1. The benchmark shows that the workload is clearly dominated by the FD solver (line 4 in forward and line 20 in backward propagators of Algorithm 15). The high computational cost of the propagator kernels is due to the elevated number of memory accesses and the access pattern that they require to conduct the stencil, turning the RTM code in a memory-bound problem.

As shown in previous chapters, the memory-bound limiting factor requires careful design of how data is accessed in the memory hierarchy. In this regard, explicit techniques to promote data locality (spatial and temporal) can be applied. Actually, since the stencil of the PDE solver uses a 25-point stencil ($\ell = 4$) is an appropriate stencil candidate to apply the *Semi-stencil* algorithm. Thus, the following sections are devoted to present an RTM kernel implementation where the *Semi-stencil* optimization has been applied on the Cell/B.E. architecture.

7.1.2 Semi-stencil Implementation in Cell/B.E.

We present a Semi-stencil implementation of the RTM computational kernel deployed for the Kaleidoscope Project. This industrial code is specifically designed to exploit the architectural characteristics of the Cell/B.E. The performance of the Semi-stencil algorithm is evaluated in two steps: as a single-SPE implementation, and then integrating it later in a fully operational multi-SPE version of the RTM numerical kernel. The purpose of the first implementation is to show the local behavior of the algorithm, whereas the second implementation purpose is twofold: to measure the expected positive impact of the Semi-stencil on a real application, and to expose the impact of the Cell/B.E. architecture on the RTM performance. Although, the RTM code of the Kaleidoscope Project is prepared to be run on a rack of IBM BladeCenter QS22 nodes through MPI, we have specifically focused on the improvement achieved on one single node. Therefore, the scalability analysis by means of MPI is out of scope.

General Cell/B.E. Optimization Considerations

As we have reviewed in Chapter 4, there are many optimization techniques that deal with stencil computations. They have been successfully used in academic benchmarks for low-order stencils. Spatial-blocking may enhance scientific applications, but time-blocking methods, due to their inherent time dependency, pose integration problems with boundary conditions, communication, and I/O tasks that are commonly incorporated in real applications. However, at SPE level, where the kernel optimization must be performed, spatial-blocking is not longer useful due to the constant access cost of the scratchpad memory (LS), which behaves as a very large register pool (256 KB). For these reasons, we discarded all these optimization techniques, thus implementing classical and Semi-stencil algorithms together with pipeline optimizations. In addition, other characteristics that must be carefully taken into consideration for the Cell/B.E. porting are: the limited LS size, the appropriate use of SIMD instructions, and the explicit coherence management between LS and main memory.

The RTM workload has been parallelized into loosely coupled threads to exploit the multiple independent processing elements (SPEs), orchestrating the data transfers to ensure the most efficient memory bandwidth utilization. As a consequence of the limited size of each private LS, it is necessary to decompose data for parallel processing. This kind of tiling has to respect the LS size constraint. In order to do so, the parallelization strategy decomposes the 3D domain along the X axis as shown in Figure 7.3, where each SPE processes a distinct $Z \times X_{SPE} \times Y$ sub-cube. As sub-cubes are much larger than the available space in the LS, the domain is traversed as a streaming of Z-X planes through the Y axis, keeping in LS only as many planes as required to compute the stencil of one plane. Additionally, to achieve an efficient scheduling of data transfers between main memory and LS with computation, we

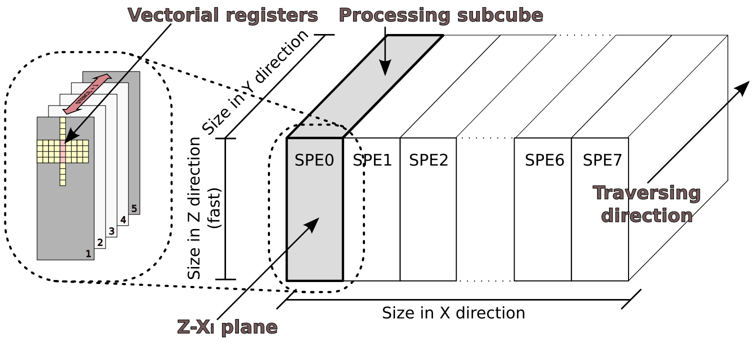


Figure 7.3: Decomposition strategy on Cell/B.E. of 3D domain along the X axis.

implement the double-buffering technique. This technique allows the overlapping of data transfers and computation, completely hiding the shorter of the two latencies (usually the computation).

Another main concern to get the most benefit from Cell/B.E. is to use the 128-bit wide SIMD instruction set in SPEs. Considering the large number of SIMD registers (128), it is possible to optimize the code to compute simultaneously up to 20 data points, which entails to a further performance gain. This large register pool endorses the use of techniques such as software pipelining and loop transformations, thus removing possible pipeline stalls and achieving a balanced pipeline.

Single-SPE Implementation

In the first porting of the Semi-stencil algorithm to Cell/B.E., we focus only in the SPE's code. The PPE code and the techniques required to make the main memory transfers efficient are exclusively considered in the multi-SPE implementation. As discussed before, our classical and Semi-stencil implementations were vectorized by hand, taking advantage of pipeline optimizations such as: software prefetching, software pipelining and loop unrolling.

The following results are expressed in terms of elapsed time and floating-point operations. Notice that the experiments were conducted in single precision arithmetic, and using a stencil size (ℓ) of 4. The intent of this experiment is to compare both implementation approaches: the classical stencil and the Semi-stencil algorithm, evaluating their results with respect to the peak performance of the Cell/B.E. architecture.

For this experiment, a simple synthetic benchmark was established where the whole 3D dataset and the instruction code were completely allocated in the LS of the SPE (in place). Considering the 256 KB available in the scratchpad memory, the maximum attainable size was a domain of 28^3 elements. This domain required a total of 172 KB using single precision (4 bytes) for source and destination buffers of the simulation (u^t and u^{t-1}). The computation of the spatial operator needed a total of 37 Flops per point (see Algorithm 16, line 5).

The peak performance achieved by the Semi-stencil implementation is 12.44 GFlops (Ta-

	Classical Stencil		Semi-stencil	
Compiler (Optimization)	Time [ms]	Performance [GFlops]	Time [ms]	Performance [GFlops]
XLC -O3	6.84	4.32	3.35	8.83
XLC -O5	3.44	8.61	2.38	**12.44**
GCC -O3	7.83	3.78	4.57	6.47

Table 7.2: Performance results of the single-SPE implementation. These results were obtained with *IBM XL C/C++ for Multi-core Acceleration for Linux, V9.0* and GCC 4.1.1.

ble 7.2), which corresponds to the 49% of the SPE peak performance (25 GFlops per SPE in single-precision). Under the same experimental setup, the classical stencil reaches only 8.61 GFlops (34% of the SPE peak performance). This means that the Semi-stencil algorithm is 44% faster than the classical stencil. The projected-aggregated performance of this algorithm is 99.52 GFlops for one Cell/B.E. processor, which is, to our knowledge, the highest peak performance for stencil computations on this architecture [22, 84].

	Classical Stencil	Semi-stencil	Gain
Total cycle count	11460147	7592643	33.7%
CPI [cycles/instructions]	0.69	0.75	-9%
Load/Store instructions	4236607	2618079	38.2%
Floating point instructions	5800000	4652400	19.8%
No-operations (NOPs)	1124903	442484	60.7%
FP/Cache ratio	1.36	1.78	24.4%
Shuffle instructions	1220001	1072401	12.1%

Table 7.3: Pipeline statistics of the single-SPE implementations. Obtained with the IBM Full-System Simulator

In order to extract accurate and insightful information about how the Semi-stencil behaves in terms of pipeline execution, we collected performance statistics of the SPE through the *IBM Full-System Simulator for the Cell Broadband Engine*. The SPEs have an in-order execution model, and the latencies of their execution units are deterministic. As a consequence, the SPE execution work-flow can be accurately simulated using the binary files. The Cell/B.E. simulator provided extremely detailed analysis on a cycle-for-cycle basis of the current state of a simulated SPE. Table 7.3 summarizes all these results comparing the classical implementation versus the Semi-stencil one. In Table 7.3 every metric is in Semi-stencil favor, except the CPI measure, which is 9% better in the classical stencil algorithm. Among all, the most important metric and the one that summarizes the better performance is the FP/Cache ratio, which is 24% higher in the Semi-stencil. These results clearly demonstrate that the Semi-stencil algorithm is a feasible solution for optimizing medium- and high-order stencils.

The reported execution times include neither I/O time, nor non-recurring allocation nor initialization delays. They have been gathered over repeated runs, to eliminate spurious

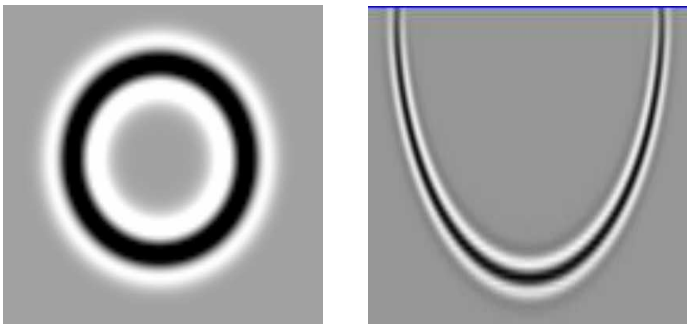


Figure 7.4: The forward wave propagation of single-SPE implementation for one shot and one receiver in a constant-velocity medium. Left: propagation after one time-step (X-Y cut). Right: final result after several time-steps (Z-X cut).

artifacts (*e.g.* bus traffic, or unpredictable operating system events). Figure 7.4 shows the dumped images of the wave propagation simulation conducted for the single-SPE experiment with a constant velocity field and one shot and one receiver.

Multi-SPE Implementation

In this second version, we integrate the Semi-stencil algorithm with a fully operational Reverse-Time Migration (RTM) implementation [9]. This version incorporates the domain decomposition strategy discussed before, where a streaming of Z-X_{SPE} planes is employed so as to compute the whole 3D problem. This experiment was conducted using the SEG/EAGE salt test (see Figure 7.5), a synthetic 3D model used broadly by the research community, which was run on a 512^3 dataset over 500 time-step iterations. In this implementation, the most optimal Z-X_{SPE} plane size in terms of transference efficiency and LS constraint must be previously computed by the PPE. Then, each SPE traverses, in a round-robin fashion, sub-cubes of $Z \times X_{SPE} \times Y$ points until the whole domain is computed.

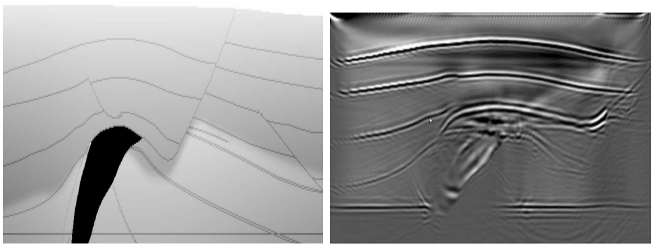


Figure 7.5: RTM output of the SEG salt synthetic model. Left: velocity field used as input. Right: correlated output image after forward and backward propagations.

Table 7.4 summarizes all the final results for both implementations, the classical and the Semi-stencil algorithms. The results are presented considering both the overlap of computation and communication tasks (*Total*) and only one of them (*Only Computation* and *Only*

Communication). The latter columns were obtained by disregarding FLOPs operations and DMA transactions respectively. The intent of these two columns is to verify which part of the SPE code dominates the execution time.

	Total		Only Computation		Only Comm.	Imba-balance
Algorithm implementation	Time [sec]	Performance [GFlops]	Time [sec]	Performance [GFlops]	Time [sec]	%
RTM + classical	74.24	39.15	71.92	41.06	70.33	2.3
RTM + Semi	63.33	**46.62**	48.13	61.35	64.62	**25.6**

Table 7.4: Performance results of the RTM implementation. These results were obtained using only one Cell/B.E. of a QS22 blade. Total time experiments cover both computation and communication times.

Reviewing the results, the RTM + Semi-stencil version is 16% faster than the RTM+classical, which partially keeps the performance benefit presented in the single-SPE experiments. There are two main reasons of this degradation in performance. First, the new tasks that must be conducted in order to perform the full RTM implementation, e.g. boundary conditions, I/O, etc. Second, and above all others, the data transfers of the streaming strategy, which in turn results to be the main bottleneck. As shown in *Only Computation* and *Only Communication* columns of Table 7.4, the imbalance between data transfers and computation stagnates the overall performance. This imbalance is specially noticeable in the RTM+Semi-stencil version, where the lost of performance is 25.6%, even using multi-buffering technique. If this 25.6% is added to the already gained 16%, we recover the 44% of advantage of the Semi-stencil over the classical stencil algorithm shown in the single-SPE implementation.

7.2 WARIS Framework

WARIS is a Barcelona Supercomputing Center (BSC) in-house multi-purpose framework aimed at solving scientific problems using Finite Difference (FD) methods as numerical scheme. The framework was designed from scratch to solve, in a parallel and efficient way, Earth Science and Computational Fluid Dynamic problems on a wide variety of architectures. WARIS uses structured meshes to discretize the problem domains, as these are better suited for optimization in accelerator-based architectures. To succeed in such challenge, WARIS framework was initially designed to be modular in order to ease development cycles, portability, reusability and future extensions of the framework. The following section details the design of the WARIS framework and its basic internals.

7.2.1 System Architecture

With the large number of physical problems to be supported by the WARIS framework, a solution composed of two primary components was devised. The two components, so-called Physical Simulator Kernel (PSK) and Workflow Manager (WM), are depicted in green and blue respectively in Figure 7.6. The PSK framework is responsible for those tasks that are common to any physical simulation being solved, such as domain decomposition, communications and I/O operations. Its aim is to provide a base for the specialization of physical problems (*e.g.* heat, wave, ADR or Navier-Stokes equations) on any forthcoming architecture. The PSK component is basically a template that provides the appropriate framework for implementing a specific simulator. Flexibility in design must be attained to let the specialization accommodate any type of physics on any kind of hardware reusing as much code as possible. This approach minimizes the development cycle by reducing the code size and the debugging efforts. The *specializations* (depicted in red in Figure 7.6) are used to configure the framework in order to have a complete solution for a given physical problem. Those specializations depend on aspects such as: the physical problem, the hardware platform (*e.g.* general purpose, GPU, FPGA, Xeon Phi) and the programming model being used for development.

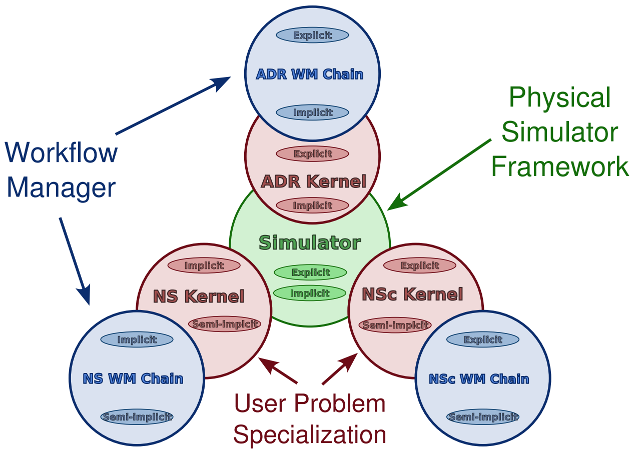


Figure 7.6: WARIS System. It is composed of three components: the Physical Simulator Kernel (PSK), the user-specialization of the problem and the Workflow Manager (WM).

On the other hand, the WM is in charge of providing a framework that allows to process several physical problems in parallel. This framework includes all the necessary components to provide a distributed application in the sense that is capable of processing independent problems using different computational nodes. Therefore, the WM can take advantage of the massive parallel approach of running multiple physical problems in parallel. This approach can be useful to tackle a statistical study or a search of an optimal parameter for any given physical problem. The WM implements a Master-Worker pattern. The Master manages, schedules and commands a set of Workers (user specializations that use PSK framework), allocated on computational nodes, by assigning them new tasks. Workers' executions can include a kernel computation or a specific data post-process step. Within this approach, the user must not take care of resilience, fault tolerance, postprocessing and checkpointing management.

Hardware Architecture Model

A wide variety of hardware architectures is emerging during the last years, from general purpose processors in multi-core and many-core chips (*e.g.* Intel Xeon) to accelerator-based devices with outstanding performance (*e.g.* Intel MIC or General-Purpose computing on Graphics Processing Units, GPGPUs). In order to build a framework able to accommodate with ease any of these architectures, one must ensure an abstraction level of the computational architecture model. To this end, the WARIS framework was designed to include an architecture model with a main entity called *Computational Node* (CN), shown in Figure 7.7. This main entity is built using two computational resources: the *host* and the *device* that communicate each other through a *Common Address Space* (CAS) memory. The *host* resource is responsible for the parallel simulation processes, such as the load balancing (domain decomposition), data communication (exchange of boundary nodes with neighbor domains) and I/O operations. The *device* resource is composed of a set of specialization routines that are used to configure the framework in order to have a functional simulator. The specialization depends on many aspects, such as the physical problem at stack, the hardware platform and the specific numerical method. Within this architecture model, the WARIS framework is executed in the *host* while the *device* deals with the specifics of the physical problem being simulated. Notice that, in the particular case where the accelerator-based *device* is not present, the *host* will be running both, the framework and the specialization code. If the amount of resources in a single computational node are not sufficient for solving a physical simulation, more nodes will be allocated and the communications will be managed by the PSK.

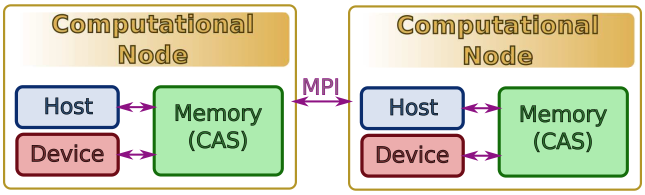


Figure 7.7: Architecture model supported by the PSK.

The domain decomposition is performed in the PSK framework at two different levels: intra-node (node-level) and inter-node (cluster-level). The former level provides decomposition within a CN by means of shared memory, allowing to efficiently exploit platforms with multi-card configuration (*e.g.* GPUs and MICs). As the memory address space across different CNs is disjoint, the latter level decomposes by means of MPI API, adding support for distributed memory implementations of the physical problems. Due to performance reasons, PSK conducts domain decompositions only along the least-stride dimension of the domain

(Y axis in a 3D problem where the dimension-ordered from unit-stride to least-stride dimension is Z-X-Y). This Y axis decomposition minimizes the gather operations (copies) of data that must be transferred to neighbor domains thanks to the unit-stride layout. Figure 7.8 illustrates this two-level decomposition. To conduct the inter-node communication tasks, the internal nodes adjacent to the boundary nodes are exchanged across neighbors using two communication steps: *Front* and *Back* (shown in the example of Fig. 7.8 for sub-domains 0-2/1-0 and 1-2/2-0). This has the advantage of faster data transference but, in contrast, impedes optimal scalability.

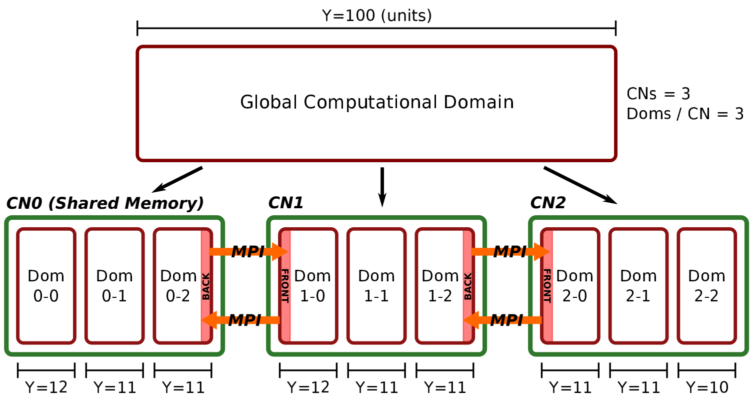


Figure 7.8: Domain decomposition conducted by the PSK infraestructure. Two levels of decomposition are used in this example: intra-node with 3 subdomains within each CN (domains *-0 to *-2) and inter-node with 3 domains (referenced as 0-* to 2-*).

Finally, in order to run the physical simulation in a efficient and concurrent way, each CN spawns a set of independent execution flows by means of POSIX threads that are in charge of specific tasks. These threads can be classified as:

- *Main thread*: each CN creates a *Main thread* in charge of orchestrating and commanding the remaining threads spawned within the CN. Its main tasks are reliability and robustness of the infrastructure, as well as allocation and deallocation of resources through control code. This thread also creates the intra-domains by spawning as many *Domain threads* as required by the platform specification.

- *Domain thread*: they are in charge of solving the physical problem by explicitly calling the specialization routines that involve the computational effort. Each *Domain thread* creates an *I/O thread* and as many *Communication threads* as domain neighbors (*Back* and *Front* communication steps).

- *Communication thread*: it performs asynchronous transferences of boundary nodes across neighbor domains. The appropriate parallel paradigm is selected automatically: shared memory for intra-domains, and MPI API for inter-domains (across CNs).

- *I/O thread*: this thread is exclusively in charge of performing I/O operations by means of the I/O library chosen by the user implementation (*e.g.* POSIX I/O, MPI-IO, HDF5 or NetCDF).

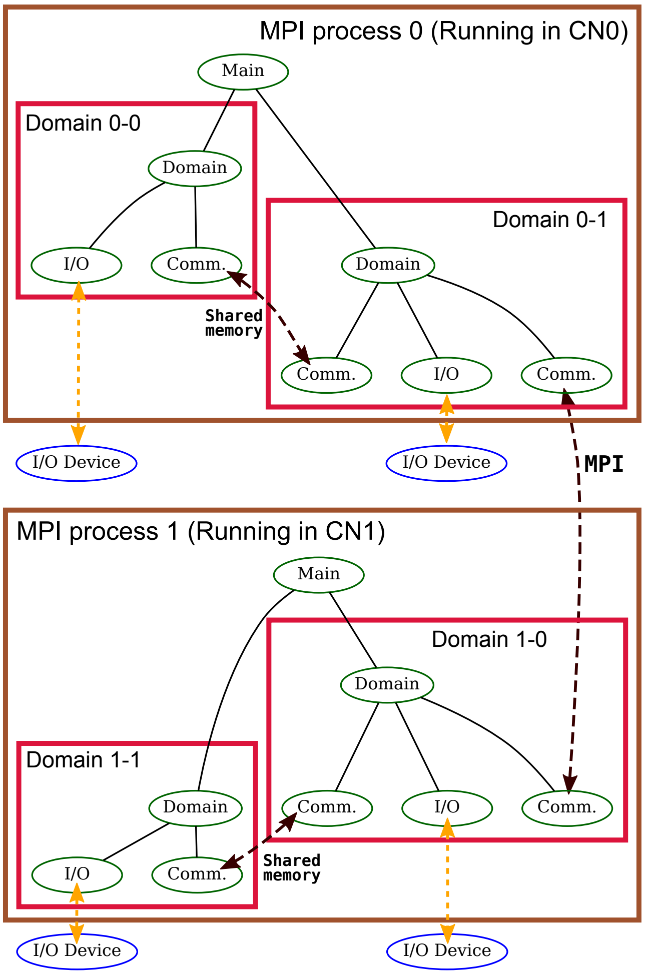


Figure 7.9: Threads spawned by the PSK framework. Example case running in two CNs, each with two intra-domains. The green and blue nodes represent POSIX threads and I/O devices respectively, whereas the brown and red boxes are the MPI processes running on a CN and the intra-node domains within a CN.

Figure 7.9 shows a case where the PSK has mapped two intra-domains per each CN. In this example, only one communication thread is required for domains assigned to sides (*Domain 0-0* and *Domain 1-1*). The hardware architecture model followed by the PSK has several advantages. First of all, the high level of parallelism that can be achieved by overlapping computation, communication, and I/O tasks using independent threads. Second, the abstraction level flexibility of the architecture model facilitates the porting to any possible architecture.

Software Architecture Model

The PSK framework provides a configurable execution flow that permit to extend or specialize it for a given physical problem. The specialization process is done by implementing a common interface defined by the PSK. Each of the functions to be implemented by the user are known as a *specialized functions*, and they must conform this common interface. Among these functions, the PSK defines: initialization and finalization routines for managing data structures of the physical problem, proper functions for the simulation processing at each iteration, and functions for scattering and gathering data across domains. Figure 7.10 shows the execution flow provided by the PSK framework. The dashed and red boxes represent the functions to be provided by the user in order to specialize the framework for a specific physical problem.

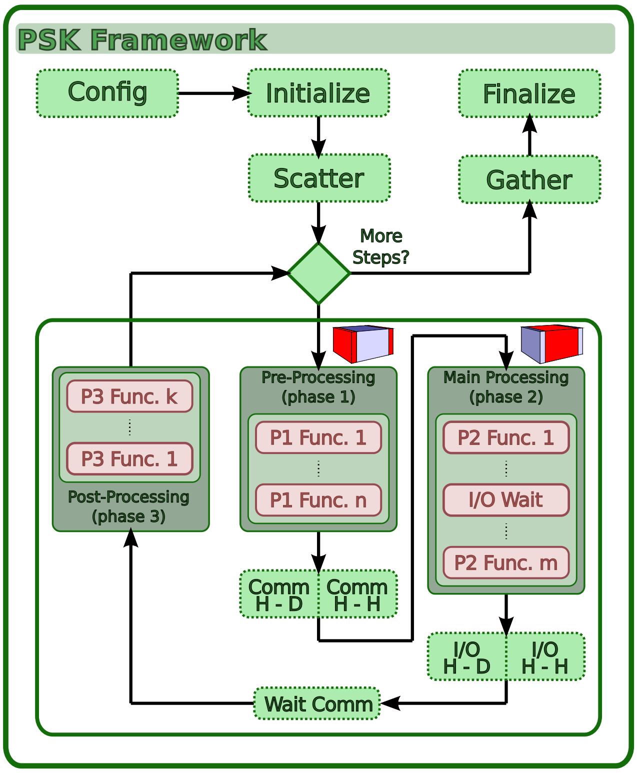


Figure 7.10: Execution flow of the PSK framework.

The execution flow inside the main loop of the PSK framework is divided in three different phases, known as Pre- Main- and Post-Processing phases, which are separated by stages that perform communication and I/O tasks. The aim is to provide an environment capable of overlapping simultaneously computation, communication and I/O through the Pthreads created by the PSK. In general terms, the stages that compose the PSK execution flow are the following:

- *Config*: it configures the internal state (*e.g.* spatial dimensions and specialization parameters) and the internal behavior (*e.g.* P1, P2 and P3 functions).
- *Initialize*: it initializes the main structures at *host* and *device* levels needed for the simulation.
- *Scatter*: as the framework is intended for multi-node, multi-domain propagation, a data scattering process may be needed before the simulation starts.
- *Pre-processing (P1)*: it is responsible for executing a chain of functions (configured by means of the *Config* module) in Phase 1. The configured functions should conduct the computation of data that will be exchanged later.
- *Main Processing (P2)*: it calculates everything else that needs to be calculated in the propagation, including anything related to I/O operations (data preparation and waiting).
- *Post-Processing (P3)*: it post-processes any output generated after P1 and P2 phases.
- *Communication H-D/H-H*: it commands the *Communication thread* to perform asynchronous communication across neighbors, first moving data between *device* and *host* (*Comm H-D*) and later between hosts (*Comm H-H*). The data exchange can be performed between CNs (inter-node) and within the CN (intra-node).
- *I/O H-D/H-H*: it commands the *IO thread* to perform the I/O, first moving data between *device* and *host* (*Comm H-D*) and finally performing the I/O operation.
- *Wait Communication*: at the end of each time-step, the framework may wait for any previous communication to be finished before starting a new time iteration.
- *I/O Wait*: it sets a synchronization point during P2 phase in order to wait for any flying I/O operation.
- *Gather*: it collects any required output from the CNs in order to generate a merged output.
- *Finalize*: it clears the internal structure by deallocating specialization and PSK resources.

In order to proceed with a correct configuration, the *Phase 1* (P1) functions provided by the user must process exclusively the boundary nodes involved in the exchange stage. As soon as the P1 functions have finished, an asynchronous communication is started by the *Communication thread* while the *Phase 2* (P2) functions conduct the physical solution over the remaining nodes of the domain. Likewise, an I/O operation may be started asynchronously at the end of *Phase 2*, enabling as well the overlapping of I/O with other tasks. Finally, *Phase 3* (P3) routines can be optionally provided by the user in cases where a post-processing step is required after the computation of each time-step.

7.2.2 Optimization Module

A new support module was integrated into the WARIS framework in order to implement the optimizations and to make them available for future physical specializations (*i.e.* reusable for solving different physical problems). This module, called *opti*, provides a set of functions that users can employ to incorporate stencil-based optimizations into their explicit solvers. The optimizations in *opti* are specially tailored for general purpose processors and multi-core architectures such as the Intel Xeon family, including also accelerator-based as the many-core Intel Xeon Phi (MIC). The features supported by this module are the following:

- **Auto-tuner**. To automatize the spatial-blocking effectiveness, a simple and straightforward auto-tuner was included. This auto-tuner, based on the research of Section 5.4, is in charge of finding out possible pseudo-optimal TJ parameter (local minima) for the spatial-blocking algorithm integrated in our stencil codes. This operation is conducted during the initialization stage of each WARIS simulation. The cost of conducting this search is marginal (order of milliseconds) compared to the whole simulation time. Selecting an appropriate blocking parameter can save up to 30-40% of the explicit solver for certain simulation cases.

- **Intra-domain decomposition**. Using the research done in Section 5.3 four different schedulers were deployed for intra-domain decomposition, *static* over X and Y axes, *balanced* (based on β parameter) and *guided* (using a tiling parameter from a previous auto-tuning search as heuristic). This routine returns a structure that contains the domain boundaries for each computational thread (OpenMP).

- **Topology**. It collects the information of the underlying architecture in order to make decisions about the best intra-node decomposition strategy in shared memory architectures. This function returns the number of sockets ($N_{Sockets}$), the number of cores per socket (N_{Cores}) and the number of threads per core (N_{SMT}).

- **Thread Affinity**. Due to the multi-threaded and concurrent environment of WARIS, it is highly recommended to establish a policy of thread affinity within a node. Through this affinity, the spawned OpenMP and WARIS framework Pthreads (*Main*, *Domain*, *Communication* and *I/O* threads) are pinned to specific and different cores, avoiding memory access disruption and interferences across threads. In order to do that, hardware cores are not fully populated with OpenMP threads (in charge of intensive computing), leaving specific cores exclusively dedicated to WARIS infrastructure management (execution flow, MPI communication and I/O). This thread affinity was achieved by setting properly different environment variables: OMP_NUM_THREADS and KMP_AFFINITY to control the binding of OpenMP threads, and EAP_AFFINITY to pin WARIS framework threads to hardware cores. In some cases, this thread management led to a reduction of 5 to 15% of the global execution time.

7.3 Atmospheric Transport Modeling - Ash Dispersal

Atmospheric transport models [74] deal with transport of substances in the atmosphere, including natural (*e.g.* mineral dust, volcanic ash, aerosols, sea salt), biogenic (*e.g.* biomass burning), and anthropogenic origin (*e.g.* emission of pollutants, radionuclide leak). Model applications are multiple including, for example, short-term dispersion forecasts of hazardous substances, long-term hazard assessments, air quality evaluations or climate studies. The physics of these models describes the transport and removal mechanisms acting upon the substance and predicts its concentration depending on meteorological variables and a source term. These models are built on the Advection–Diffusion–Sedimentation (ADS) equation (see Equation A.15), derived in continuum mechanics from the general principle of mass conservation of particles within a fluid. The terms in the equation describe the advection of particles by wind, turbulent diffusion of particles, and gravitational particle sedimentation.

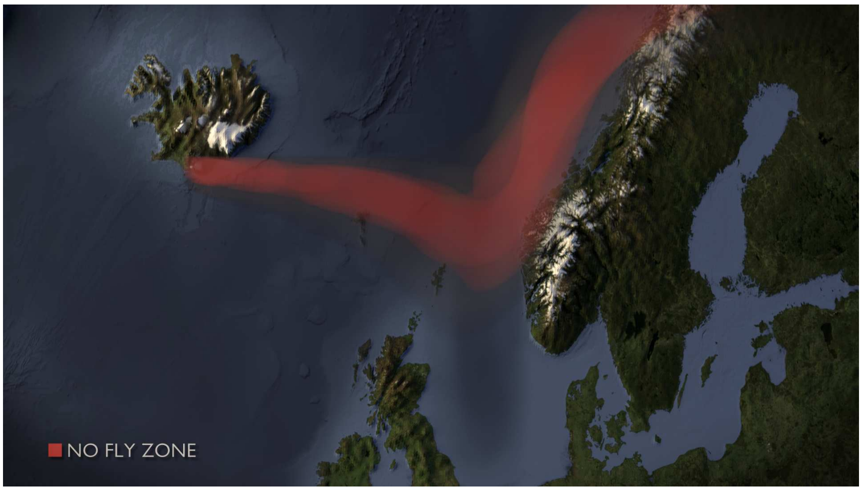


Figure 7.11: Output example of the volcanic ash dispersion model.

This transport model can be effectively used to simulate the atmospheric dispersion of volcanic ash. Volcanic ash generated during explosive volcanic eruptions can be transported by the prevailing winds thousands of kilometers downwind posing a serious threat to civil aviation. Volcanic ash models run worldwide operationally to provide advice to the civil aviation authorities, and other stakeholders, which need to react promptly in order to prevent in-flight aircraft encounters with clouds. Figure 7.11 shows a simulation example of a paradigmatic case occurred during April-May 2010, when ash clouds from the Eyjafjallajökull volcano in Iceland disrupted the European airspace for almost one week. This eruption entailed thousands of flight cancellations and millionaire economic losses [47].

7.3.1 WARIS-Transport Specialization

As an application example, the FALL3D model (see Appendix A.4) has been ported to WARIS, resulting in the so-called WARIS-Transport module. FALL3D [18, 35] is a multi-scale parallel Eulerian transport model coupled with several mesoscale and global meteorological models, including most re-analyses datasets. This model admits several parameterizations for vertical and horizontal diffusion, particle sedimentation velocities, and source term, handling a wide spectrum of particle sizes (from microns to few centimeters). Although FALL3D can be applied to simulate the transport of any substance, the application is particularly tailored for modeling the volcanic ash dispersal. FALL3D has a worldwide community of users, including the Buenos Aires VAAC (Argentina) which has an operational setup to forecast ash cloud dispersal under its area of influence.

Algorithm 17 Volcanic ash dispersion pseudo-code in WARIS-Transport. The tasks to be performed can be categorized in four different types: Kernel Computation (KC), Data Communication (DC), Input/Output (IO) and Load Balancing (LB).

```
 1: Read particle class properties (granulometry)                                  (IO)
 2: Read time and configuration variables (input)                                  (IO)
 3: Read grid data and topography from database (dbs)                              (IO)
 4: Domain decomposition of the structured mesh                                    (LB)
 5: for $t = time_{start}$ to $time_{end}$ in $\Delta t$ steps do
 6:     if $source\_time \leq t$ then
 7:         Read source term for each particle class ($S^{np}$)                    (IO)
 8:         Scale source terms ($S_*^{np}$)                                        (KC)
 9:     end if
10:     if $meteo\_time \leq t$ then
11:         Read meteorological variables ($u_{x,y,z}$, $T$, $p$, etc.) from dbs   (IO)
12:         Compute horizontal and vertical diffusions ($K_{x,y,z}$)               (KC)
13:         Compute terminal velocity for each particle ($V_{sj}$)                 (KC)
14:         Scale velocities depending on coordinate system ($U_{x,y,z}$)          (KC)
15:         Calculate the critical time-step ($\Delta t$)                          (KC)
16:         AllReduce of $\Delta t$                                                (DC)
17:     end if
18:     Set boundary conditions ($C_\Gamma$)                                       (KC)
19:     for each particle class in $np$ do
20:         Advance ADS with stencil ($C^{t+1}$)                                   (KC)
21:         Exchange overlap points between neighbor domains ($C_{ovl}$)           (DC)
22:     end for
23:     Compute ground accumulation of ash ($C_{accu}$)                            (KC)
24:     Compute mass lost at boundaries ($\nabla \cdot (UC_\Gamma)$)               (KC)
25:     AllReduce of mass lost                                                     (DC)
26:     Mass balance correction                                                    (KC)
27:     if $output\_time \leq t$ then
28:         Post-process output ($C_{thickness}$, $C_{CFL}$, $C_{PM}$)             (KC)
29:         Write output for $t$ ($C^{t+1}$, $C_{accu}$, $C_{thickness}$, $C_{CFL}$, $C_{PM}$)   (IO)
30:     end if
31: end for
```

The WARIS-Transport specialization was implemented following the Algorithm 17, which shows the execution flow required to simulate the dispersion of volcanic ash. In our implementation, each WARIS thread is in charge of different tasks. So, *Domain thread* is in charge of commanding the computational kernels (KC), *Communication thread* is involved in communicating the boundary nodes to neighbors (DC) and *IO thread* reads the meteorological data and writes the post-processed output of the ash dispersal (IO).

7.3.2 Volcanic Ash Dispersal Results

The aim of this section is to quantify how different code optimizations and porting to emerging architectures speed up the FALL3D model execution times. We also investigate to which extent WARIS-Transport can accelerate model forecasts and analyze whether the resulting improvements make feasible a future transfer of ensemble forecast strategies into operations [11].

Test case	Horizontal resolution	Vertical resolution	$n_z \times n_x \times n_y$ (ALT×LON×LAT)	Number of nodes
Caulle-0.25	0.25^o	500 m	$64 \times 121 \times 121$	0.93 M
Caulle-0.10	0.10^o	250 m	$64 \times 301 \times 301$	5.8 M
Caulle-0.05	0.05^o	250 m	$64 \times 601 \times 601$	23.1 M

Table 7.5: Domain resolutions and number of grid nodes for the 3 different cases considered in the Cordón Caulle reference simulation. Horizontal resolutions along a meridian are approximately 25, 10 and 5 km respectively.

To this purpose, we conducted experiments using a real test-case, the 2011 Cordón Caulle eruption, and compared performance of both models (FALL3D and WARIS-Transport) for different computational domains.The Puyehue-Cordón Caulle volcanic eruption took place on 4 June 2011 after decades of quiescence. The explosive phase of the eruption generated ash clouds that were dispersed over the Andes causing abundant ash fallout across the Argentinean Patagonia [17]. Ash dispersal was operationally forecasted by the Buenos Aires VAAC using the ETA-HYSPLIT and the WRF/ARW-FALL3D modeling systems. Here, we simulate the first three days of the eruption, from 4 June at 19h UTC to 7 June at 12h UTC. The idea is to have a real-case reference simulation of sufficient duration to compare the execution times. The computational domain cover $30^o \times 30^o$ (LON: -76^oW to -46^oW, LAT: -23^oS to -53^oS) at three spatial resolutions of 0.25, 0.1 and 0.05^o respectively (see Table 7.5) in order to analyze the effect of increasing the size of the computational mesh. The 0.05^o eruption case involves an input dataset of 71 GBytes of meteorological data (72 hours) and 1.2 GBytes of concentration and dispersal simulated data (66 hours). Although the scope of this research is not to match simulations and satellite observations, Figure 7.12 compares, for illustrative purposes, true color MODIS Terra satellite image with the current simulation output.

The results are shown using two different WARIS-Transport implementations, a naive ver-

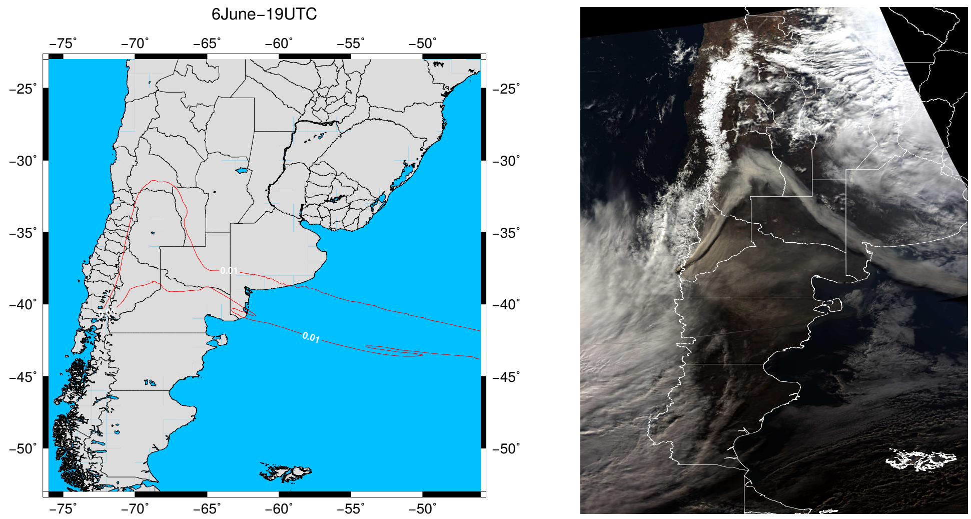


Figure 7.12: Comparison of simulated ash column mass (in g/m^2) and true-color TERRA/AQUA satellite images at 6 June 19 UTC time instant.

sion and a fully optimized one. The intent is to measure which is the effect of *opti* module over the execution times. The baseline implementation (naive) incorporates a pure MPI version, where the ADS kernel has been fully *SIMDized* including some pipeline optimizations (loop transformations). In the parallel I/O aspect, this naive version also includes an active buffering strategy [50] with two-phase collective I/O calls [33].

On the other hand, the optimized version introduces further improvements into the WARIS-Transport code through hybrid MPI-OpenMP parallelization, Semi-stencil, spatial-blocking, auto-tuning, and, finally, thread affinity (pinning threads to specific cores). To conduct the OpenMP optimization, first we identified the main computational loops of the WARIS-Transport routines. As expected, the most time consuming parts in Algorithm 17 resulted to be the explicit solver routines (lines 20, 23 and 24) and, to a lesser extent, the pre-process and post-process section codes for I/O. Then, we proceeded to annotate with OpenMP pragmas the loops of these routines. The annotation was combined with the intra-domain scheduler of *opti* module which bestowed an appropriate domain decomposition across threads ensuring not only a balanced workload but also an efficient data access to main memory (*e.g.* streaming access and NUMA aware). The second-order FD solver of the ADS equation was likewise improved by using the Semi-stencil approach [25]. Although the FALL3D model does not use a high-order stencil, which are better suited for Semi-stencil, its slope-limiter method for stabilization accesses data as a 13-point stencil. Thus, the Semi-stencil integration entailed an additional improvement of 10% compared to the classical and vectorized stencil kernel. Finally, the spatial-blocking algorithm of the ADS code was combined with the auto-tuner from the *opti* module. As example, Figure 7.13 shows how a proper blocking parameter is crucial to reduce the execution time of the explicit kernel. In this particular case (256$\times$2048$\times$64 domain size), the execution time has been reduced by 24.1% using spatial-blocking and auto-tuning.

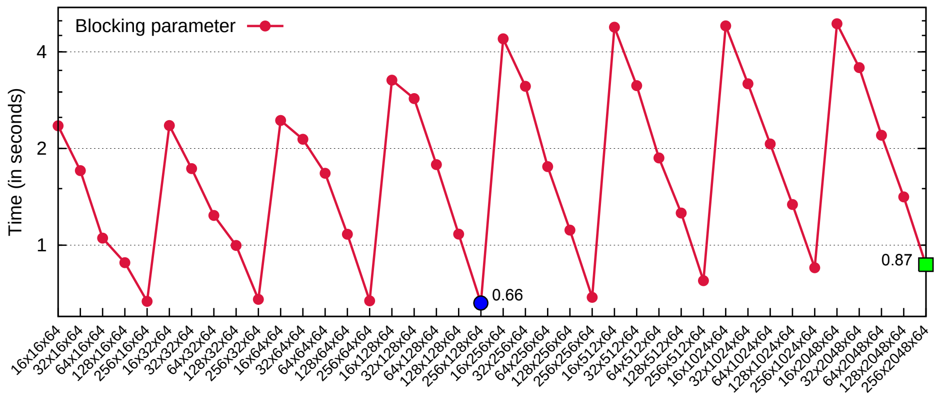


Figure 7.13: Spatial-blocking impact on the ADS kernel execution time. The green rectangle shows the performance of the naive implementation, whereas the blue circle depicts the best blocking parameter. The *opti* module automatically selects the best parameter.

The execution times were evaluated on two platforms, a multi-core platform with Intel Sandy Bridge-EP processors, and a prototype based on Intel Xeon Phi (MIC).

Intel Sandy Bridge-EP: As first platform, the MareNostrum supercomputer facility installed at the BSC was used. MareNostrum is part of the Partnership for Advanced Computing in Europe (PRACE) research infrastructure. MareNostrum has a peak performance of 1.1 Petaflops, with 48896 Intel Sandy Bridge processors in a total of 3056 nodes (16 processors per node). Each MareNostrum node incorporates 2 Intel Sandy Bridge-EP E5-2670 chips (dual socket). All nodes are interconnected through a dual-port high speed network Infiniband FDR10 for MPI communication (through POE). Finally, the supercomputer nodes have access to 2 PetaByte disk storage with General Parallel File System (GPFS).

Intel Xeon Phi (MIC): The MIC prototype configuration is composed of two Intel Xeon Phi 5110P cards attached to the PCI express bus of a MareNostrum compute node (dual Intel Sandy Bridge-EP E5-2670 chips). Due to the lack of GPFS drivers specifically compiled for MIC architecture, the global disk access is unavoidable configured through a double-mounted system of a NFS server in host which is in turn also mounted to the GPFS system of MareNostrum. The main drawback of this configuration is its poor I/O performance due to the NFS access, which provides a low bandwidth. Finally, as MPI implementation, OpenMPI was used.

Table 7.6 summarizes the FALL3D model and the non-optimized WARIS-Transport execution times for all 3 study cases. All these runs were conducted using 2 entire MareNostrum nodes with MPI for intra- and inter-node communication between different domains. Times

Domain cases	CPU time (s)	FALL3D	WARIS-Transport	Speed-up
Caulle-0.25	Input	1991	27	-
	Output	229	15	-
	Kernel	1441	97	-
	TOTAL	**3458**	**163**	**21.21**$\times$
Caulle-0.10	Input	2085	71	-
	Output	1508	17	-
	Kernel	10099	1030	-
	TOTAL	**12525**	**1138**	**11.00**$\times$
Caulle-0.05	Input	2245	187	-
	Output	4132	46	-
	Kernel	38622	4735	-
	TOTAL	**41334**	**4978**	**8.30**$\times$

Table 7.6: Time comparison between FALL3D and the naive (non-optimized) version of WARIS-Transport using 32 cores of MareNostrum supercomputer (2 Intel sandy bridge nodes).

are broken down for the three main parts of the simulation: input, output and kernel. Input time includes the cost of reading and pre-processing the meteorological data for each hour of simulated time (8 variables of $n_z \times n_x \times n_y$ and 4 variables of $n_x \times n_y$ read hourly). Output time considers the execution time of post-processing and writing results every hour of simulated time (13 variables of $n_z \times n_x \times n_y$ dimension). Finally, the kernel time refers to the cost of computing explicit kernels such as: ADS equation, boundary conditions, ground accumulation (ash fallout), mass lost at boundaries and the correction of the unbalanced mass due to non-null divergence terms.

Looking at results in Table 7.6 it is observed that WARIS-Transport outperforms FALL3D on all resolutions. The speed-ups shown with the preliminary WARIS-Transport implementation range from 8.30$\times$ in the worst case up to 21.21$\times$ in the best depending on the balance between computation and communication. Several reasons explain this higher performance. First, I/O operations are serialized in FALL3D due to the use of the sequential NetCDF-3 library. As a result, only the master process (MPI task 0) performs reading and writing operations, and therefore, it is in charge of sending to and receiving from each computational domain the required I/O data. This sequential I/O strategy does not scale because the master process becomes the bottleneck as the number of domains involved in the simulation increases. In opposition, WARIS-Transport uses the parallel implementation of the NetCDF-4 library, which takes advantage of MPI-IO, enabling a much better scaling. Second, the WARIS-Transport module makes use of SIMDized kernels that take advantage of the SIMD units in the Intel Sandy Bridge architecture (AVX). This vectorization may speed-up the performance up to 8$\times$ compared to a scalar implementation when single-precision floating-point operations are carried out (8 FLOPs per AVX operation). In addition, the loops of these explicit kernels have been also rearranged to reduce memory access latency by accessing required data in a proper way. Finally, the multi-threaded execution flow of WARIS framework permits the WARIS-Transport module to overlap I/O, MPI communications and computation of the explicit kernel. By doing so, a fraction of I/O and MPI exchange of boundary areas can be hidden with the remaining parts of the simulation.

Although the initial naive WARIS-Transport results unveil a considerable speed-up compared to FALL3D, further margin of performance improvement exists for the current HPC scenario. Actually, parallel limitations can be expected in pure MPI applications when strong scaling is performed because of the decreasing ratio of internal nodes with respect to nodes that must be exchanged across neighbor domains (the more MPI decompositions are performed, the more dominant communication across domains becomes). Modern architectures expose high levels of parallelism through the ability of spawning threads that can run simultaneously in the same chip by means of multiple cores. Within this multi-core environment, computation and implicit communications can be efficiently conducted at intra-node level by using shared memory paradigms such as OpenMP instead of MPI. The hybrid MPI-OMP version follows this concept.

Processing	Intel Sandy Bridge		Intel Xeon
Units	Pure MPI (naive)	MPI+OMP (*opti*)	Phi (MIC)
1	7015 (7.3×)	5368 (9.6×)	5633 (9.1×)
2	4978 (8.3×)	2812 (14.6×)	2845 (14.6×)
4	2812 (12.7×)	1541 (23.3×)	-
8	1954 (16.9×)	806 (41.0×)	-
16	1815 (16.0×)	527 (55.2×)	-
32	-	475 (N/A)	-

Table 7.7: WARIS-Transport simulation times (in seconds) on Intel Sandy Bridge and Intel Xeon Phi. Values are for the Caulle-0.05 case considering 3 days of simulation and hourly I/O. Results are given in *processing units*, corresponding to 1 MareNostrum node (16 processors) for the Intel Sandy Bridge and to 1 host (node) plus 1 card for Intel Xeon Phi. Speed-up time factor with respect to the FALL3D original implementation is shown in parenthesis.

Table 7.7 compares the WARIS-Transport execution time for the Caulle-0.05 case considering the naive pure MPI and the optimized hybrid MPI-OMP version on Intel Sandy Bridge and Intel Xeon Phi (MIC). In order to fairly compare performance between these two platforms, the scalability results are ordered by the term *processing unit*, which refers to the minimum hardware unit available to purchase (*e.g.* node or accelerator card). The WARIS-Transport specialization for MIC has been deployed using a pure native implementation, where OpenMP and WARIS infrastructure threads are run exclusively in MIC device. The host only provides access to the global disk. As many-core architecture with a cache-based memory hierarchy, the MIC processor can also take advantage of *opti* module optimizations. Regarding the thread affinity policy, on Intel Sandy Bridge nodes, 15 OpenMP threads were mapped to 15 cores, and 1 core was specifically dedicated to WARIS Pthreads. On the other hand, for Intel Xeon Phi runs, 232 OpenMP threads were spawned and mapped to 58 cores (4 SMT threads per core), and 2 entire cores were exclusively assigned to WARIS tasks (data flow, communication and I/O). As constraint, certain cases were not feasible to be conducted. First, the FALL3D and naive WARIS-Transport runs were only executed up to 256 CPUs because of MPI decomposition limitations (n_y dimension too small for 512 MPI tasks). Second, due to the network configuration of the MIC prototype system with OpenMPI, the analysis was limited to the 2 MIC cards available on the same host.

As observed in Table 7.7, the Intel Sandy Bridge optimized version of WARIS-Transport gives additional improvements between 23% and 70% with respect to the pure MPI naive one, where the gain increases with the number of processing units. Likewise, the speed-up factor with respect to the original FALL3D implementation (shown in parenthesis) is significantly increased, specially with more than 128 CPUs. The Intel Xeon Phi gives very similar results per processing unit. On the other hand, the scalability efficiency of Intel Sandy Bridge nodes remains above 80% up to 8 processing units (128 CPUs), and the MIC accelerator-based implementation presents nearly optimal scalability (above 90%), but our analysis limits to the 2 MIC cards available. Nevertheless, a significant degradation is observed in the Intel Sandy Bridge scalability with 16 and specially 32 processing units (corresponding to 256 and 512 CPUs respectively).

Num.	Explicit kernel			Meteo data		Dispersal		Wait	Total	Comp.	Scal.
Proc.	P1	P2	P3	Input	Post-	Pre-	Output	sync. I/O	time	efficiency	efficiency
16	0	2971	2046	*816*	302	8	*65*	34	5368	100%	100%
32	17	1507	1065	*438*	151	5	*79*	62	2812	97.0%	95.4%
64	16	778	598	*275*	76	2	*69*	67	1541	90.6%	87.0%
128	16	410	190	*1*	38	1	*72*	144	806	100%	83.2%
256	16	217	104	*1*	20	1	*113*	165	527	93%	63.6%
512	15	126	64	*1*	11	1	*217*	255	475	76.4%	35.3%

Table 7.8: Break-down of the optimized WARIS-Transport times on Intel Sandy Bridge with Caulle-0.05 case. Efficiency is shown disregarding the I/O (Computation efficiency) and considering the whole time (Scalability efficiency).

In order to explain the reasons of this lost of performance, Table 7.8 breaks down the execution time of the Caulle-0.05 case with Intel Sandy Bridge platform. Results are categorized in four groups, explicit kernels (*P1*, *P2* and *P3* stages), meteorological input, ash dispersal output and active wait due to synchronous I/O. *P1* (boundary elements), *P2* (internal elements) and *P3* stages refer to the kernel functions in the PSK framework structure. *Post-process* and *pre-process* columns consider the required computation for the data arrangement after reading meteorological data and before writing ash dispersal results respectively. Finally, unlike *Input* and *Output* columns, *wait synchronous I/O* includes the time spent on serialized I/O (not overlapped with other WARIS-Transport tasks).

Three main issues arise on these executions. First I/O becomes a bottleneck, specially with a large number of MPI tasks. As the number of processors is increased, the required time for I/O cannot be olverlapped with the remaining tasks. The reason is that computing

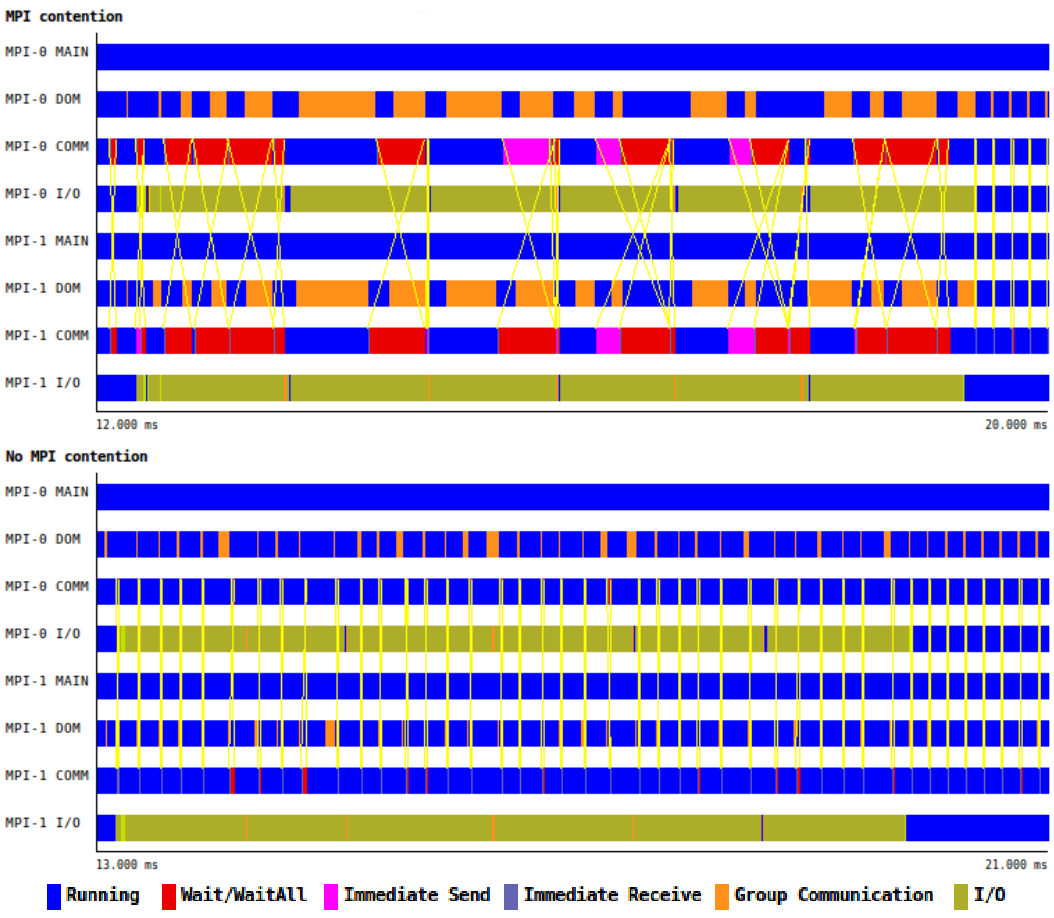


Figure 7.14: EXTRAE traces of WARIS-Transport with Caulle-0.05 case. For clarity purposes, only WARIS Pthreads are shown for two MPI tasks in a 8 seconds timeline. Each row represents a different execution flow (MPI task and Pthread) and the operation being performed (see colors in legend). In these traces, the operations shown are: non-blocking MPI calls (Immediate Receive/Send), waits for non-blocking MPI calls (Wait/WaitAll), MPI collectives (Group Communication), I/O and others (Running). The yellow lines draw the point-to-point communications between *Communication threads* when boundary nodes are being transferred at each time-step. Top: case where MPI contention is produced with a large number of tasks. Bottom: a better MPI communication pattern is obtained with a lower number of tasks.

resources are augmented whereas the GPFS I/O servers in MareNostrum remain the same. Second, as the number of domains is increased, each MPI process must read and write smaller chunks of data that entails inefficient gather and scatter operations in the MPI-IO layer. And third, at a certain number of processors, we have experienced serious network contention when MPI routines (point-to-point and collective) and the MPI-IO layer (through NetCDF) are concurrently used by PSK threads.

The EXTRAE traces shown in Figure 7.14 clearly depict this issue. A strong MPI degra-

MPI case	Running	Wait	Imm. Send	Imm. Receive	Group Comm.	I/O
Contention	38.0	5.7	1.3	0.003	6.7	12.3
No contention	49.7	0.374	0.039	0.004	0.889	13.0

Table 7.9: Time break-down of EXTRAE traces with Caulle-0.05 case. Times (in seconds) have been aggregated for all threads.

dation can be observed in threads involving communication while *I/O thread* is performing the MPI-IO. The MPI contention entails a poorer performance due to the stagnation of *Domain* and *Communication threads* that must wait for critical messages (boundary nodes and collective reductions) to proceed with the next time-step. For example, during the timeline shown in Figure 7.14, a total of 21 time-steps were conducted for the worst case (top) and 42 time-steps in the case without contention (bottom). Table 7.9 breaks down this timeline, showing the aggregated time spent on each part for all threads. In the case without MPI contention, threads spend more time computing (Running) than performing operations that involve communications.

In order to minimize the MPI contention on executions with more than 128 CPUs, the most critical part of the I/O (71 GBytes of meteorological data) was previously read in the initialization part of WARIS-Transport. Although this strategy entailed the serialization of I/O and therefore higher *Wait synchronous I/O* values, the whole execution time was improved by the fact that the better MPI behavior overcame the cost of serializing the I/O.

7.4 Summary

In the first part of this chapter, we have demonstrated that Semi-stencil algorithm is a valid technique to optimize scientific applications, such as the RTM on heterogeneous architectures like Cell/B.E. Both, the 25-point stencil of the RTM kernel (wave propagator) and the low-latency scratchpad memory of the SPEs are good candidates for our novel algorithm. A fully-operational RTM code used in Oil & Gas industry has been enhanced by 16% using this novel algorithm.

In the second part of this chapter, we have presented the WARIS framework. The WARIS infrastructure has shown appealing capabilities by providing successful support for scientific problems using FD methods. In order to assess its performance, a code that solves the vectorial Advection–Diffusion–Sedimentation equation has been ported to the WARIS framework. This problem appears in many geophysical applications, including atmospheric transport of passive substances. As an application example, we focus on atmospheric dispersion of volcanic ash, a case in which operational code performance is critical given the threat posed by this substance on aircraft engines. The results of WARIS-Transport are very promising, performance has been improved by 55.2$\times$ with respect to the baseline code of FALL3D using a realistic case. This opens new perspectives for operational setups, including efficient ensemble forecast.

Chapter 8

Conclusions and Future Work

This thesis is devoted to leverage the performance of stencil computations that arise not only in academia examples but also in real-applications used in industrial cases. To this extent, this thesis has focused on four different topics that affects FD-based codes: optimization of the stencil computation, strategies to facilitate the scalability on complex SMP architectures, design of a flexible and accurate performance model for stencil computations, and the assessment of the optimizations proposed in real applications for industry. The following sections elaborate on the contributions and future work of each developed topic.

8.1 Optimizing Stencil Computations

Stencil codes are usually affected by low data reuse and non-unit stride memory access patterns that jeopardize the global performance of the computations. Some techniques like loop unrolling, software pipelining or software prefetching can help improve the performance of stencil codes by partially hiding the dependency stalls. However, these techniques have a limiting factor, namely the available register pool in the processor. Other algorithms like Rivera or Cache-oblivious tackle the memory access pattern issue with partial success, reducing the overall transfer latency in cache-based architectures.

In order to address the performance issues of stencil-based computations, in Chapter 4, we have presented a novel optimization technique called *Semi-stencil*. This new algorithmic approach has shown significant performance improvements on test cases conducted with StencilProbe micro-benchmark over medium- and high-order stencils ($\ell > 2$). The Semi-stencil contributions are:

- It reduces the working data set, thus minimizing register pressure and cache memory footprint. This effect becomes more pronounced for high-order stencils.

- It increases spatial and temporal data locality, owing to a reduction in loads issued, but increasing slightly the required stores. In cache-based architectures, the cache coherence and the write allocate policy (usually associated with write-back) may produce further memory traffic due to the extra store transactions. Despite the additional stores, the benefit in time of the load reduction overcomes the penalty of stores.

- Due to its orthogonal property, the Semi-stencil outperforms other techniques on many architectures, either when implemented alone or combined with space and time-blocking algorithms.

The experimental results show that the best classical implementations of a 25-point stencil are typically able to deliver up to 30% of the peak performance. Under the same conditions, the Semi-stencil implementations can achieve up to 1.32× performance improvement. Additionally, through the insight provided by the Roofline model and the $FP/Cache$ metric, we have revealed how the Semi-stencil algorithm performs in terms of *Operational Intensity* and memory traffic.

In SMP systems, the Semi-stencil algorithm has also added further scalability on most of the analyzed platforms. In the POWER7 architecture, the scalability soars to 13.06× for the Oblivious+*Semi-stencil* case when is run over 32 threads. Furthermore, when only 4 threads are used in this platform, the performance is widely enhanced in most cases and the execution time almost halved with respect to the classical version. We have also demonstrated the much better parallel behavior of the Semi-stencil algorithm on novel architectures such as Intel Xeon Phi (MIC). For instance, on this architecture, the scalability reaches 93.8× over 244 threads for certain tests cases. Although the scalability has been reached with certain success, the obtained results reveal that stencil-based optimizations must be combined with load balancing strategies addressed specifically for SMP architectures.

Future work will focus on further research of this novel algorithm in several aspects. First, the cost of each computational part (*head*, *body* and *tail*) will be broken down with respect to the global performance. Second, the benefits in performance of each *forward* and *backward* update will be analyzed in detail for each Semi-stencil axis. Third, on cache-based architectures with write allocate policy, write misses may have a negative impact due to the pollution that the cache-line allocation produces. Architectures with cache-bypass techniques that minimize cache pollution when writing data to memory have already appeared (*vmovntpd* on x86 and *dcbtst* on POWER architectures). Our future efforts will assess the performance of the Semi-stencil algorithm when combined with non-temporal stores. Finally, the Semi-stencil strategy will be also evaluated on numerical methods that implement similar memory access pattern such as Lattice Boltzmann Magnetohydrodynamics (LBMHD).

8.2 SMT, Multi-core and Auto-tuning

Currently, the GHz race for higher frequencies has slowed down due to technological issues. The main challenge has been the CPU power consumption, which increases proportionally to the clock frequency and to the square of the die voltage. As a solution, manufacturers have migrated to CMP architectures, where the main source of performance comes from the intra-node exploitation of multi- and many-core processors at lower frequencies. Within this approach, the scalability is a major concern with an increasing number of replicated cores and accelerators. Resources including memory, communication buses and shared caches become bottlenecks and entail stagnation of performance. Therefore, the parallelization of stencil codes requires careful considerations to fully exploit the CMP capabilities.

In this regard, Chapter 5 has proposed a set of intra-node strategies addressed to CMP architectures in order to leverage the performance of stencil computations. The intra-node strategies of this thesis contributes in the following areas:

- **SMT affinity**: we have proposed a work-load distribution by planes for SMT threads that is two-fold. First, the private-core resources are better shared across SMT threads, not only affecting caches, but also prefetcher engines. Second, as a consequence of the thread cooperation, cache reuse is improved, reducing conflict and capacity misses due to the smaller data footprint. This data reuse also promotes a better prefetching effectiveness by reusing streams and avoiding prefetch disruption.

- **Multi-core decomposition**: in order to tackle the work-balance on CMP platforms, we have suggested four decomposition schedulers for stencil computations that take into account data reuse (spatial and temporal locality) and prefetching effectiveness. In order to do that, threads mapped into the same core are assigned to a certain number of consecutive planes guaranteeing stream access by preventing cutting along the unit-stride dimension (Z).

- **Auto-tuning**: to guarantee a better scalability, we have also included a simple auto-tuner that searches for a good tiling candidate (TJ) of the spatial-blocking algorithm. This auto-tuning step uses a gradient descent search to find a pseudo-optimal value in a small portion of time. Then, threads traverse their respective core domains in $NZ_{core} \times TJ \times NY_{core}$ blocks.

Through experiments on two leading platforms, we have demonstrated that the combination of these three techniques are crucial in order to allow higher scalability results. This is specially true in novel architectures with a high level of thread parallelism such as Intel Xeon Phi, where stagnation exacerbates as SMT threads are increased. Indeed, in certain cases these three combined techniques allowed to improve the scalability from $\approx$50$\times$ up to 123$\times$ for an ADR kernel.

Among all decomposition schedulers, we have proposed the *balanced* scheduler, which strives to minimize the imbalance across intra-domains while a new introduced metric, named β, is maximized. This metric measures the percentage of internal points with respect to the halos that are required to compute the stencil operator. The higher this metric is, the less memory traffic the halos represent. As a consequence of its higher β metric, the *balanced* scheduler always yielded the best scalability results on both platforms. By means of this scheduler, we have demonstrated not only the importance of reducing the imbalance, but also of attaining a reasonable β to reduce the redundant data transferred from memory to processor. On the other hand, we have also presented a scheduler called *guided*, that employs a pseudo-optimal TJ parameter from a prior auto-tuning step. This scheduler uses the TJ parameter as heuristic to prevent the decomposition on X axis under the TJ value. Nevertheless, the experimental results expose poorer performance than *balanced* because NX_{core} has been bound as $> TJ$ at the expense of decreasing the β metric.

Finally, we have also realized, that the memory enhancements proposed in this chapter cannot be easily implemented through `schedule(type[,chunk])` and `collapse(2)` clauses from OpenMP. These clauses do not permit to promote data reuse neither between threads residing in the same core nor within the same thread due to spatial-blocking. The main issue of these OpenMP clauses is the inability to enforce that computational blocks reside in the contiguous direction for the least-stride dimension (Y).

8.3 Performance Modeling of Stencil Computations

In Chapter 6, we have presented a performance model tailored for stencil computations. Our stencil modeling study started with a preliminary model that despite some weaknesses was able to capture roughly the performance behavior of naive stencil computations. The accuracy of the prediction ranged depending on the platform. In x86 architectures case, on which we spent most of our research time, an acceptable level of prediction accuracy was obtained. Furthermore, the average relative error in the execution time prediction was 13% for the whole range of stencil sizes. These results proved that we were in the right path to achieve a reliable multi-level cache model, which at the same time remained simple, flexible and extensible. Nevertheless, the prefetching mechanism devised for the preliminary model lacked accuracy when the streaming bandwidth was dominant in the execution time (low-order stencils and small problems) and especially when several threads were concurrently triggering the prefetching engine.

As an enhancement, we extended the model to an advanced version. Several new features were included into the model such as: multi-core support, improved hardware prefetching modeling, cache interference due to conflict and capacity misses and spatial blocking and Semi-stencil optimization techniques. The most challenging part was the prefetching modeling, specially when too many arrays are accessed concurrently, overwhelming the hardware prefetching system and hampering the bandwidth performance. In these scenarios, an aggressive prefetching intervention causes eviction of data that could be reused later (temporal reuse), polluting the cache and affecting adversely the bandwidth performance. Loop fission and data layout transformations can occasionally improve the performance in these cases. Nevertheless, they must be applied carefully because some side effects may appear. In order to effectively capture the stream engine behavior in all above mentioned cases, the *prefetching effectiveness* approach was adopted. As shown in the experiments, this approach can be successfully used in SMT context, where the prefetching efficiency is substantially reduced due to contention of the shared resources.

The new methodology in the performance model contributes to unveil insights about how stencil codes might be built or executed in order to leverage the prefetching efficiency. The proposed model could also be included as static analysis in auto-tuning frameworks to guide in making decisions about algorithmic parameters for stencil codes; thus providing a rich synergy towards efficient stencil code implementation. Likewise, our model might be useful in expert systems, not only for compilers or auto-tuning tools, but also in run-time optimizations for dynamic analysis. For instance, the model might decide the SMT configuration and the number of threads to be spawned that outperform the remaining combinations based on the prefetching engines, the problem size ($I \times J$) and the stencil order (ℓ).

To our knowledge, this is the first stencil model that takes into account two important phenomena: the cache interference (due to $II \times JJ$ and P_{read} parameters) and the prefetching effectiveness when concurrent threads are running in the same core. Despite the fact that the current work has been only conducted for 1st order in time and constant coefficient stencils, the model could be adapted to higher orders in time and variable coefficients (anisotropic medium) by adjusting the cost of cache miss cases ($C_{1,2,3,4}$) and their rules ($R_{1,2,3,4}$) through $P_{read,write}$ and $S_{read,write}$ variables.

Future work will include temporal blocking as optimization method, and different thread domain decomposition strategies apart from the static scheduling (cutting in the least-stride dimension). Nonetheless, the addition of software prefetching behavior into the model is unattainable since it depends on the internal compiler heuristics, and the pragmas inserted by the user.

8.4 Case Studies

In Chapter 7 of this thesis, we have devoted our efforts to demonstrate the soundness of the optimizations of previous chapters into real-life applications used in the industry.

First, we have reviewed the effect of Semi-stencil algorithm in a RTM code from the Kaleidoscope project for the Cell/B.E. architecture. The RTM model is a seismic imaging method used in Oil & Gas industry to search hidrocarbon-rich reservoirs where up to 80% of the execution time is spent on solving the wave equation through stencil computations. The Semi-stencil has shown two benefits compared to the classical stencil algorithm for this scientific application. First, it reduces the number of Z-X_{SPE} planes kept in LS, allowing to increase the X_{SPE} parameter and thus reducing the transferences. Second, although stores are increased in Semi-stencil algorithm, the number of loads issued in the pipeline are reduced in a noticeable way, alleviating significantly the pressure over the load-store unit (almost 40%). Compared to cache hierarchy architectures, the additional stores do not pose any negative impact on the global performance due to the constant latency of accessing the scratchpad memory. In the single-SPE version of the FD scheme for the wave equation, the best classical stencil implementation with SIMD code and pipeline optimizations was able to reach up to 34% of the SPE peak performance. Under the same conditions, we achieved 49% deploying the Semi-stencil algorithm. This algorithm has also revamped the performance of a fully operational RTM code by speeding up the overall execution time by 16%.

In the second part of the chapter, we have presented the WARIS framework. WARIS, a BSC in-house infrastructure on which this thesis has partially contributed with some improvements, has shown appealing capabilities by providing successful support for scientific problems using FD methods. Through a modular perspective, it provides support for a wide-range of hardware platforms, easing the specific platform support as the computation race keeps the hardware changing everyday. We have included a new module called *opti* that incorporates most of the optimizations techniques devised in this thesis. As application example to validate our research, we have developed WARIS-Transport, a porting of the FALL3D model. FALL3D is an Eulerian Atmospheric Transport Model (ATM) tailored to simulate transport and deposition of volcanic tephra at both research and operational levels, a case in which operational code performance is critical given the threat posed by this substance on aircraft engines. We have compared the WARIS-Transport performance with respect to the baseline code of FALL3D using a real simulation occurred in 2011 at Puyehue-Cordón Caulle. The optimized version of WARIS-Transport, which makes use of *opti* module and takes advantage of an hybrid MPI-OpenMP parallelization, spatial-blocking, auto-tuning and thread affinity, lead to much better scalability and a speed-up of $9.6\times$ to $55.2\times$. From the ATM operational point of view, the importance of code optimizations on HPC platforms is clear. On one hand, for a given operational model setup, to dispose of a code at least one order of magnitude faster allows to face ensemble forecast strategies with execution time constrains compatible with operations. On the other, given a limited computing time and computational resources, higher resolution simulations can be considered.

Future work will explore the reasons of the poor performance encountered between MPI routines (point-to-point and collectives) and MPI-IO layer when they are simultaneously used by the WARIS-PSK infrastructure. MPI_Send/Recv and specially collectives calls, which are critical to conduct the simulation steps, must not be stagnated by lower priority MPI transfers such as MPI-IO. In the near future with the exascale computing, parallel I/O will definitely become a bottleneck which must be rapidly addressed. The solution to this issue comes necessarily from the fact that I/O, computation and communication must be efficiently overlapped without contention in scientific applications.

Appendix A

Numerical Equations

A.1 Heat Equation

The heat equation, a simplification of the diffusion equation, is a parabolic PDE that describes the heat conduction (variation of temperature) in a given medium over time. The heat equation for u in any coordinate system is given by

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{A.1}$$

and, more specifically for $u(x, y, z, t)$, in a Cartesian coordinate system, is

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{A.2}$$

where α is a positive constant depending on the coefficient of thermal conductivity over the medium and $\nabla^2 u$ is the Laplace operator. The Laplace operator merely states that heat flows down a temperature gradient depending on the thermal diffusivity that might represent the insulation value of the material being described (α). The heat equation is one of the simplest PDEs in physics, being first-order in time and second-order in space.

The finite difference approximation for the 1D heat equation can be written as

$$\frac{u_i^{t+1} - u_i^t}{\Delta t} = \alpha \left(\frac{u_{i-1}^t - 2u_i^t + u_{i+1}^t}{\Delta x^2} \right) + \mathcal{O}(\Delta x^2, \Delta t) \tag{A.3}$$

using a forward difference in time and a centered difference scheme in space. Thus, obtaining a Forward-Time and Centered-Space (FTCS) approximation of the heat equation.

Then, the temperature for $u(x, t)$ can be easily deduced by the following explicit equation,

$$u_i^{t+1} = u_i^t + \frac{\alpha \Delta t}{\Delta x^2} \left(u_{i-1}^t - 2u_i^t + u_{i+1}^t \right). \tag{A.4}$$

A.2 Wave Equation

The wave equation is a hyperbolic PDE that models the displacement of a wave or perturbation (u) in a specific medium over time (t). The wave equation for u in any coordinate is,

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{A.5}$$

and, for a 3D problem $u(x, y, z, t)$ in a Cartesian coordinate system is

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right), \tag{A.6}$$

where c is a constant wave speed that depends on the medium. The wave equation resembles the heat equation, but differing in the second-order term in time.

Approximating the second-order derivatives by central differences (CD) both in space and time, the finite difference scheme for the 1D wave equation can be written as

$$\frac{u_i^{t-1} - 2u_i^t + u_i^{t+1}}{\Delta t^2} = c^2 \left(\frac{u_{i-1}^t - 2u_i^t + u_{i+1}^t}{\Delta x^2} \right) + \mathcal{O}(\Delta x^2, \Delta t^2). \tag{A.7}$$

Therefore, the equation can be explicitly solved by

$$u_i^{t+1} = \underbrace{2u_i^t - u_i^{t-1}}_{\text{Temporal integrator}} + \left(\frac{c\Delta t}{\Delta x} \right)^2 \underbrace{(u_{i-1}^t - 2u_i^t + u_{i+1}^t)}_{\text{Spatial integrator}}. \tag{A.8}$$

A.3 Advection–Diffusion–Reaction Equation

The Advection–Diffusion–Reaction (ADR) equation is a PDE that models the concentration of particles in a process where the particles (usually chemical species) suffer a reaction, diffusing over the medium and transported by the movement of the fluid (advection). The ADR equation modeling the concentration C of the particles in any coordinate system is given by

$$\frac{\partial C}{\partial t} + \nabla \cdot (vC - K\nabla C) = S \tag{A.9}$$

and, more specifically for $C(x, y, z, t)$, in a Cartesian coordinate system, is

$$\begin{aligned} &\frac{\partial C}{\partial t} + \frac{\partial}{\partial x}(UC) + \frac{\partial}{\partial y}(VC) + \frac{\partial}{\partial z}(WC) = \\ &\frac{\partial}{\partial x}\left(K_x \frac{\partial C}{\partial x} \right) + \frac{\partial}{\partial y}\left(K_y \frac{\partial C}{\partial y} \right) + \frac{\partial}{\partial z}\left(K_z \frac{\partial C}{\partial z} \right) + S, \end{aligned} \tag{A.10}$$

where C is the transported quantity; (U, V, W) are the given fluid velocity components; $K_x(x, y, z)$, $K_y(x, y, z)$, and $K_z(x, y, z)$ are the turbulent diffusion coefficients, and $S(x, y, z)$ is the source term.

The solving algorithm for Equation A.10 considers a Lax-Wendorff scheme for advection (second-order accurate in both space and time) with a slope-limiter method (minmod), whereas the diffusive terms are evaluated using a central difference scheme accounting for a variable turbulent diffusivity tensor. Given the values of $C_{ijk}^n = C_{ijk}$ at the previous time step and at point of indexes (i, j, k), the value of C_{ijk}^{n+1} at time $t = t + \Delta t$ is given by:

$$
\begin{aligned}
& C_{ijk}^{n+1} = C_{ijk}^{n} + \Delta t S_{ijk} \\
& -\frac{1}{2}\left[1 + sgn(1, U_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta x}\left(U_{ijk}C_{ijk} - U_{i-1jk}C_{i-1jk}\right) + \frac{\Delta x}{2}\left[Cr_i\left(1 - Cr_i\right)\sigma_i - Cr_{i-1}\left(1 - Cr_{i-1}\right)\sigma_{i-1}\right]\right) \\
& -\frac{1}{2}\left[1 - sgn(1, U_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta x}\left(U_{i+1jk}C_{i+1jk} - U_{ijk}C_{ijk}\right) + \frac{\Delta x}{2}\left[Cr_{i+1}\left(1 - Cr_{i+1}\right)\sigma_{i+1} - Cr_i\left(1 - Cr_i\right)\sigma_i\right]\right) \\
& -\frac{1}{2}\left[1 + sgn(1, V_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta y}\left(V_{ijk}C_{ijk} - V_{ij-1k}C_{ij-1k}\right) + \frac{\Delta y}{2}\left[Cr_j\left(1 - Cr_j\right)\sigma_j - Cr_{j-1}\left(1 - Cr_{j-1}\right)\sigma_{j-1}\right]\right) \\
& -\frac{1}{2}\left[1 - sgn(1, V_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta y}\left(V_{ij+1k}C_{ij+1k} - V_{ijk}C_{ijk}\right) + \frac{\Delta y}{2}\left[Cr_{j+1}\left(1 - Cr_{j+1}\right)\sigma_{j+1} - Cr_j\left(1 - Cr_j\right)\sigma_j\right]\right) \\
& -\frac{1}{2}\left[1 + sgn(1, W_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta z}\left(W_{ijk}C_{ijk} - W_{ijk-1}C_{ijk-1}\right) + \frac{\Delta z}{2}\left[Cr_k\left(1 - Cr_k\right)\sigma_k - Cr_{k-1}\left(1 - Cr_{k-1}\right)\sigma_{k-1}\right]\right) \\
& -\frac{1}{2}\left[1 - sgn(1, W_{ijk})\right] \times \\
& \left(\frac{\Delta t}{\Delta z}\left(W_{ijk+1}C_{ijk+1} - W_{ijk}C_{ijk}\right) + \frac{\Delta z}{2}\left[Cr_{k+1}\left(1 - Cr_{k+1}\right)\sigma_{k+1} - Cr_k\left(1 - Cr_k\right)\sigma_k\right]\right) \\
& +\frac{\Delta t}{2\Delta x^2}\left[\left(K_{x_{i+1jk}} + K_{x_{ijk}}\right)\left(C_{i+1jk} - C_{ijk}\right) - \left(K_{x_{ijk}} + K_{x_{i-1jk}}\right)\left(C_{ijk} - C_{i-1jk}\right)\right] \\
& +\frac{\Delta t}{2\Delta y^2}\left[\left(K_{y_{ij+1k}} + K_{y_{ijk}}\right)\left(C_{ij+1k} - C_{ijk}\right) - \left(K_{y_{ijk}} + K_{x_{ij-1k}}\right)\left(C_{ijk} - C_{ij-1k}\right)\right] \\
& +\frac{\Delta t}{2\Delta z^2}\left[\left(K_{z_{ijk+1}} + K_{z_{ijk}}\right)\left(C_{ijk+1} - C_{ijk}\right) - \left(K_{z_{ijk}} + K_{z_{ijk-1}}\right)\left(C_{ijk} - C_{ijk-1}\right)\right],
\end{aligned}
\tag{A.11}
$$

where Cr is the Courant number and σ is the minmod function:

$$
\begin{aligned}
Cr_i &= \frac{\Delta t}{\Delta x} U_{i-\frac{1}{2}jk} = \frac{1}{2}\frac{\Delta t}{\Delta x}\left(U_{ijk} + U_{i-1jk}\right) \\
Cr_j &= \frac{\Delta t}{\Delta y} V_{ij-\frac{1}{2}k} = \frac{1}{2}\frac{\Delta t}{\Delta y}\left(V_{ijk} + V_{ij-1k}\right) \\
Cr_k &= \frac{\Delta t}{\Delta z} W_{ijk-\frac{1}{2}} = \frac{1}{2}\frac{\Delta t}{\Delta z}\left(W_{ijk} + W_{ijk-1}\right) \\
\sigma_i &= \frac{1}{2}\left[sgn\left(1, C_{i+1jk} - C_{ijk}\right) + sgn\left(1, C_{ijk} - C_{i-1jk}\right)\right] min\left(\frac{|C_{i+1jk} - C_{ijk}|}{\Delta x}, \frac{|C_{ijk} - C_{i-1jk}|}{\Delta x}\right) \\
\sigma_j &= \frac{1}{2}\left[sgn\left(1, C_{ij+1k} - C_{ijk}\right) + sgn\left(1, C_{ijk} - C_{ij-1k}\right)\right] min\left(\frac{|C_{ij+1k} - C_{ijk}|}{\Delta y}, \frac{|C_{ijk} - C_{ij-1k}|}{\Delta y}\right) \\
\sigma_k &= \frac{1}{2}\left[sgn\left(1, C_{ijk+1} - C_{ijk}\right) + sgn\left(1, C_{ijk} - C_{ijk-1}\right)\right] min\left(\frac{|C_{ijk+1} - C_{ijk}|}{\Delta z}, \frac{|C_{ijk} - C_{ijk-1}|}{\Delta z}\right).
\end{aligned}
\tag{A.12}
$$

The stability of the numerical scheme is ensured by using a time step Δt lower than the critical. As established by Hindmarsh *et al.*, an explicit scheme for the multidimensional advection-diffusion equation is numerically stable given the condition:

$$
\Delta t \leq \frac{1}{2\left(\frac{K_x}{\Delta x^2} + \frac{K_y}{\Delta y^2} + \frac{K_z}{\Delta z^2}\right) + \frac{|U|}{\Delta x} + \frac{|V|}{\Delta y} + \frac{|W|}{\Delta z}} \tag{A.13}
$$

A.4 FALL3D Governing Equation

The FALL3D governing equation simulates a 3D time-dependent Eulerian model for the transport and deposition of tephra. This model, derived from the ADR equation (see Appendix A.3), solves a set of Advection–Diffusion–Sedimentation (ADS) equations on a structured terrain-following grid using a second-order Finite Differences (FD) explicit scheme. The model inputs are meteorological data, topography, vent coordinate, eruption source parameters such, mass flow rate, eruption duration, and particle shape and density information. The outputs of the model are tephra ground load/thickness, airborne ash concentration and other related variables.

In a rectangular Cartesian (x, y, z) coordinate system, the FALL3D tracer continuity equation is given by

$$
\begin{aligned}
&\frac{\partial c}{\partial t} + u_x\frac{\partial c}{\partial x} + u_y\frac{\partial c}{\partial y} + \left(u_z - v_{sj}\right)\frac{\partial c}{\partial z} + c\left(\nabla \cdot \mathbf{u}\right) - c\frac{\partial v_{sj}}{\partial z} = \\
&\frac{\partial}{\partial x}\left(\rho k_x \frac{\partial c/\rho}{\partial x}\right) + \frac{\partial}{\partial y}\left(\rho k_y \frac{\partial c/\rho}{\partial y}\right) + \frac{\partial}{\partial z}\left(\rho k_z \frac{\partial c/\rho}{\partial z}\right) + S,
\end{aligned}
\tag{A.14}
$$

where c is the tracer concentration ($[c] = kg/m^3$); $\mathbf{u} = (u_x, u_y, u_z)$ is the wind field; $v_{sj} =$

$(0,0,v_{sj})$ is the settling velocity of the particle class j $(j=1:n_p)$; $k_x = k_y$ are the horizontal diffusion coefficients ($[k]=m^2/s$); k_z is the vertical diffusion coefficient; ρ is the fluid (air) density, and S is the source term ($[S]=kg/m^3s$).

Equation (A.14) can be generalized to other coordinate systems (X,Y,Z) using scaled quantities as:

$$\begin{aligned}&\frac{\partial C}{\partial t}+U_X\frac{\partial C}{\partial X}+U_Y\frac{\partial C}{\partial Y}+(U_Z-V_{sj})\frac{\partial C}{\partial Z}+C\,(\nabla\cdot\mathbf{U})-C\frac{\partial V_{sj}}{\partial Z}=\\&\frac{\partial}{\partial X}\left(\rho_*K_X\frac{\partial C/\rho_*}{\partial X}\right)+\frac{\partial}{\partial y}\left(\rho_*K_Y\frac{\partial C/\rho_*}{\partial Y}\right)+\frac{\partial}{\partial Z}\left(\rho_*K_Z\frac{\partial C/\rho_*}{\partial Z}\right)+S_*,\end{aligned}\tag{A.15}$$

or in conservative form:

$$\begin{aligned}&\frac{\partial C}{\partial t}+\frac{\partial}{\partial X}(U_XC)+\frac{\partial}{\partial Y}(U_YC)+\frac{\partial}{\partial Z}\left[(U_Z-V_{sj})C\right]=\\&\frac{\partial}{\partial X}\left(\rho_*K_X\frac{\partial C/\rho_*}{\partial X}\right)+\frac{\partial}{\partial y}\left(\rho_*K_Y\frac{\partial C/\rho_*}{\partial Y}\right)+\frac{\partial}{\partial Z}\left(\rho_*K_Z\frac{\partial C/\rho_*}{\partial Z}\right)+S_*.\end{aligned}\tag{A.16}$$

In particular, in a spherical terrain-following coordinate system $(X,Y,Z)=(\lambda,\phi,Z)=(\lambda,\phi,z-h(\lambda,\phi))$, where z is the elevation above sea level, h is the topographic elevation, λ is longitude and ϕ is latitude, the scaled quantities are given in Table A.1

Scaled parameter	Scaling factor
Coordinates	$dX=d\lambda=R_edx;\quad dY=d\phi=R_edy;\quad dZ=dz$
Velocity	$U_X=u_x/\sin\gamma;\quad U_Y=u_y;\quad U_Z=u_z$
Settling velocity	$V_{sj}=v_{sj}$
Diffusion Coefficients	$K_X=k_x/\sin^2\gamma;\quad K_Y=k_y;\quad K_Z=k_z$
Concentration	$C=c\sin\gamma$
Density	$\rho_*=\rho\sin\gamma$
Source Term	$S_*=S\sin\gamma$

Table A.1: Scaling factors for a spherical terrain-following coordinate system where γ is the colatitude and R_e the radius of the Earth (spherical assumption).

The computational domain consists of a brick $x\in[x_0,x_1]$, $y\in[y_0,y_1]$,$z\in[z_0,z_1]$ (z positive upwards) divided in n_x, n_y, and n_z points respectively (not necessarily equally spaced in z). Free flow conditions are assumed at all boundaries of the computational domain. The choice of the boundary conditions is important in order to avoid absorption or reflection from these boundaries. Different boundary conditions are imposed: for outgoing flux, zero derivative conditions, whereas for incoming flux, null concentrations at boundaries are assumed:

$$\begin{aligned}
&\text{At } x = x_0 \begin{cases} C_{0jk} = 0 & \text{if } U_{1jk} \geq 0 \\ C_{0jk} = C_{1jk} & \text{if } U_{1jk} < 0 \end{cases} \\
&\text{At } x = x_1 \begin{cases} C_{n+1jk} = 0 & \text{if } U_{njk} \leq 0 \\ C_{n+1jk} = C_{njk} & \text{if } U_{njk} > 0 \end{cases}
\end{aligned} \tag{A.17}$$

and the same along y and z directions.

Bibliography

[1] Extrae. `http://www.bsc.es/computer-sciences/extrae`, 2015.

[2] FALL3D. `http://bsccase02.bsc.es/projects/fall3d`, 2015.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[4] J. L. Alonso, X. Andrade, P. Echenique, F. Falceto, D. Prada-Gracia, and A. Rubio. Efficient formalism for large-scale ab initio molecular dynamics based on time-dependent density functional theory. *Physical Review Letters*, 101, August 2008.

[5] Ulf Andersson. *Time-Domain Methods for the Maxwell Equations*. PhD thesis, Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, 2001.

[6] Mauricio Araya-Polo and Raúl de la Cruz. Semi-stencil algorithm: Improving data locality and reuse in stencil computation. 14th SIAM Conference on Parallel Processing for Scientific Computing, Seattle, Washington, February 2010. Part of CP4 PDEs (Talk).

[7] Mauricio Araya-Polo and Raúl de la Cruz. Performance model for 3D stencil computation. 2012 Rice Oil & Gas HPC Workshop, Rice University, Houston, March 2012. Parallel session A: Benchmarking, Optimization & Performance (Talk).

[8] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. High-performance seismic acoustic imaging by reverse-time migration on the Cell/B.E. architecture. *ISCA2008 - WCSA2008*, 2008.

[9] Mauricio Araya-Polo, Félix Rubio, Mauricio Hanzich, Raúl de la Cruz, José María Cela, and Daniele P. Scarpazza. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Scientific Programming, Special Issue on the Cell Processor*, 17, 2009.

[10] 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.

[11] C. Bonadonna, A. Folch, S. Loughlin, and H. Puempel. Future developments in modelling and monitoring of volcanic ash clouds: outcomes from the first IAVCEI-WMO

workshop on Ash Dispersal Forecast and Civil Aviation. *Bulletin of Volcanology*, 74:1–10, 2012.

[12] Axel Brandenburg. *Computational aspects of astrophysical MHD and turbulence*, volume 9. CRC, April 2003.

[13] Repsol - BSC. Kaleidoskope project. `http://www.bsc.es/projects/kaleidoskope_tmp`, 2008.

[14] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, New York, NY, USA, 1991. ACM.

[15] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, F. Lorenzen X. Andrade, M. A. L. Marques, E. K. U. Gross, and A. Rubio. Octopus: a tool for the application of time-dependent density functional theory. *Physica Status Solidi: Towards Atomistic Materials Design*, 243:2465 – 2488, June 2006.

[16] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[17] E. Collini, S. Osores, A. Folch, J.G. Viramonte, G. Villarosa, and G. Salmuni. Volcanic ash forecast during the June 2011 Cordón Caulle eruption. *Natural Hazards*, 66(2):389–412, 2013.

[18] A. Costa, G. Macedonio, and A. Folch. A three-dimensional Eulerian model for transport and deposition of volcanic ashes. *Earth and Planetary Science Letters*, 241(3£4):634–647, 2006.

[19] Daimler-Benz. Mercedes-Benz W111 fintail crash tests, September 1959. The first crash test in the history of Mercedes-Benz.

[20] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning stencil computations on multicore and accelerators. In *Scientific Computing on Multicore and Accelerators*, pages 219–253. CRC Press, 2010.

[21] 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, 2009.

[22] 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 *SC '08: Proceedings*

of the 2008 ACM/IEEE conference on Supercomputing, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *Proceedings of iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[24] Raúl de la Cruz and Mauricio Araya-Polo. Towards a multi-level cache performance model for 3D stencil computation. In *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*, pages 2146–2155, 2011.

[25] Raúl de la Cruz and Mauricio Araya-Polo. Algorithm 942: Semi-stencil. *ACM Transactions on Mathematical Software (TOMS)*, 40(3):23:1–23:39, April 2014.

[26] Raúl de la Cruz and Mauricio Araya-Polo. Modeling stencil code optimizations. 16th SIAM Conference on Parallel Processing for Scientific Computing, Portland, Oregon, February 2014. CP8: Performance Optimization for Stencils and Meshes (Talk).

[27] Raúl de la Cruz and Mauricio Araya-Polo. Modeling stencil computations on modern HPC architectures. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 149–171. Springer International Publishing, 2015.

[28] Raúl de la Cruz and Mauricio Araya-Polo. Using modeling to develop stencil codes. 2015 Rice Oil & Gas HPC Workshop, Rice University, Houston, March 2015. Coarse-grained Seismic Algorithms (Talk).

[29] Raúl de la Cruz, Mauricio Araya-Polo, and José María Cela. Introducing the *Semi-stencil* algorithm. In *Parallel Processing and Applied Mathematics, 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*, pages 496–506, 2009.

[30] Raúl de la Cruz, Arnau Folch, Pau Farré, Javier Cabezas, Nacho Navarro, and José María Cela. Optimization of atmospheric transport models on HPC platforms. *Computational Geosciences*, 2015. (Submitted).

[31] Raúl de la Cruz, Mauricio Hanzich, and José María Cela. Stencil computations: from academia to industry. 16th SIAM Conference on Parallel Processing for Scientific Computing, Portland, Oregon, February 2014. Part of MS66 Optimizing Stencil-based Algorithms - Part II of II (Talk).

[32] Raúl de la Cruz, Mauricio Hanzich, Arnau Folch, Guillaume Houzeaux, and José María Cela. Unveiling WARIS code, a parallel and multi-purpose FDM framework. In *Numerical Mathematics and Advanced Applications - ENUMATH 2013 - Proceedings of ENUMATH 2013, the 10th European Conference on Numerical Mathematics and Advanced Applications, Lausanne, Switzerland, August 2013*, pages 591–599, 2013.

[33] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, December 1993.

[34] Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An empirical study of intel xeon phi. *CoRR*, abs/1310.5842, 2013.

[35] A. Folch, A. Costa, and G. Macedonio. FALL3D: A computational model for transport and deposition of volcanic ash. *Computers & Geosciences*, 35(6):1334–1342, June 2009.

[36] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *19th ACM International Conference on Supercomputing*, pages 361–366, June 2005.

[37] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[38] C. De Groot-Hedlin. A finite difference solution to the Helmholtz equation in a radially symmetric waveguide: Application to near-source scattering in ocean acoustics. *Journal of Computational Acoustics*, 16:447–464, 2008.

[39] C. De Groot-Hedlin. A finite difference solution to the Helmholtz equation in a radially symmetric waveguide: Application to near-source scattering in ocean acoustics. *Journal of Computational Acoustics*, 16:447–464, 2008.

[40] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Efficient analytical modelling of multi-level set-associative caches. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, HPCN Europe '99, pages 473–482, London, UK, UK, 1999. Springer-Verlag.

[41] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, April 2010.

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

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

[44] W. L. Ko, R. D. Quinn, and L. Gong. Reentry heat transfer analysis of the space shuttle orbiter. In *NASA. Langley Research Center Computational Aspects of Heat Transfer in Struct. p 295-325 (SEE N82-23473 14-34)*, pages 295–325, 1982.

[45] Jean Kormann, Pedro Cobo, and Andres Prieto. Perfectly matched layers for modelling seismic oceanography experiments. *Journal of Sound and Vibration*, 317(1-2):354 – 365, 2008.

[46] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM.

[47] B. Langmann, A. Folch, M. Hensch, and V. Matthias. Volcanic ash over Europe during the eruption of Eyjafjallajökull on Iceland, April-May 2010. *Atmospheric Environment*, 48:1–8, 2012.

[48] CA. Applied Numerical Algorithms Group (ANAG) Lawrence Berkeley National Laboratory, Berkeley. Chombo. `http://seesar.lbl.gov/ANAG/software.html`, 2015.

[49] G.L. Lore, D.A. Marin, E.C. Batchelder, W.C. Courtwright, R.P. Desselles, and R.J. Klazynski. 2000 assessment of conventionally recoverable hydrocarbon resources of the Gulf of Mexico and Atlantic outer continental shelf. Technical report, U.S. Department of the Interior, Minerals Management Service, Gulf of Mexico OCS Region, Office of Resource Evaluation, 2001.

[50] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 68.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[51] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Trans. Parallel Distrib. Syst.*, 8:193–209, February 1997.

[52] Gabriel Marin, Collin McCurdy, and Jeffrey S. Vetter. Diagnosis and optimization of application prefetching performance. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 303–312, New York, NY, USA, 2013. ACM.

[53] John McCalpin and David Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical report, Report 379, Department of Computer Science, Rutgers University, 1999.

[54] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. `http://www.cs.virginia.edu/stream/`.

[55] John D. McCalpin. STREAM2. `https://www.cs.virginia.edu/stream/stream2/`, 2015.

[56] Collin McCurdy, Gabriel Marin, and Jeffrey S Vetter. Characterizing the impact of prefetching on scientific application performance. In *International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13)*, Denver, CO, 2013.

[57] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18:424–453, July 1996.

[58] George A. McMechan. A review of seismic acoustic imaging by reverse-time migration. *International Journal of Imaging Systems and Technology*, 1(1):18–21, 1989.

[59] Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 73–82, New York, NY, USA, 2014. ACM.

[60] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled data prefetching. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[61] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[62] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical Report UCB/CSD-04-1335, EECS Department, University of California, Berkeley, 2004.

[63] United States. Office of Aviation Safety. Full-scale transport controlled impact demonstration program: Aircraft accident investigation experiment and report of investigation. `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19880000639.pdf`, December 1984.

[64] Brazil official cites giant oil-field discovery. Technical report, The International Herald Tribune, Associated Press, April 2008.

[65] S. Operto, J. Virieux, P. Amestoy, L. Giraud, and J. Y. L'Excellent. 3D frequency-domain finite-difference modeling of acoustic wave propagation using a massively parallel direct solver: a feasibility study. *SEG Technical Program Expanded Abstracts*, pages 2265–2269, 2006.

[66] Francisco Ortigosa, Mauricio Araya-Polo, Raúl de la Cruz, and José M. Cela. Seismic imaging and the road to peta-scale capacity. *17th SPE - 70th EAGE Conference*, 2008.

[67] Francisco Ortigosa, Mauricio Araya-Polo, Félix Rubio, Mauricio Hanzich, Raúl de la Cruz, and José Maria Cela. Evaluation of 3D RTM on HPC platforms. *SEG 2008, Las Vegas, USA*, 2008.

[68] Francisco Ortigosa, Hongbo Zhou, Santiago Fernandez, Mauricio Hanzich, Mauricio Araya-Polo, Felix Rubio, Raúl de la Cruz, and José M. Cela. Benchmarking 3D RTM on HPC platforms. Instituto Argentino del Petroleo y del Gas, November 2008.

[69] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[70] Vincent Pillet, Vincent Pillet, Jesús Labarta, Toni Cortes, Toni Cortes, Sergi Girona, Sergi Girona, and Departament D'arquitectura De Computadors. PARAVER: A tool to visualize and analyze parallel code. 1995.

[71] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10, New York, NY, USA, 2011. ACM.

[72] Gabriel Rivera and Chau Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proc. ACM/IEEE Supercomputing Conference (SC 2000)*, page 32, November 2000.

[73] Anne Rogers and Kai Li. Software support for speculative loads. *SIGPLAN Not.*, 27(9):38–50, 1992.

[74] A. Russell and R. Dennis. NARSTO critical review of photochemical models and modelling. *Atmospheric environment*, 34:2261–2282, 2000.

[75] Muhammad Shafiq, M. Pericas, Raúl de la Cruz, Mauricio Araya-Polo, Nacho Navarro, and E. Ayguade. Exploiting memory customization in FPGA for 3D stencil computations. In *IEEE International Conference on Field-Programmable Technology*, 2009.

[76] J. Steppeler. A three dimensional global weather prediction model using a finite element scheme for vertical discretization. *International Journal for Numerical Methods in Engineering*, 27, 1989.

[77] Erich Strohmaier. Top500 supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[78] Robert Strzodka, Mohammed Shaheen, and Dawid Pajak. Impact of system and cache bandwidth on stencil computation across multiple processor generations. In *Proc. Workshop on Applications for Multi- and Many-Core Processors (A4MMC) at ISCA 2011*, June 2011.

[79] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 261–271, New York, NY, USA, 1994. ACM.

[80] Vidar Thomée. From finite differences to finite elements: a short history of numerical analysis of partial differential equations. *J. Comput. Appl. Math.*, 128(1-2):1–54, 2001.

[81] Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics*, volume 6067 of *PPAM'09*, pages 615–624. Springer-Verlag, 2009.

[82] Jan Treibig, Georg Hager, and Gerhard Wellein. Multi-core architectures: Complexities of performance prediction and the impact of cache topology. *CoRR*, abs/0910.4865, 2009.

[83] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.

[84] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the Cell processor. 35(3), June 2007.

[85] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353349.

[86] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

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