UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

COMPILER AND RUNTIME SUPPORT FOR
THE EXECUTION OF SCIENTIFIC CODES
WITH UNSTRUCTURED DATASETS ON
HETEROGENEOUS PARALLEL
ARCHITECTURES

Doctoral Thesis

Pablo Barrio López-Cortijo
Ingeniero en Informática
M.S. in High Performance Computing
2017

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE MADRID

COMPILER AND RUNTIME SUPPORT FOR
THE EXECUTION OF SCIENTIFIC CODES
WITH UNSTRUCTURED DATASETS ON
HETEROGENEOUS PARALLEL
ARCHITECTURES

Doctoral Thesis

Author:

Pablo Barrio López-Cortijo
Ingeniero en Informática
M.S. in High Performance Computing

Advisor:

Carlos Carreras Vaquer
Profesor Titular del Dpto. de Ingeniería Electrónica

2017

Contents

	Page
Abstract	v
Resumen	vii
Acknowledgements	ix
List of Figures	xi
List of Tables	xv
1 Introduction	1
1.1 Motivation	2
1.1.1 FPGA acceleration of codes with unstructured-mesh datasets	2
1.1.2 Design automation	4
1.2 Challenges	5
1.2.1 Memory bandwidth limitations	5
1.2.2 Irregular data accesses	6
1.2.3 Code transformations for distributed systems	7
1.2.4 Automated insertion of communications	8
1.3 Structure of the thesis	9
1.4 Contributions	9
2 Memory optimization for unstructured meshes	13
2.1 Background and related work	14
2.1.1 Unstructured meshes	14
2.1.2 The memory wall	17
2.1.3 Improving data locality	19
2.1.4 Reducing data size	23
2.1.5 Memory systems for FPGA accelerators	25
2.2 Locality optimization with mesh sorting	26
2.2.1 The BFsort breadth-first algorithm	27

2.2.2	Window size results	30
2.2.3	Seed selection	34
2.3	A streaming methodology for pipelined kernels	42
2.3.1	Hybrid sequential/random access	42
2.3.2	Coding and compression with BFsort	43
2.3.3	Implementation details	46
2.4	Performance results	52
2.4.1	Data locality improvements on a multicore	53
2.4.2	Streaming methodology on an FPGA accelerator	54
2.5	Conclusions	60
3	Refactoring of sequential codes for distributed systems	61
3.1	Related work	63
3.1.1	Code refactoring	63
3.1.2	Distributed control flow	64
3.2	High-level view of the compiler toolchain	65
3.3	Block outlining	69
3.3.1	Initiator search	70
3.3.2	Updating the control structures	71
3.3.3	Data structures	74
3.4	Remote Block Branches (RBBs)	74
3.4.1	Compiler passes	78
3.4.2	The runtime library	79
3.5	Results	82
3.5.1	Results of refactoring with vectorization on a multicore	82
3.5.2	Remote Block Branches on synthetic benchmarks	84
3.5.3	2-dimensional matrix multiplication	91
3.6	Conclusions	94
3.6.1	Future work	94
4	Automated pipelining with the inspector-executor paradigm	97
4.1	Background and Related Work	99
4.1.1	Skeletons	100
4.1.2	The ROSE compiler	101
4.1.3	Inspector-Executor	101
4.1.4	IEC	103
4.2	Modifications to IEC for pipelining	108
4.2.1	Load partitioning in two dimensions	109
4.2.2	Data dependencies	110
4.2.3	Changes to the inspector	113
4.2.4	Changes to the executor	115

4.3	Performance investigation	119
4.3.1	Performance results of the pipelined <i>quake</i>	120
4.3.2	Parallel profiling of the pipelined code	123
4.4	Conclusions	126
5	Conclusions and future work	129
5.1	Summary of the conclusions of the Thesis	129
5.2	Future lines of work	131
	Bibliography	133
	Glossary	149

Contents

Abstract

Simulation codes based in the discretization of time and space for solving systems of partial differential equations are used nowadays in relevant industrial sectors. These include, for example, finite volumes and finite element methods typically used in Computational Fluid Dynamics (CFD). These simulation applications are widely used various industries such as aeronautics, automotive or weather prediction. More recently, they have been used in novel applications such as the simulation of blood streams for medical diagnosis, or the design of wind turbines for energy generation. The complexity of these simulations and the sheer size of the datasets used often require a considerable amount of computing power in order to obtain results in a reasonable time. In the last years, the optimization of these applications has been a priority for many companies, and even real time performance is now often considered as a stretch goal for optimization attempts.

High Performance Computing (HPC) systems have been used for some time now to run these simulations, achieving reasonable speedups by partitioning the datasets and running the simulation in several processors. However, the need to run them in as little time as possible, the increase in the dataset sizes to allow for higher accuracy, and limitations in the multi-process scalability of CFD simulations, have resulted in joint worldwide efforts to achieve the ever-increasing performance and problem size objectives by means of newer, disruptive technologies.

This thesis approaches the problem of optimizing the execution of these codes with the help of Heterogeneous HPC systems. These systems differ from standard, homogeneous HPC systems in that the architectures of the processing elements used as building blocks differ from each other. A special interest of this work is to analyze the feasibility of using Field-Programmable Gate Arrays (FPGAs) in these systems as accelerators to scientific simulations. Because these devices are essentially reconfigurable hardware, they allow a finer-grain parallelism than general-purpose processors, which translates into a higher throughput when computational kernels are ported to them. Additionally, FPGAs achieve levels of power efficiency that are currently unparalleled by any existing mainstream computing device. Unfortunately, the novelty of this approach implies that the

programming effort required to implement such solutions is still high compared to other well-known accelerating technologies such as General-Purpose Graphics Processing Units (GPGPUs).

FPGAs also show limitations with external memory throughput. Due to the characteristics of data dependencies in the datasets used for these codes, this problem might render FPGAs useless for the optimization of such applications. A deep optimization of the data transfer mechanisms is vital to ensure that the high-performance capabilities of these reconfigurable devices are not shadowed by their lower input/output capabilities as compared to other existing processing devices.

The algorithms, methodologies and software libraries introduced throughout this Thesis improve data transfer of the aforementioned codes in FPGA-based parallel heterogeneous systems, while also reducing the development effort required to implement these. Specifically, we propose a methodology to reduce the size of the datasets and transfer them efficiently to the FPGA, as well as two compiler and runtime techniques to automate the parallelization of the codes suitable for heterogeneous systems, one focused on control flow distribution and another one based on pipelining of loop sequences. The latter are techniques at the system level, therefore they are independent from the architectures used in the heterogeneous system.

Although the original purpose of this work was to optimize CFD simulations, it has become clear that the techniques proposed are applicable to a more general set of applications. In the future, HPC systems could use FPGAs in the same way that they now benefit from GPGPU technologies, but with the added benefits of reconfigurable hardware.

Resumen

Las simulaciones computacionales basadas en la discretización temporal y espacial para la resolución de sistemas de ecuaciones en derivadas parciales se utilizan hoy en día en sectores industriales de gran relevancia económica. Un ejemplo significativo de dichas aplicaciones son las simulaciones de dinámica de fluidos. Estas simulaciones se utilizan constantemente en industrias tales como la aeronáutica o la automoción. Más recientemente, se han identificado nuevos campos de aplicación como la simulación del torrente sanguíneo para el diagnóstico médico, o el diseño de turbinas eólicas para la generación de energía. La complejidad de dichas simulaciones y el propio tamaño de los conjuntos de datos usados hacen que requieran de una capacidad de procesamiento considerable si se quieren obtener resultados en un tiempo razonable. En los últimos años, la optimización de estas aplicaciones se ha convertido en una prioridad para diversos organismos, hasta el punto de considerar aplicaciones en tiempo real como un objetivo a largo plazo.

Estas simulaciones han venido usando recursos de Computación de Altas Prestaciones (HPC, por sus siglas en inglés), los cuales obtienen aceleraciones razonables mediante el particionado de datos y su ejecución en múltiples procesadores. Aún así, la necesidad de ejecutarlas en el menor tiempo posible, el crecimiento de los conjuntos de datos para incrementar la exactitud de los resultados, y las limitaciones de escalabilidad de los códigos de CFD, han motivado la aparición de iniciativas conjuntas entre varios países, con el objetivo de conseguir los objetivos de rendimiento y tamaño, siempre en crecimiento, a través de tecnologías novedosas e incluso disruptivas.

La presente tesis aborda el tema de la optimización de dichos códigos con la ayuda de sistemas heterogéneos de altas prestaciones. Dichos sistemas se diferencian de los sistemas homogéneos en el hecho de que las arquitecturas de sus elementos de procesamiento son distintas entre sí. Se pone un especial énfasis en el análisis de viabilidad en el uso de FPGAs en dichos sistemas como aceleradores para simulaciones científicas. Al estar estos dispositivos basados en hardware reconfigurable, permiten un grado más fino de paralelismo que los procesadores de propósito general, lo cual se traduce en un incremento del rendimiento de las porciones de código más relevantes cuando éstas se asignan a la FPGA. Por otra

parte, las FPGAs obtienen unos niveles de eficiencia energética muy por encima de cualquier otro dispositivo de cómputo disponible hoy en día. Por desgracia, la novedad de este tipo de propuestas también significa que el esfuerzo de desarrollo requerido para implementarlas es todavía significativamente superior a otras tecnologías más establecidas, como por ejemplo las GPUs de propósito general (GPGPUs, por sus siglas en inglés).

Las FPGAs también muestran limitaciones a la hora de manejar datos procedentes de memorias externas. Debido a las características de las dependencias en los conjuntos de datos utilizados en muchas de las aplicaciones CFD, este problema podría suponer la no idoneidad de las FPGAs como alternativa para optimizar dichas aplicaciones. Por ello, es de vital importancia asegurarse de optimizar las transferencias de datos en profundidad, de forma que la alta capacidad de cómputo de la FPGA no se vea limitada por sus capacidad de transferencia, relativamente pobre si se compara con otros dispositivos de procesamiento.

Los algoritmos, metodologías y librerías de software presentados a lo largo de la Tesis están enfocadas a la mejora de la transferencia de datos de códigos CFD en arquitecturas paralelas heterogéneas con aceleradores FPGA, tratando de reducir al mismo tiempo el coste de implementación de dichos sistemas. Específicamente, se propone una metodología de reducción del tamaño de los conjuntos de datos y de transferencia desde y hacia la FPGA, así como dos propuestas de automatización para la paralelización de códigos enfocada a sistemas heterogéneos y basadas en compiladores con la asistencia de librerías de comunicación, la primera de las cuales está basada en la distribución del flujo de control, mientras que la segunda se basa en el *pipelining* de secuencias de bucles. Éstas últimas son técnicas a nivel de sistema, y por tanto son independientes de las arquitecturas presentes en el sistema heterogéneo.

A pesar de que el propósito original del trabajo era la optimización de simulaciones de dinámica de fluidos, se ha observado que las técnicas propuestas son aplicables a un conjunto de problemas más general. En el futuro, muchos sistemas HPC podrían beneficiarse de la tecnología disponible en las FPGAs de la misma forma en la que se benefician actualmente de las GPUs, con los beneficios adicionales de rendimiento y consumo energético que ofrece el hardware reconfigurable.

Acknowledgements

First and foremost, this Thesis would have never been possible without Carlos Carreras. This Thesis belongs to him as much as to me, from the initial ideas to the results. From him I have learned the importance of research standards and the value of analyzing ideas to the detail. I would like to thank him for his support and perseverance over the last months.

I have been very lucky to work with the people at the Integrated Systems Laboratory (*LSI*). I would like to thank Roberto Sierra, Enrique Sedano, José Antonio Fernández, Ángel Fernández and Juan Antonio López for the valuable suggestions and numerous proofreading iterations prior to the publication of the articles that gave birth to this work. I would also like to acknowledge the computer resources and technical assistance provided by the *Centro de Supercomputación y Visualización de Madrid (CeSViMa)*.

I should express my gratitude to the reviewers of the manuscript, who have kindly agreed to provide me with valuable feedback and be part of the board of examiners: José Manuel Moya, Gabriel Caffarena, Francisco Gómez, Christian Plessl and Manuel Hermenegildo. I have the honor to have met them personally during my academic life, and they are all fantastic professionals. I hope to continue learning from them in the future.

Throughout the years, I have seen people in the team come and go. This work has a bit of all of them. In particular, Pablo Ituero provided me with a L^AT_EX template, dispossessing me of any excuses to not deal with the moment of writing the first page, feared by many. Fernando García has proven to be an invaluable asset over the last stages of the writing, when I was thousands of miles away from Madrid, and greatly improved my presentation skills. With him and Javier Agustín, I traveled far in pursuit of the perfect hamburger.

And because gastronomy is my religion, I would like to thank my beloved colleagues of the *Peña Gastronómica Colegas de Valdeón* for the numerous occasions where they brought me out of my imprisonment to enjoy some time among friends and food. I am lucky to have met them, especially Eduardo, Ignacio and Germán, fellow PhD students, with which I have shared many thoughts about science and research.

My parents deserve half of the credit for my achievements. They have always supported my projects even when these meant being far from each other. A big thanks to my sisters Ana and Inés for being there for me even when we could not see each other. Hard things look easy with a family like this.

Last but not least, I would like to thank my partner Carol for her unconditional support, especially over the last months when I gradually transformed into a grunting mass of hatred. Or she should thank me for all the new useful concepts that she learnt in the process, such as the many possible layouts of a communication network, or the difference between SysVinit and Systemd. I am sure she will put that to good use in her everyday job as a publicist.

List of Figures

2.1	Structured and unstructured meshes. Only a) is unstructured. Courtesy of Adrian Lange, Wikipedia.	14
2.2	A two-dimensional mesh and its associated graphs. The dual graph has an extra edge for each cell that connects to the imaginary cell surrounding the mesh.	15
2.3	Manifoldness. Mesh a is a manifold. Mesh b is not a manifold, since it does not meet requirement 2. Mesh c is not a manifold, since it does not meet requirement 1. Courtesy of Prof. Mario Botsch, U. Bielefeld.	17
2.4	Triangle strips. The first is the original depth-first traversal; the second is the breadth-first traversal as used by Mitra and Chiueh.	20
2.5	Sorting the mesh appropriately results in smaller windows. Figures 2.5a and 2.5b show the adjacency matrices. Figure 2.5c shows the subsets of vertices that belong to two windows: vertex 8 with a sequence based on vertex identifiers, and vertex 6 with a Cuthill-McKee sorting.	22
2.6	Difference between bandwidth and window due to edge processing	23
2.7	Three sorting methods applied to a mesh. For each method, the upper line represents the sequence generated by the sorting algorithm, whereas the lower line represents the state of the intermediate buffer during the computation.	26
2.8	Customizable BFsrt	28
2.9	Random and clockwise tie resolution strategies.	29
2.10	Breadth-first evolution by cells (upper line) as proposed by Mitra, and by vertices (lower line) as proposed in this work, together with the FIFO queues used to build the order. The lightly coloured vertices and cells are visited but not closed. The dark-coloured vertices are finished.	30
2.11	Complete and zoomed views of the NACA test meshes. The lightly coloured vertices belong to the maximum frontier.	32

2.12	(Continues Figure 2.11) Complete and zoomed views of the NACA test meshes.	33
2.13	Maximum vertex window with breadth-first sequences in terms of vertices and cells.	34
2.14	Required size of the FPGA buffer in percentage over the complete mesh for various vertex-based sortings.	35
2.15	Maximum frontiers obtained with the best seed (left) and worst seed (right). The region of interest is shown in the main figures, whereas the boxed figures show the complete mesh.	36
2.16	Bipartition seed selection	38
2.17	Execution of the bipartition algorithm on the NACA 0012 mesh.	39
2.18	Maximum window size obtained by seed selection algorithms.	40
2.19	Execution time of seed selection algorithms.	41
2.20	A traditional cache system (left) compared to a custom hybrid sequential/random system (right) on an FPGA.	43
2.21	Frame coding without ID compression and with ID compression.	45
2.22	Target architecture showing the data movements between host and accelerator.	47
2.23	Interface between the memory and the kernel for the read-write part of the mesh. At each iteration, the read and write memories are swapped.	47
2.24	Left: edge-based read-only frame. Right: vertex-based read-write frame. Dark blocks represent vertices; light blocks represent edges in the connectivity.	51
2.25	Compression ratio (CR) of the encoded connectivity data with different sorting methods and with or without quantization.	55
2.26	Compression ratio of the algorithmic data	58
2.27	Overall compression results in percentage of the original mesh. The first bar represents the original mesh, while the second bar represents the compressed mesh after applying the discussed techniques except for sorting. The last bar includes sorting.	58
3.1	Proposed compilation toolchain.	66
3.2	RSD description and graph representation of a machine with two quad-core processors, 2.4 GHz and 4 GB main memory.	67
3.3	Graph representing the architecture defined by the Resource and Service Description (RSD) file shown in Figure 3.2.	68
3.4	Code generation steps	69

3.5	CFG of a function at different stages of the refactoring. The figure on the left shows the device partitioning, where the Basic Blocks (BBs) in the highlighted boxes are mapped to one device and the BBs in the regular boxes are mapped to another. The center figure shows the resulting initiators, and the right figure shows the newly-partitioned functions.	71
3.6	The initiator search finds the entry blocks for future functions and marks BBs with its root initiator.	72
3.7	Updating a phi node	73
3.8	Transformation of branch instructions into a distributed application with Remote Block Branches.	76
3.9	Startup, RBB call and shutdown process. A closed lock appears when a thread is put into sleep. An open lock represents the wakeup condition.	81
3.10	Performance of the partitioned codes with different combinations of partitioning and vectorization.	84
3.11	seq benchmark: three basic blocks in sequence.	85
3.12	nested benchmark: two nested loops.	86
3.13	improper benchmark: an improper region consisting in a single loop with two possible entry and exit points.	87
3.14	Combined startup and shutdown time of the RBB system compared to the MVAPICH MPI distribution.	90
3.15	Performance of chained Remote Block Branches as generated by a loop.	90
3.16	RBB performance variation with different parameter sizes.	91
3.17	2mm kernel in C.	92
3.18	Performance of the 2mm kernel with RBBs	94
4.1	Pseudocode structure of the codes that can be considered for parallelization.	98
4.2	Creation of inspector and executor	102
4.3	Pseudocode example of inspector generation	106
4.4	Example of an IEC hypergraph	107
4.5	Partial diagram of a seismic simulation	111
4.6	Two approaches to pipeline dependency communication	112
4.7	A blocked loop in the pipeline as a consequence of a dependency in an unrelated loop	116
4.8	perf analysis of the sequential code	122
4.9	perf analysis of the pipelined code	122
4.10	Distribution of execution time per thread among different functions	124
4.11	Detailed profile in process 0	125

List of Figures

4.12 Detailed profile in process 7	125
4.13 Detailed profile in process 14	125

List of Tables

2.1	Characteristics of the datasets	31
2.2	Frequency and area results, RO frame	51
2.3	Frequency and area results, RW frame	52
2.4	Timings and speedups of the test code with two vertex sequences	53
2.5	Data-L1 and L2 cache misses, measured as a percentage of the total accesses to the memory system.	53
2.6	Differential compression variation in variables from the CFD ap- plication. Sizes given as percentage of the original mesh.	56
2.7	Bit-widths for different quantization choices	56
2.8	Compression ratio with two tie resolution strategies	59
3.1	Benchmark timings (in seconds) with three implementation choices and different locality of communications.	89
3.2	Timings of the 2mm original and partitioned code with different data sizes, and calculated overhead.	93
4.1	Characteristics of the datasets used with <i>quake</i>	121
4.2	Execution time comparison between the original and the pipelined <i>quake</i>	121

Chapter 1

Introduction

HPC is becoming popular as a consequence of the increasing computational needs of scientific and industrial applications. A broad spectrum of algorithms demand a variety of computational resources that traditional general-purpose architectures find hard to cater for. Also, many of these applications manipulate massive datasets that require memory systems of considerable size. These facts, in conjunction with power efficiency requirements, have motivated a growing interest in heterogeneous distributed systems [BDH⁺10][DMM⁺10][SKS⁺07].

Despite their obvious advantages, heterogeneous distributed architectures require a considerable effort from the application developers. A sequential application must undergo several analysis and transformation steps, such as performance estimation, parallelization, dataset partitioning and code refactoring, in order to accommodate the code for the different architectures available in the system. Most of these development stages are currently done manually, and are often more complex than the design of the serial code itself [SQJ11]. There is a general consensus that the compiler should perform the above steps with minimal human participation and, hence, automation by means of intelligent compilers is an active field of research [WQZ⁺13][BGA12][Ozt11].

As opposed to shared memory systems, distributed systems require distributing the data structures into many address spaces. This implies that the data transfer between different parts of the system must be made explicit in the code. This communication paradigm is known as message passing. Analyzing the application and its data access patterns, and inserting such communications, is a difficult process to automate, and further complicates the portability of existing legacy codes.

Heterogeneity in HPC clusters mainly comes in the form of accelerators. GPGPUs are currently the most popular. However, it has been proved that FPGAs outperform the former for several types of applications, while achieving better power efficiency in terms of Floating-point Operations per Second (FLOPS) per

Watt; this will be discussed in greater detail in Section 1.1.1. Since power efficiency is increasingly important as the computing power of clusters and data centers grow (see the works by Salinas-Hilburg et al. [SHZRM⁺16] or Zapater et al. [ZTM⁺15]), FPGAs are initially good candidates to build more cost-effective systems. On the other hand, the use of FPGAs is still under research for many other types of applications, specially when complex data dependencies are involved. This is the case of applications using unstructured meshes, which remain one of the big challenges of HPC.

This thesis approaches some of the problems that arise when parallelizing programs that represent their data with unstructured meshes. The applicability of heterogeneous systems and, in particular, of FPGAs clusters, to unstructured-mesh applications, has received little attention in industry and academia. As a means to analyze its feasibility, this work focuses on two different but related topics: FPGA accelerators, and distributed heterogeneous systems at the system level. However, many of the solutions presented are applicable beyond the scope of the thesis.

The structure of this introductory chapter is as follows. Section 1.1 introduces the motivations of the thesis. Section 1.2 describes the specific problems that will be analysed in this thesis. Section 1.3 explains the structure of the book and the ideas that glue the chapters together. Finally, Section 1.4 enumerates the contributions of the work, which have resulted in a number of project reports, publications and a patent granted.

1.1 Motivation

1.1.1 FPGA acceleration of codes with unstructured-mesh datasets

GPGPUs are the predominant accelerators in the market. However, FPGAs have recently attracted considerable attention. In the last years, many studies have compared the performance of both accelerators with typical scientific and industry codes. None of them obtain conclusive evidence of either of these accelerators outperforming the other in a broad range of applications. The following is a representative selection of such comparative studies.

Che et al. [CLS⁺08] compare the performance of GPGPUs vs FPGAs on three benchmarks. A qualitative comparison suggests that GPGPUs are most efficient with codes with little data dependencies and significant coarse-grain parallelism, while FPGAs can take advantage of data streaming and pipelining for more complex dependencies and fine-grain parallelism. Also, this work measures the development effort of the benchmarks in terms of Lines Of Code (LOC). The results

show that the implementations for FPGAs require between 1.3 and 2.8 times more LOC than their GPGPU counterparts.

Kapre and DeHon [KD09] analyze the performance of a SPICE simulation in FPGAs, GPGPUs, the Cell Broadband Engine (CBE) processor and multi-core processors. They report speedups of up to 132x for GPGPUs and 182x in the case of FPGAs. While the latter obtain a higher top speedup, their performance degrades with the problem size due to their limited resources.

Monte Carlo simulations obtain considerable speedup as well as energy efficiency running on FPGAs. Tian and Benkrid [TB10] report speedups over two orders of magnitude in a financial simulation compared to a general-purpose processor, while being 336 times more energy efficient. The same implementation obtains a 3x performance speedup against a GPGPU, also being 16 times more energy-efficient.

Kestur et al. [KDW10] undertake a performance study of the Basic Linear Algebra Subroutines (BLAS) on GPGPU and FPGA accelerators. BLAS is a library of algebra routines extensively used in the scientific community. The results show that FPGAs obtain performance comparable to GPGPU while being 2.7 to 293 times more energy efficient depending on the subroutine and input data.

Technology keeps improving every year. A more recent comparison between GPGPUs and FPGAs can be found in the white paper by the Berten company [Ber16]. The conclusions are still similar to the earlier papers, development effort and cost being the main drawbacks preventing wide adoption of FPGAs, and power efficiency being the main advantage over other types of accelerators.

The comparative works to date can be summarized in the following ideas. First, none of the two accelerators have a performance advantage for all types of applications. Second, FPGAs seem to obtain better results in power efficiency in all the surveyed works. Third, GPGPUs are easier to program. The last point is the main reason that FPGAs are still a niche market, in contrast to the market growth experienced by GPGPU accelerators.

Acceleration of unstructured-mesh codes with FPGAs has drawn little attention so far [AIO⁺12][AIO⁺11][SRSLB⁺11]. These datasets are widely used in scientific and industry simulations. In particular, many codes based on Partial Differential Equations (PDEs) use them. CFD solvers, such as Euler and Navier-Stokes, and earthquake simulations are some examples. This lack of previous research is due in no small part to the design complexity and the demanding memory requirements. In fact, even in the existing Central Processing Unit (CPU) codes, optimizing memory traffic for unstructured meshes is of utter importance due to their unpredictable data access patterns. FPGA accelerators are prone to show performance problems due to data access performance, as they do not have a default memory hierarchy to hide transfer latencies. Hence, solving or at least alleviating the memory bottleneck is one of the main motivations of this thesis.

1.1.2 Design automation

Much research work has been published in the world of embedded systems about the development of systems where some parts of the specification are implemented directly in hardware, and some others are implemented in software. What is commonly known as *hardware/software co-design* [MOS16][Sch10] consists in partitioning a specification into several hardware and software components, developing both in parallel and accounting for tradeoffs such as performance, power and development effort. Automating this process has been under heavy investigation in the last years [JVTB17][KZC16][YGR16][JNH16]. While embedded hardware platforms are often heterogeneous, they show notable differences when compared to HPC systems:

- The hardware platform in an embedded system is developed *ad-hoc*. In contrast, HPC hardware is general-purpose, and multiple jobs must typically co-exist in a system.
- The software for embedded systems is also fully *ad-hoc*. In contrast, the software in HPC systems is only partially developed for the application: at least an operating system is placed between the hardware and the application software, which gives runtime support for the application. A runtime system for HPC must be very lightweight in order to scale well up to the size of the machine.
- Embedded systems usually consist of a few computing cores (< 10), whereas current HPC systems are massively parallel ($> 10^4$ cores) and parallelism must be extracted accordingly.
- Because embedded systems are small, they allow implementing shared memory schemes more easily. Communication is efficient, and moving data between devices is simple by accessing the same memory locations. Massively-parallel HPC systems are based on distributed memory, with opposite results on efficiency and ease of use.

Despite the previous differences, the trend is to adapt some of the codesign principles to HPC. For example, GPGPU accelerators require partitioning the application in kernels and host code, which must be preceded by a careful analysis of the memory, communication and computing power tradeoffs between the GPGPU and the host processor. Some programming languages, such as CUDA or OpenCL, allow to specify such partitioning. They also let the programmer decide whether to optimize some details by hand or keep the compiler default behaviour, such as with mappings of variables to different levels of the memory hierarchy.

Traditionally, FPGA accelerators have been programmed in hardware description languages such as VHDL or Verilog. However, in the last decade, there has been an interest on C-like languages for hardware synthesis [WMH⁺15][Edw06]. These languages have specific extensions to allow hardware constructs that would be hard to specify in plain C. At the same time, they show a C-like syntax that is familiar to most of the programmers in order to improve the learning curve. But most important, because they are C syntax, integrating the FPGA and the host components is easier compared to the traditional approach, where CPUs were programmed in a high-level language and FPGAs were programmed in a hardware description language. Previous studies [EAMEG11][XSAH10] have analyzed the performance of various C-like languages against VHDL and Verilog, reporting that, although VHDL still beats the former from a performance perspective, the programming effort is several times higher.

As of today, codesign has not given a complete solution to the problem of distributed machines. Previous works dealt with performance estimation and simulation [SQJ11][SAA⁺08][ABDS02]. Distributed systems, such as HPC clusters or the Massively Parallel Processing (MPP) systems shown in the Top500 list, require a deep understanding of parallel architectures and communications from the application programmer. This adds up to the expertise required to use accelerators. Automating the process of application development on such systems is therefore the second main motivation of this thesis. Although implementation details may vary between different architectures used in the system, some problems can be dealt with from the system perspective, making them independent of the architectures used in the heterogeneous system.

1.2 Challenges

The previous motivational topics are too broad and complex to be addressed at once. This section introduces four specific problems that will be analysed throughout this work.

1.2.1 Memory bandwidth limitations

One frequent observation in accelerated HPC codes is that the movement of data between the accelerator and its memory becomes the bottleneck of the system well before the accelerator is used at maximum speed [NFMM13][KSY10]. This effect is caused by the memory wall [WM95], which states that the faster evolution of processor speed compared to memory makes computing power to be effectively bound by data access. The problem is worsened by the iterative nature of many scientific simulations, where datasets are transferred back and forth

between processor and memory multiple times. The memory wall has been hitting the performance of General-Purpose Processors (GPPs) for some time now.

Because accelerators achieve faster processing speeds than GPPs, they would be more affected by the memory wall should equal memory systems be used. For that reason, modern GPGPUs have dedicated memories arranged in custom hierarchies. For comparison purposes, a modern Intel Core i7 processor supports a maximum bandwidth of up to 50 GB/s, while the Nvidia Kepler K80 obtains 480 GB/s, which is almost an order of magnitude faster. This is not the case for FPGA accelerators, where the trend is to use technologies similar to the general-purpose processors or even a step behind. In addition, some accelerator boards implement all or part of the memory controller as a soft-core inside the FPGA in order to reduce cost. This in turn leads to slower memories, due to the inherent limitations in clock frequency. Furthermore, the programmer must manually implement part of the memory hierarchy by hand in order to take advantage of the on-chip memory. Reducing the traffic between the bandwidth-limited off-chip memory and the FPGA is central to the thesis.

On the other side, FPGA accelerators have on-chip memory resources. These are distributed over the whole area of the chip in multiple blocks, and each of these blocks have separate input/output interfaces. The theoretical bandwidth surpasses that of GPGPUs by an order of magnitude. However, this is not achieved in practice, and giving an exact bandwidth figure is not possible, since the application design affects the frequency of the buses, and the designer chooses the bus widths as well. In any case, an appropriate memory hierarchy that successfully integrates the big and bandwidth-limited off-chip memory with the fast and size-limited internal memories is key to obtaining speedup in FPGA-accelerated codes. Chapter 2 will use a custom *streaming* approach where a preprocessing stage sorts the mesh by analyzing the uses of each vertex and edge in the code, and serializes the result so that it can be transferred as a stream at run time.

1.2.2 Irregular data accesses

Maximum FPGA acceleration requires creating custom memory hierarchies that adapt to the algorithm and the data structures. General memory hierarchies and generators have been proposed for FPGAs [CHM11] [KPN⁺09] [DVCS08] [NS05], but they still rely upon spatial and time locality or at least predictable access patterns. Unstructured meshes generate unpredictable accesses to the memory because these are defined by the graph connectivity, this is, the connections between vertices as defined by the edges. The application only knows the structure of the dataset after reading it at runtime. Consequently, the aforementioned general and automated solutions are not likely to obtain good performance figures.

Improving spatial locality and predicting memory access patterns will be sur-

veyed. As the exact memory accesses are hard to predict, we will give methods that at least bound these accesses to specific regions of the mesh. These regions should be small enough to be able to store them temporarily inside the FPGA. This is tightly coupled to the task of reducing data transfers between off-chip and on-chip memory, and a wise allocation and layout of the on-chip memory.

1.2.3 Code transformations for distributed systems

Porting a piece of code to a heterogeneous distributed architecture involves mapping parts of the code to components of the hardware system with computational power. We refer to these hardware components as processors. Examples of processors are a general-purpose processor, a graphics coprocessor or a FPGA. Each processor is expected to perform differently, and different parts of the code are supposed to be mapped to the processor where the performance will be maximal. In this mapping, not only the computational power must be taken into account, but also the communications generated by splitting up the code. In heterogeneous systems, the communication network is usually heterogeneous as well; this needs to be taken into account in the mapping.

Distributed applications consist of many processes that execute on different processors and memory spaces. Once the mapping is decided, a transformation stage must rewrite the original sequential code, so that the behaviour of the distributed code is equivalent to the original. In the case of heterogeneous distributed systems, these transformations move segments of code to different functions or even new executables that will run in different processors.

Typically, the transformation of the code is done manually. CUDA [SK10] and OpenCL [GHK⁺12] are programming frameworks for heterogeneous computing that make this process easier; for instance, moving parts of the code to the accelerator in OpenCL is done by marking functions with the `kernel` keyword. Another relevant example of code transformation mechanism in distributed systems is the Remote Procedure Call (RPC)[BN84], which has been extensively used in client-server applications for the past thirty years.

Cardoso et al. [CDW10] enumerates several code transformations in the context of reconfigurable computing (such as FPGAs). These are classified in several categories according to the level of granularity: bit-level transformations, instruction-level, loop-level and so on. All the previous works on the subject require manual development to different extents.

In this thesis, we firstly propose an extension to the compiler's internal representation and an algorithm that allows the compiler to partition the code and automatically distribute control flow. On top of it, code partitioning strategies can be implemented. Secondly, we modify an existing parallelizing compiler that distributes data among processors. With our modifications and with the help

of our custom communications library, the compiler is also able to partition sequences of loops into multiple processors. Consequently, the compiler is able to distribute the workload in two dimensions: functional and data.

1.2.4 Automated insertion of communications

Communications between partitions are really a consequence of the previous challenge. Apart from taking advantage of the heterogeneous nature of the system, speedups are often obtained by splitting the problem into multiple processors, thus making use of the parallelism available in the system. In both cases, it is preceptive to communicate processors at certain points of the program, either to exchange data generated and consumed in different parts of the system, or to synchronize the execution between processors. The behaviour of the distributed application must be obviously kept identical to the original.

Developers have been using the Message Passing Interface (MPI) [The12], OpenMP [CJdP08] and similar communication standards for a long time. A comprehensive classification of parallel programming languages can be found in the survey by Diaz et al. [DMnCN12]. Each of them has tradeoffs between scalability, programmability and performance. For example, OpenMP has a simpler programming interface than MPI. However, it is only valid for shared memory systems, which usually scale up to some tens of processors at most. MPI can be scaled up to thousands of processors but results in cumbersome code and requires a considerable level of expertise even for parallelization of simple code.

Automating the parallelization in distributed systems without affecting performance is still an unresolved problem. Past efforts resulted in parallel programming languages such as OpenCL [SGS10], Julia [BCK⁺14] or Sequoia [FHK⁺06], and automated compiler frameworks such as HArtes [BSY⁺10], Polly [GGL12] or Graphite [PCB⁺06]. At the same time, heterogeneous systems came into play with the popularization of GPGPUs. Still, the problem of distributed memory is hard to handle automatically, due to the difficulty of mapping data flow to multiple address spaces. Partitioned-Global Address Space (PGAS) languages such as Co-Array Fortran [NR98] or Unified Parallel C (UPC) [EGS06] partially solve the problem by making address spaces visible within the programming API, but still require a considerable understanding of the memory structure from developers.

The last part of this thesis will propose a technique to split sequences of loops that iterate over unstructured meshes into different processors in a pipeline fashion. Sequences of loops are a common pattern in scientific simulations such as PDEs solvers, where an outer loop iterates over time and a sequence of inner loops iterate over the mesh. The proposed technique is based on the inspector-executor paradigm [SCMB90] and combines static compiler analysis and a runtime communication library to calculate the communications required between processors.

1.3 Structure of the thesis

After this introduction to the subject, Chapter 2, *Memory optimization for unstructured meshes*, introduces several optimization techniques to optimize the data transfer between processors in a distributed system. First, it presents a preprocessing algorithm to sort the meshes so that the access patterns are more efficient and suitable for streaming. Second, it defines a complete methodology to implement streaming mechanisms between processors. After an initial test with shared-memory processors, these techniques are applied to a specially complex and resource-demanding case where one of the processors is an FPGA. This chapter tackles the challenges of irregular data accesses and memory bandwidth limitations.

Chapter 3, *Refactoring of sequential codes for distributed systems*, proposes a compiler toolchain for automatic distribution of sequential codes. As this is a very broad subject; this work focuses on the low-level mechanisms necessary for the compiler to efficiently support transformations for distributed applications, not on the actual decisions about which parts of the code or data are mapped to each processor. First, an algorithm for automatic function outlining is proposed; this falls within the challenge of automatic code transformations. Subsequently, a communication mechanism is introduced that allows the compiler to represent distributed control transfer very similarly to how it represents the sequential one. This chapter tackles the challenges of code transformations and automated communications.

Chapter 4, *Automated pipelining with the inspector-executor paradigm* focuses again on the specific family of algorithms that use irregular data structures, such as unstructured meshes. It builds upon an existing parallelizing compiler and adds the possibility of pipelining sequences of multiple loops across several processors. In this way, the compiler is capable of parallelising two dimensions of the problem: the data structures, this is, the mesh, and the software specification, this is, the loops. The compiler takes advantage of the benefits of the well-known inspector-executor paradigm to process the unstructured datasets prior to the execution of the main algorithm. This chapter mainly addresses the challenge of automated communications.

1.4 Contributions

This thesis advances the state of the art with the following contributions:

- A breadth-first method to improve reference locality for general two- and three-dimensional unstructured meshes in scientific simulations. This method

improves the position of vertices in a streamed mesh so that the data dependencies are placed closely in the stream, thus allowing for fast eviction of unneeded data and reducing the amount of memory required inside the FPGA. Evidence proves that this technique is also beneficial for general-purpose processors and, potentially, to a broad set of architectures.

- A methodology to design high-performance mesh formats that accounts for the hardware design of the encoders and decoders, aimed at hardware accelerators based on FPGAs. This methodology allows calculating the requirements of the hardware components to achieve maximum speed.
- An algorithm to distribute arbitrary sections of code into several processors in a parallel architecture, where the communications for control transfer can be inserted automatically by means of a compiler.
- A communication library, based in the concept of RPC, that allows passing control between sections of code.
- A communication runtime library that lets a compiler split sequences of loops into several processors in a pipelined fashion, which can be easily applied by the compiler based on the inspector-executor paradigm.

The findings of this thesis have been presented in several publications in international conferences and journals:

P. Barrio, C. Carreras, J. López, O. Robles, R. Jevtic, and R. Sierra. Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes. *Journal of Systems Architecture*, 60(7):579–591, August 2014

P. Barrio, C. Carreras, R. Sierra, T. Kenter, and C. Plessl. Turning control flow graphs into function calls: Code generation for heterogeneous architectures. In *2012 Int’l Conf. on High Performance Computing and Simulation (HPCS)*, pages 559–565, July 2012

P. Barrio and C. Carreras. Turning Control Flow Graphs into Function Call Graphs: Transformation of Partitioned Codes for Execution in Heterogeneous Architectures. In *2012 European Conference on the LLVM Compiler Infrastructure*, <http://llvm.org/devmtg/2012-04-12>, 2012

P. Barrio and C. Carreras. Mesh traversal and sorting for efficient memory usage in scientific codes. In *Proc. of the 30th IEEE Int’l Performance Computing and Communications Conf. (IPCCC)*, Nov. 2011

Also, part of the contents of Chapter 2 resulted in a U.S. patent granted:

P. Barrio, C. Carreras, R. Sierra, J. A. López, G. Caffarena, E. Sedano, J.A. Fernández, and R. Jevtic. Systems and methods for improving the execution of computational algorithms. Patent number: US 9,311,433, Country: US, Owners: Airbus España and Universidad Politécnica de Madrid

These research activities have been undertaken in the context of the following national and international projects:

Fusim-E DOVRES: Design Optimization by Virtual REality Simulation. Main researcher: Carlos Carreras. Contractor: Airbus España S. L., Universidad Politécnica de Madrid, contract number A8213093G. January 2008 to December 2010.

AMURA: Sistemas UWB Multielemento y Reconfigurables para Aplicaciones Avanzadas de Señal - Diseño Microelectrónico. Main researcher: Carlos Carreras. Contractor: National Research and Development Plan, Ministerio de Ciencia e Innovación, contract number TEC2009-14219-C03-02. January 2010 to December 2012.

FastCFD: Desarrollo de técnicas avanzadas para aceleración de las simulaciones CFD para aplicaciones aeronáuticas. Main researcher: Juan Antonio López. Contractor: ETSI Telecomunicación. January 2010 to December 2010.

AMIGA 5. El Gas en las Galaxias y en su Entorno. Preparación Científica y Tecnológica para el SKA Procesado de Datos en Hardware. Main researchers: Carlos Carreras y Juan Antonio López. Contractor: Ministerio de Economía y Competitividad, Programa Estatal de I+D+i Orientada a los Retos de la Sociedad, project AYA2014-52013-C2-2-R. January 2015 to December 2016. (External collaborator)

AMIGA 6. El Gas en el Interior y en el Entorno de las Galaxias. Preparación Científica para SKA y Contribución al Diseño del Flujo de Datos. Procesado de Datos en Hardware. Main researchers: Carlos Carreras y Juan Antonio López. Contractor: Ministerio de Economía y Competitividad, Programa Estatal de I+D+i Orientada a los Retos de la Sociedad, project AYA2015-65973-C3-3-R. January 2016 to December 2018. (External collaborator)

Chapter 2

Memory optimization for unstructured meshes

Simulations based on mesh data structures have been identified as major drivers of High Performance Computing (HPC) [ADK⁺09]. The typical applications based on meshes are numerical simulations using finite element or finite volume methods. The mesh describes the state of a physical system, and the algorithm computes the evolution of the system over time by iteratively processing the mesh in a series of timesteps. Due to the high complexity of the requirements in terms of precision, algorithm convergence and dataset size, the computational demands of such simulations are among the highest in HPC.

Meshes may be either structured or unstructured, as shown in Figure 2.1. Unlike structured meshes, the impact of unstructured meshes on the memory system has been seldom studied in the context of high-performance numerical simulations. This chapter proposes a series of memory optimizations for unstructured mesh codes, suitable for both general-purpose processors and custom FPGA computational kernels. Background concepts are introduced in Section 2.1 together with relevant work that has been previously published on the subject. Section 2.2 proposes a general optimization to improve data locality by rearranging the order in which the vertices and edges of a mesh are used in the computations and stored in memory. This is followed, in Section 2.3, by a complete methodology to improve memory traffic in FPGA accelerator boards. This methodology reuses the previous optimization to improve the efficiency of data compression techniques, and allows efficient streaming access to the mesh stored off-chip from the FPGA. The chapter is closed by an analysis of the lessons learned, in Section 2.5, of the work and results achieved.

One of the main users of such unstructured mesh simulations is the CFD community. Therefore, the proposed techniques have been tested on meshes regularly used in the industry, such as the National Advisory Committee for Aeronautics

Figure 2.1: Structured and unstructured meshes. Only a) is unstructured. Courtesy of Adrian Lange, Wikipedia.

(NACA) airfoils (explained in [JWP33]) and datasets gathered from DLR Germany, ONERA France and Airbus Spain. The results are very promising: up to 63% of the L1 cache misses are removed in a general-purpose CPU just by sorting the mesh. We have obtained speedups of up to 2.58 on the memory access time. Additionally, applying the complete methodology reduces the memory traffic of CFD meshes down to an average 34% of the original traffic in a FPGA accelerator board.

2.1 Background and related work

2.1.1 Unstructured meshes

Graph algorithms are among the most memory-demanding in general computing. In the last years, major HPC facilities and data centers have started to measure their performance with benchmarks based on graphs. This has materialized in Graph 500 [gra16], a list of highest-performing systems similar to the classic Top500 list [top13]. The benchmark used in this case is the homonymous Graph 500 benchmark [MWBA10], which is composed of two kernels, one of which creates a general undirected graph with a parametrizable degree (i.e. number of edges per vertex), and the second of which runs a breadth-first search over the graph.

Meshes are based on the mathematical concept of simplicial complexes [Har94]. Unlike graphs $G(V, E)$, where V is the set of vertices and E is the set of edges, meshes also contain elements, sometimes called cells, which are the simplices of the simplicial complex. Some applications associate data to these cells. As an example, in graphics applications, the cells are surfaces; a “colour” property could be associated to the faces so that the rendering processor can print out the mesh with colours.

Assigning data to cells is also common in HPC applications. In many cases, when cell data are required, the data structures are made of two complementary

Figure 2.2: A two-dimensional mesh and its associated graphs. The dual graph has an extra edge for each cell that connects to the imaginary cell surrounding the mesh.

graphs: the primal and the dual graph, as shown in Figure 2.2. The dual graph represents the cells as vertices and the adjacency between cells as edges. In addition, it is often possible to rewrite the computational algorithm so that the cell properties are remapped to vertices and edges of the primal graph. For instance, a CFD code used throughout this research revealed that the dual graph was only used in a preprocessing stage to calculate geometry data associated to the primal graph. The computational kernel then used the primal graph only. Hence, the optimization techniques discussed in the following sections apply only to a single graph, this is, with no cells involved. From this point onwards and for simplicity, the terms “mesh” and “graph” are used indistinctly unless otherwise noted.

Two vertices connected by an edge are said to be neighbours. If the shortest path between two vertices of a mesh involves k edges, we refer to them as k -distance neighbours or, in short, k -neighbours¹. k -dependencies mean that

¹Since we refer to this concept extensively throughout the text, this is to avoid the traditional and longer form: “vertices at a distance k ”.

data from k -neighbours are required in the computations of each vertex. It is also common to encounter additional $k - 1$ down to 1-dependencies. A high k results in a dependency explosion that greatly limits data locality, thus hiding the effect of memory optimizations.

We classify the mesh data into two types. First, the connectivity (or topology) information defines these relationships between vertices and edges, which in turn define the shape of the mesh. Second, the algorithmic data represents all the information relevant to the particular mesh algorithm. Strictly speaking, if changing the value of a variable in the dataset results in an isomorphic graph (i.e. the graph does not change its structure), then the variable is part of the algorithmic data, and vice versa.

The connectivity in unstructured meshes is commonly described either as an adjacency list or as an adjacency matrix [CLR⁺01][SLL01]. In this work we use adjacency lists, but equivalent optimizations are possible for adjacency matrices. Hence, vertex and edge lists are used to keep the data required by the computational algorithm. Each vertex specifies its associated algorithmic data, a vertex identifier (ID) and also a variable number of identifiers that point to the edges that the vertex connects to. Similarly, each edge specifies its associated data, an edge ID and the two vertex IDs of its endpoints. The IDs of vertices and edges can be made implicit, for example, as their index in the list in case that these lists are implemented as arrays (e.g. the vertex in position 3 has ID = 3). Typically, a mesh-based scientific code processes the mesh iteratively, either by vertices or by edges. Processing in terms of edges only requires the edge list acting as the adjacency list by keeping the neighbouring vertex IDs. Similarly, processing in terms of vertices requires the vertex list acting as the adjacency list by keeping the IDs of all incident edges. For the sake of simplicity, we assume edge-based processing for the rest of the thesis, meaning that the processing order will be given by a sequence of edges. Obviously, vertices are still computed in the process. It is also possible to reformulate the optimizations for vertex-based processing.

Manifold meshes

A mesh in the three-dimensional space is a 2-manifold [Lee10] if it meets these two requirements:

1. Each edge is incident to only one or two faces.
2. The faces incident to a vertex form a closed or open fan.

The faces are the surfaces that delimit cells in three-dimensional meshes. Figure 2.3 gives some examples of manifold and non-manifold meshes. For the rest

Figure 2.3: Manifoldness. Mesh **a** is a manifold. Mesh **b** is not a manifold, since it does not meet requirement 2. Mesh **c** is not a manifold, since it does not meet requirement 1. Courtesy of Prof. Mario Botsch, U. Bielefeld.

of this work, we simply refer to 2-manifolds as just manifolds. Mathematically speaking, other manifolds are possible in higher-dimensional spaces, but they are rare in scientific computing applications.

One of the contributions of the proposed optimizations is that they are valid for non-manifold meshes. Previous optimizations of unstructured meshes were targeted at graphics processing, where the meshes are surfaces. In practice, a meshed surface results in a manifold mesh, with a very small chance that non-manifold edges and faces appear. In contrast, scientific datasets often model solid bodies or spaces, which result in a high percentage (over 95% in our test meshes) of the edges being non-manifold.

2.1.2 The memory wall

Working with large datasets is a common requirement in HPC codes. Since the stress put at the memory increases with the size of the dataset, memory traffic is a major concern when porting codes to HPC systems. Modern CPUs include a complex hierarchy with many levels of caches, which effectively hides the latency of the upper memory levels and prevents processor stalls to some extent.

However, although certainly making life easier to the programmer, cache mechanisms are not an efficient solution for every code, but a fairly good one that works for all. Caches do not account for access sequences specific to each code, but expect a considerable amount of time and space locality in order to work efficiently. In addition, some novel devices used as accelerators, such as FPGAs, are typically best suited to sequential memory access rather than cache-like random access. Unfortunately, the vast majority of existing codes require random

access, as they are originally developed for memory systems with caches.

Cope et al. [CCLH10] analyze various architectural issues that impact performance on GPU and FPGA accelerators. The study concludes that, while FPGA accelerators are better suited to massive and regular memory access, GPUs perform better with variable data reuse. However, while the bandwidth available at the FPGA internal block RAMs is huge, FPGAs are typically embedded into a PCIe board with on-board memories. Their controllers are often implemented as soft cores that do not achieve a comparable clock speed to its CPU and GPU counterparts, and thus memory optimizations are more critical.

Structured- and unstructured-mesh codes have similar patterns of computation and communication [ADK⁺09]. In the past, many techniques have been developed in order to optimize the memory accesses for these datasets, such as different loop and data transformations [LS04][RT98][MCT96]. However, unstructured meshes are more challenging. While the regular connectivity of structured meshes easily allows locality optimizations, unstructured meshes do not allow assumptions about the mesh connectivity. These result in unpredictable access patterns that do not map well to the linear memory model. Automated generation of optimized meshes for complex interdisciplinary problems is still an active field of research [Shi06][DSS07][KdA07][GR09a]. The mesh generation process is currently guided by the requirements of the computational algorithm in terms of convergence and precision rather than computational efficiency, and therefore there is no general solution to automatically increase locality.

Interestingly, meshes result in sparse graphs. Each vertex is connected to a small amount of vertices compared to the total number of vertices. The number of neighbours of a vertex is known as its degree². In our 2D test meshes, the maximum degree oscillates between 7 and 11. Our 3D meshes occasionally reach 20, but degrees above that number are rare. This enables runtime memory optimizations based on spatial data locality since, although the exact memory operations within a vertex computation are unknown, their number is bound to a small amount.

There is a growing interest in solving the memory bottleneck in FPGA accelerators with mesh codes. Very recently, Sano et al. [SHY14] proposed a streaming memory approach together with a massively pipelined FPGA implementation in order to optimize general stencil computations. The approach is similar to that proposed in Section 2.3. However, the former proposal has two major drawbacks with respect to ours: first, it only targets structured meshes; second, the massively parallel implementation inside a single FPGA is only feasible for small

²Strictly speaking, the degree of a vertex is the number of edges incident to the vertex, with loops counted twice. However, typical simulation meshes allow two simplifications. First, there are no loops, and therefore no edge is counted twice. Second, the graph is undirected, hence there is no distinction between in-degree and out-degree.

computational kernels due to resource limitations. Our memory optimization has proved to work with an industrial CFD kernel of considerable size. This code uses the computational kernels from the DLR-TAU code [GGF⁺], a fluid dynamics simulation created by the German Aerospace Centre (DLR, for its german acronym) and widely used in the industry.

2.1.3 Improving data locality

The data dependencies of a vertex are determined by that vertex and its neighbours. To a lesser extent, they may also access data from the neighbours of these first-level neighbours and consecutive neighbourhood levels, which we called k -dependencies. Assuming $k = 1$ and in a classical unoptimized access scheme, when a vertex V_o is processed, its data will be loaded to the computing element together with all of its neighbours V_1 to V_N . Every time that a vertex V_i , $i \in [1, N]$ is computed, V_o will be requested to the memory, since it belongs to the neighbourhood of V_i . Hence, the scheme forces the transfer of each vertex as many times as its number of neighbours plus one. Because the maximum degree of a mesh is small due to its sparse connectivity, it is possible to improve spatial locality by placing neighbours close to each other in memory. In caches, this has the effect of saving vertex transfers because some of the neighbours are transferred together in the same cache line. In other memory systems, spatial locality is also an advantage, as will be shown in Section 2.3.

A common approach for triangular meshes in graphics applications is to divide them into *triangle strips* [ESV96] that determine a particular triangle sequence, which is then sequentially fetched into the graphics pipeline. Strip processing greatly reduces bus traffic to the GPU compared to a random triangle-by-triangle approach, since the vertices are reused inside the GPU. Virtually all rendering APIs, such as OpenGL, as well as graphics hardware, such as Nvidia GPUs, have native support for triangle strips. The process of generating a sequence of vertices or edges that represent the mesh is typically referred to as mesh sorting.

Mitra and Chiueh [MC98] propose an efficient mesh representation based on a breadth-first traversal. With the same motivation as the triangle strips, they exploit a different traversal method in order to maximize the usage of each vertex in several triangles. This is represented in Figure 2.4. The breadth-first approach generates strips, although these wrap around the initial cell. Hence, the vertices belong to more than two triangles inside the strip, allowing a wider use of the vertices in the future. This improvement comes at the cost of a larger buffer capable of storing more reusable vertices. A hardware implementation is proposed in [MC02].

Derived from research on graphics, Isenburg et al. [ILGS03] generalize mesh sorting to a wider set of computing applications with their *processing sequences*.

Figure 2.4: Triangle strips. The first is the original depth-first traversal; the second is the breadth-first traversal as used by Mitra and Chiueh.

Their goal is to force the algorithm to compute the dataset in a particular sequence so that a small portion is stored in the lower levels of the memory, similarly to the techniques that will be proposed in Section 2.3. In [IL05], the approach is applied to out-of-core mesh simplification in a CPU.

Sorting the vertices and edges of a mesh can be done by traversing the graph at a preprocessing stage. This creates a spanning tree and generates a vertex and edge sequence in the visiting order. Graph traversal algorithms are intensively used in other computing disciplines such as artificial intelligence. Korf [Kor95] analyzes the behaviour of several graph traversal algorithms with respect to memory. Zhou and Hansen [ZH06] observe the growing size and shape of the *frontier*, which is the part of the graph with vertices that have been visited by the traversal algorithm, but have not been closed. The study concludes that breadth-first search (BFS) is more memory-aware than other popular graph search algorithms such as A* search. Since the vertices and edges are computed in the order defined by the sorting process, the memory access patterns are identical at the traversal stage and the computational kernel. Therefore, a memory-efficient traversal algorithm also optimizes memory access at compute time.

Vertex windows

In the context of a mesh sorted into a sequence of vertices or edges, we define the term “window” as the portion of the sequence between the first use of the vertex and its last use. With 1-dependencies, dependent vertices are only the neighbours. Note that, by definition, it is guaranteed that no vertex will be used out of its window. We use the concept of window to evaluate spatial locality. In Sections 2.2 and 2.3, we use window minimization techniques to reduce memory traffic in both caches and a custom FPGA memory hierarchy.

Some of the abovementioned works use concepts equivalent to the vertex windows. Sano et al. [SHY14] use a buffer to keep vertices that will be used in the

future inside the FPGA. Since the target is a structured mesh for stencil computations, the buffer size is constant and of size $2y$ plus the active vertex. Here, y is the width or height of the structured mesh, depending on the direction of the traversal. On the contrary, unstructured meshes lead to windows of various sizes across vertices.

In the case of Mitra et al. [MC98], a breadth-first traversal iteratively grows a frontier and separates it into three components (previous, current and next), which behave similarly to a propagating wave. When a certain vertex is being executed, the state of these three sub-frontiers represent the window as defined above. However, they define a concept called active windows that allows further partitioning of the frontiers in smaller parts that can be prefetched from memory on demand. Hence, our definition of window is more similar to their definition of frontiers rather than that of active windows.

When the graph is represented as an adjacency matrix instead of an adjacency list, window optimization is similar to bandwidth reduction. This is illustrated in Figure 2.5. This problem is NP-complete [Fei00], and therefore only solutions based on heuristics exist. Nagy et al. [NNH⁺12] use the Gibbs-Pole-Stockmeyer algorithm for that purpose, resulting in potential reductions. However, they do not report bandwidth results for the memory accesses but only the speedups of a FPGA-accelerated Euler solver on a synthetic sample mesh. Breadth-first sorting is similar to the Cuthill-McKee algorithm, as in Figure 2.5b. A discussion of matrix bandwidth reduction algorithms can be found in [Eve79].

There is a key difference between bandwidth reduction and window optimization. In the example of Figure 2.5b, the traversal algorithm is assumed to iterate through the mesh in terms of vertices as described by the sequence in the X or Y axis. For instance, vertex 1 is required by vertex 7 for the first time, therefore it is not needed in memory before processing vertex 7, and it is required for the last time by vertex 12, therefore it can be evicted from memory after processing vertex 12. However, this assumes that the algorithm iterates over the graph by vertices. If the algorithm iterates through the edges instead, the window for each vertex grows.

To illustrate the problem, Figure 2.6 introduces an adjacency matrix where the vertices after vertex 8 have dependencies only with nodes placed earlier in the sequence. Algorithms based on bandwidth reduction compute the bandwidth of vertex 9 as 8 (vertices 1 to 9), which is the number of vertices that should be loaded in memory at that time. Here we are assuming that we cannot discard vertices between the start and end of the window of vertex 9. Unfortunately, because the processing is in terms of edges, later vertices are needed earlier on because of their edges. For example, vertex 16 and 8 have an edge in common and, because 8 is processed before 9, 16 will be already loaded when vertex 9 is processed. Therefore, vertex 16 is part of the window of vertex 9. Because of

(a) Sorting according to vertex ID.

(b) Cuthill-McKee sorting.

(c) Original mesh with superposed windows.

Figure 2.5: Sorting the mesh appropriately results in smaller windows. Figures 2.5a and 2.5b show the adjacency matrices. Figure 2.5c shows the subsets of vertices that belong to two windows: vertex 8 with a sequence based on vertex identifiers, and vertex 6 with a Cuthill-McKee sorting.

this, and considering other vertices required by those preceding 9, the effective window of vertex 9 is now 16 vertices, from 1 to 16. The parts of the adjacency matrix in red show how the original bandwidth calculations span to the end of the sequence because of this edge between 8 and 16.

It is still possible to calculate the window using the adjacency matrix for edge-based processing: the original window calculation for a vertex V expands to the

Figure 2.6: Difference between bandwidth and window due to edge processing

right (later in the sequence) to the latest vertex L that is needed by a previous vertex P ; in the previous case, $V = 9$, $P = 8$ and $L = 16$. Similarly, the window will expand to the left (earlier in the sequence) until the earliest vertex E that is needed in a later vertex T . The example has no cases of this situation, as can be seen in the bottom-left part of the adjacency matrix. Correctly calculating the window sizes will become more important when streaming data transfers are introduced in Section 2.3.

2.1.4 Reducing data size

Gotsman et al. [GGK01] group several mesh compression techniques into three types: mesh decimation, connectivity coding, and geometric data compression. Decimation changes the resolution of the mesh, hence it is a lossy compression technique and it requires a deep understanding of the numerical stability and precision of the algorithms. On the contrary, coding can be easily applied without affecting the original results, and lossless data compression techniques also exist.

Section 2.3 takes advantage of connectivity coding and data compression to reduce bandwidth use. In the context of this work, encoding refers to the process of packing the different components of a mesh description in a common frame where all the connectivity redundancies are removed. Data compression refers to representing data values, either algorithmic or related to connectivity, using less bits than the original data.

Connectivity coding

Mesh coding has been widely studied in the field of graphics processing and visualization. However, scientific datasets have little resemblance with their graphics counterparts. Graphics techniques usually require either triangle or tetrahedral meshes to achieve good compression rates. Scientific codes often have a variety of polygons and polyhedra as cells in order to guarantee convergence of the algorithms. Also, quasi-manifold meshes are assumed in the graphics datasets, since they typically represent surfaces. Scientific datasets model solid bodies or spaces, which result in non-manifold meshes. Mitra and Chiueh [MC98] design an encoding for triangle meshes, which are extensively used in computer graphics. Similarly, Isenburg et al. [IG03] encode the mesh with a sequence of commands, followed by Huffman and run-length compressions. The command sequence relies on the edges being manifold, and a more expensive encoding mechanism is provided for non-manifold cases. While they report non-manifoldness of 4% in their worst-case dataset, our 3D CFD meshes show a non-manifoldness close to 100%, making such approaches unfeasible.

The Edgebreaker scheme proposed by Rossignac [Ros99] uses a sequence of commands to encode the mesh iteratively from a starting cell. However, the initial approach is only valid for planar triangulated graphs with no boundaries, although an extension is proposed for boundary encoding that requires extra storage. Such graphs are popular in graphics applications, but rarely used in three-dimensional scientific simulations. More recently, Gurung et al. [GR09b] proposed a compact representation scheme for tetrahedral meshes. Again, the encoding is not independent of the cell types but tied to tetrahedrals.

Lossless data compression

As opposed to connectivity coding, lossless data compression has been widely studied in the field of numerical simulations. Although general-purpose compression algorithms can be applied to scientific data (e.g. GZIP, BZIP2, etc.), Engelson et al. [EFF00] state that specific properties can be used to design data-specific compression schemes that result in better compression ratios. By taking advantage of the smooth changes of physical variables in time and space, they compress sequences of floating-point numbers by converting them to integers and computing their delta values (n -th order differences).

Katahira et al. [KSY10] further evolve floating-point compression to predict future values with polynomial functions, resulting in better compression ratios. A hardware implementation over an FPGA device is also presented and evaluated. The compression scheme used in this work, explained in Section 2.3.2, results in a slightly worse compression ratio. In return, it only uses a small fraction of the

FPGA resources required by the abovementioned implementation.

2.1.5 Memory systems for FPGA accelerators

The memory in a FPGA accelerator is either inside the FPGA (on-chip) or outside (off-chip, on-board). The application programmer is free to arrange it in a wide variety of memory architectures. This freedom allows deep optimizations of the memory to the data structures, but comes at the cost of increased development complexity. Previous works on the optimization of hardware accelerator memories mainly deal with the automated mapping of regular data structures (i.e. arrays) into the FPGA memories or the generation of cache-based memory hierarchies for irregular data structures.

Gokhale and Stone [GS99] develop an algorithm to map arrays into on-chip and off-chip memories in a FPGA accelerator board. The aim is to minimize the execution time of certain loops. This algorithm is integrated into a compiler so that the mapping process is automated. However, each array must be entirely mapped to a single memory, since partitioning is not allowed.

Nalabalapu and Sass [NS05] generate application-specific caches by analyzing the memory access patterns in loops. The induction variables are recognized and used to determine the data reuse patterns, which in turn are used to address a data cache with a special address generator core, so that redundant memory accesses are saved. Similarly, Devos et al. [DVCS08] analyze the induction variables of loops to prefetch parts of arrays so that they do not generate a cache miss. Because these approaches require an analysis of the induction variables of loops, they are not appropriate for unstructured meshes, since the memory access patterns are not given by the loop index but by the contents of the memory itself.

Adler et al. [AFP⁺11] propose a higher-level abstraction called LEAP scratchpads, where LEAP stands for Logic-based Environment for Application Programming. By instantiating several scratchpads, the FPGA programmer can instantiate multiple memories for different purposes without taking care of the different memory levels (i.e. on-chip, off-chip). The scratchpads use traditional cache techniques such as block transfers and associativity, and they use the internal memory blocks and the external RAM under the hood.

Other works handle software issues related to memory performance, such as pointers. Park and Diniz [PD11] use the FPGA as a sophisticated accelerator for memory operations. By analyzing the memory access patterns at runtime, they are capable of anticipating the necessary data accesses and reorder the data structures so that the traversal over the data is more efficient. Unlike the previous works, they are capable of handling access to pointer-based data structures rather than stride-based access only. Chung et al. [CHM11] expose the FPGA memory organization to the programmer by means of a higher-level programming

Figure 2.7: Three sorting methods applied to a mesh. For each method, the upper line represents the sequence generated by the sorting algorithm, whereas the lower line represents the state of the intermediate buffer during the computation.

API called CoRAM. This abstraction is easier to use than the original memory interface, but the memory organization must still be designed by hand.

2.2 Locality optimization with mesh sorting

This Section presents an optimization to spatial locality based on sorting the dataset according to a graph traversal algorithm. A thorough explanation of graph traversal algorithms can be found in the book by Gross and Yellen [GY06].

We introduce three sorting methods: random, depth-first sort (DFS) and breadth-first sort (BFS). Figure 2.7 depicts the initial processing sequence and the state of an intermediate buffer for these algorithms. The buffer has a capacity of 15 vertices. The algorithm defines 1-dependencies, and all sequences start from vertex A.

The computation of A triggers the transfer of its neighbours B, C and D

from the main memory to the buffer. In the random approach, the sequence may continue with any vertex. Let us assume that the visiting order starts with the sequence A-L-J-Q. When L is selected, G, H, M and P are also transferred. Subsequently, J is computed and E, F, I, K and O are written to the buffer. At this point, the buffer is full and the next vertex, Q, cannot be computed unless some vertices are discarded in order to bring its absent neighbour, N, and itself. Only three vertices (A, L and J) are computed up to this point.

We begin the sequence of the DFS approach with A-D-F-J-O-N-M, one of many valid DFS sequences. A generates the same transfers as the random approach, but the second vertex, D, takes advantage of the space locality and only transfers three vertices, since some of its neighbours (C and D) have been already transferred with the previous vertex. At the end, with the same buffer size, 20 vertex transfers are saved. Before vertex M forces to discard vertices in the buffer, six vertices are processed. This sequence is the vertex-centered equivalent to the triangle strips introduced in Figure 2.4.

In the case of BFS, we observe 11 computed vertices and 46 saved transfers, requiring only 25% of the memory transfers generated by the random sequence. Observe that the window of vertex C spans from vertex A (its first neighbour) to vertex G (its last neighbour), adding up to a window size of ten vertices. After computing G, vertex C can be removed from the intermediate buffer. Our breadth-first algorithm, BFsrt for convenience, traverses the mesh starting from a seed vertex and building a spanning tree level by level. The neighbours of the seed form the first level, the second-generation neighbours will be the second level, and so forth. We call these levels frontiers to use the same naming as earlier works on the subject. When a vertex is discovered, only two frontiers are open: the frontier that the vertex in process belongs to, and the next frontier with some of its neighbours. Any lower frontier is completely processed by the algorithm, and successive ones are not yet started. We take advantage of this property to reduce vertex windows as defined in Section 2.1.3.

The BFsrt algorithm is explained in Section 2.2.1. This is followed by a discussion of the window sizes in Section 2.2.2. Finally, the problem of seed selection is introduced in Section 2.2.3. Seed selection is the process of selecting the vertex from where to start the mesh traversal. To our knowledge, we are the first to study such problem, evaluating its impact on the overall quality of the resulting sequences, and proposing several algorithms with different tradeoffs between seed quality and execution time.

2.2.1 The BFsrt breadth-first algorithm

The pseudocode of BFsrt is given in Figure 2.8. The algorithm starts from a seed vertex, visiting all of its neighbours. Once the seed is finished, we take

2.2. Locality optimization with mesh sorting

```

1: for every vertex do {Initialize}
2:   stat(vertex)  $\leftarrow$  UNDISCOVERED
3: end for
4: seq  $\leftarrow$  0
5: seed  $\leftarrow$  select_seed()
6: queue  $\leftarrow$  insert(seed)
   {Loop over the mesh}
7: while queue  $\neq$  empty do
8:   central  $\leftarrow$  extract(queue)
9:   order(central)  $\leftarrow$  seq
10:  seq  $\leftarrow$  seq + 1
   {Insert neighbors in the queue}
11: while at least one neighbor is not visited do
12:   neighbor  $\leftarrow$  select_neighbor(central)
13:   if stat(neighbor) = UNDISCOVERED then
14:     queue  $\leftarrow$  insert(neighbor)
15:     stat(neighbor)  $\leftarrow$  VISITED
16:   end if
17: end while
18:  stat(central)  $\leftarrow$  FINISHED
19: end while

```

Figure 2.8: Customizable BFSort

the first processed neighbour and look for neighbouring vertices that are not yet added. These are the second-level neighbours of the seed. The algorithm continues visiting and discovering vertices until the complete graph is traversed. Two functions serve us as customization points: *select_seed()* in line 5 and *select_neighbor()* in line 12. Seed selection is covered in Section 2.2.3. Neighbour selection, also called tie resolution, occurs when many vertices can be selected as the next in the sequence. Two tie resolution strategies have been tested in this work: random and clockwise. Figure 2.9 shows how applying these strategies affects the resulting sequences. We have found that different tie resolution strategies do not have an effect on vertex window sizes, but some of them improve differential compression schemes [GGK01] suitable for particular target applications. These results are introduced in Section 2.4.2.

Figure 2.10 shows the evolution of the cell-based traversal, as proposed by Mitra et al. [MC98][MC02], and the vertex-based breadth-first traversal as we propose. In short, the cell-based scheme takes an initial triangle and adds its vertices to the sequence. Here we follow a clockwise vertex selection to choose the vertex priority inside a triangle. Subsequently, further neighbour triangles are chosen and continue filling the sequence with new vertices. Observe that a FIFO queue is used to track vertices that have been discovered but not completely processed. The vertices that reside in the queue reflect the window of the vertex

Figure 2.9: Random and clockwise tie resolution strategies.

in process.

Similarly, the vertex-based scheme proceeds as follows. At each step, we define the *central* vertex as the one that triggers unvisited neighbours to be added to the sequence. We also define the *seed* as the first *central vertex* in the traversal, which is also the first vertex to be added to the sequence. This example arbitrarily takes vertex 0 as the seed. Starting from vertex 0, all of its neighbours are visited next and added to the sequence. For clarity, clockwise tie resolution is used. Next, a neighbour is selected (for example, vertex 6) and proceeds as the new central. This process iterates until every vertex is visited, resulting in the sequence 0-6-1-2-3-4-5-17-18-7-8-10-11-12-13-14-15-16-19.

The states of the vertices in the BFsort preprocessing stage have a meaning in the subsequent computational stage. At any given moment, an undiscovered vertex in BFsort translates into a vertex that has not been requested by the kernel. A visited vertex implies that its window is open, and therefore must be loaded into the lower memory levels. When the vertex is finished, the vertex is not required in the computations. The cell-based approach is used by Mitra et al. to improve vertex reuse inside graphics pipelines. Nonetheless, we propose the second as being more efficient for scientific codes. The point where vertices are finished is reached earlier in a vertex-based traversal.

If the mesh is transferred sequentially, as in stream access, a random order will face the storage of the complete mesh in the worst case. This is, if the first and last vertices are neighbours, they will be brought together to the low-level memory, and the first will not be eligible for deletion until the end of the traversal. This is because the last vertex belongs to the window of the first vertex. With BFsort, at any point, the open frontier is a small percentage of the mesh.

2.2. Locality optimization with mesh sorting

Figure 2.10: Breadth-first evolution by cells (upper line) as proposed by Mitra, and by vertices (lower line) as proposed in this work, together with the FIFO queues used to build the order. The lightly coloured vertices and cells are visited but not closed. The dark-coloured vertices are finished.

2.2.2 Window size results

The techniques proposed in this Chapter have been applied to a set of benchmark meshes used in Computational Fluid Dynamics (CFD). Table 2.1 shows relevant features of these datasets; Figures 2.11 and 2.12 show general views and a zoomed view of the modeled object. We pay special attention to the edge to vertex ratio (δ), which represents how dense the connectivity is, and the percentage of connectivity data against algorithmic data (% connect.). δ determines how effective the locality optimization will be, since a high density of the connectivity results in bigger vertex windows. The percentage of connectivity with respect to algorithmic data is important for the methodology described in Section 2.3, since the techniques applied to both types of data obtain different compression ratios.

Most of these meshes are standard NACA airfoils publicly available. Only the 2D “turbulent” and “fine”, and the 3D “wing” and “plane” have been gathered from Airbus Spain, ONERA France and DLR Germany and are not in the public domain, hence they are not shown in the figures. We consider this group of

	Vertices	Edges	δ	%connect.	Preprocess (s)
2D					
NACA 2412	3070	8869	2.89	31%	0.034
NACA inviscid	5233	15449	2.95	31%	0.059
NACA 0012	7908	23296	2.95	31%	0.087
NACA NS	25651	57429	2.24	28%	0.240
NACA laminar	31204	92706	2.97	31%	0.379
turbulent	54165	138791	2.56	30%	0.693
fine	494128	1478960	2.99	32%	5.898
3D					
NACA	83888	292410	3.49	23%	2.360
wing	108396	710525	6.55	31%	5.931
plane	275561	1785732	6.48	31%	10.301

Table 2.1: Characteristics of the datasets

meshes to be general enough to extrapolate the results to the meshes used in other scientific codes, due to their varying cell shapes (triangles, rectangles, tetrahedra, etc.), resolutions and manifoldness along the mesh.

Figure 2.13 shows the maximum window results for three sequences (original, cell-based breadth-first and vertex-based breadth-first) and five two-dimensional test meshes. Three-dimensional meshes are not analyzed here, since the available cell-based sorting is currently valid only for surface cells. Three-dimensional cell-based sorting is possible, but has been discarded due to the better results obtained with vertex-based sorting in two-dimensional meshes.

The window size and the mesh size are not necessarily proportional, which in turn makes difficult to set upper bounds for the sizes of the input meshes. Breadth-first sorting achieves smaller frontier sizes than the original order, reducing the window at least by a factor of 5 and reaching in some cases a factor of 20. Additionally, the vertex-based approach obtains smaller windows than the cell-based one, as Figure 2.10 predicted. Both vertex- and cell-based algorithms work well compared to the original sequence; however, the first one improves the results of the second by 1% to 7% of the mesh size. Hence, if the computational kernel does not make use of cell-centered data, a sequence based on vertices is more efficient in memory.

Figure 2.14 shows the window results for several sortings compared to the original sequence of vertices that preserves the order given by the input file. BFS and DFS sequences are generated by breadth-first and depth-first sortings, where the "rand" and "clockw" BFS versions differ in its tie resolution algorithm, either random or clockwise. Random sequences are not shown since the results are almost equal to DFS.

(a) NACA 2412

(b) NACA inviscid

(c) NACA 0012

Figure 2.11: Complete and zoomed views of the NACA test meshes. The lightly coloured vertices belong to the maximum frontier.

(a) NACA NS

(b) NACA laminar

(c) NACA (3D)

Figure 2.12: (Continues Figure 2.11) Complete and zoomed views of the NACA test meshes.

Figure 2.13: Maximum vertex window with breadth-first sequences in terms of vertices and cells.

Some meshes appear to be originally memory-aware, such as "NACA NS" or, to a less extent, "turbulent". Nonetheless, BFSort still achieves window sizes of 8% of the original sequence in those cases. In other words, only those sequences generated by BFSort are suitable for streaming; otherwise it would be necessary to store at least 25% of the mesh inside the FPGA. Also, it becomes clear that the behaviour of the original sequences in memory is not predictable.

Clockwise resolution has a subtle positive effect on 2D meshes, whereas it may even increase the maximum windows of 3D meshes. Overall, tie resolution does not affect window sizes significantly.

2.2.3 Seed selection

Seed selection is the process of choosing the vertex to be used as the starting point of BFSort. Seed selection is a separate problem itself; the BF algorithm will work regardless of the seed chosen. However, choosing specific seeds has proven to reduce the maximum vertex window. To our knowledge, this is the first work that considers seed selection as a means to minimize the memory footprint of a breadth-first-sorted mesh. This section analyzes the impact of selecting different seeds and suggests several algorithms to find a cost-effective solution.

In order to compare the possible variations between different seeds, we applied brute-force BFSort to all possible seeds, and measured the maximum vertex

Figure 2.14: Required size of the FPGA buffer in percentage over the complete mesh for various vertex-based sortings.

window obtained for each. Brute force gives best results but it is not a realistic approach. Even for small test meshes such as the NACA 2412 and turbulent, the time to find the best seed is higher than the computing time of a complex CFD simulation kernel. As the number of vertices grows, the time to compute all the seeds increases dramatically. The standard Breadth-First Search (BFS) has an algorithmic complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges. Since brute force search adds an outer loop to run BFS over all the vertices, the complexity is $O(V^2 + VE)$. A quadratic complexity quickly becomes intractable with the number of vertices and edges. On the other hand, selecting the wrong seed may increase the memory usage by a factor of two. Hence, a low cost heuristic is needed.

For all test meshes, brute force has helped us identify where the good and bad seeds are located. These are unstructured meshes and, as such, they show regions with different concentrations of the vertices, making up a representative general test set. An appropriate seed selection is less important for structured meshes, since the maximum window sizes are similar for all seeds.

As an example, Figure 2.15 shows the meshed profile of the NACA 0012 airfoil, where the longest frontiers are marked for the cases where the best and worst seed have been used. The frontier obtained with the best seed gives a good idea of

Figure 2.15: Maximum frontiers obtained with the best seed (left) and worst seed (right). The region of interest is shown in the main figures, whereas the boxed figures show the complete mesh.

how the size of the maximum vertex window can be diminished:

1. Some areas of the mesh show higher concentration points. These areas, located at the left and right sides of the wing, must be reached early in the traversal, since the frontier will be small at that point. As an example, the Figure shows that the best seed is located near the concentration point at the right of the image.
2. The best seed is displaced to the left in order to start with one concentration point early in the execution. Besides, the maximum frontier occurs immediately after the right branch of the frontier has passed through that concentration point, but right before the left branch passes through another concentration point. The conclusion is that dealing with various concentration areas simultaneously may result in an early growth of the frontier, which is not desirable.

On the opposite, the worst seed reaches the concentration areas late in the breadth-first traversal. Furthermore, both areas are reached simultaneously, generating a frontier that is twice the size of the frontier for the best seed. The next paragraphs propose two methods to select a seed for BFsort. We measure the quality of these methods in terms of two metrics: the execution time and the quality of the seed compared to the best and worst seeds.

Center of mass

This approach is based on the vertex coordinates, and takes its name from the calculation of the center of mass for a discrete solid body. The center of mass is

calculated assuming that the weight of each vertex is 1. Equation 2.1 calculates the center of mass $C\vec{M}$ based on the position vector $\vec{r}$ of each vertex. Since the coordinates of the CM may not match up with any of the vertices, the closest vertex to the CM, using the Euclidean distance, is selected as seed.

$$C\vec{M} = \frac{\sum_{i=1}^N \vec{r}_i}{N} \quad (2.1)$$

Some improvements are observed when the weight of each vertex varies according to a given property. In particular, a vertex can be assigned a high weight if its neighbours are close to it, thus giving more overall weight to the concentration areas. The individual weight of a vertex receives the name of *vertex mass*. The calculation of the center of mass with variable vertex mass is given by Equation 2.2.

$$C\vec{M} = \frac{\sum_{i=1}^N m_i \vec{r}_i}{\sum_{i=1}^N m_i} \quad (2.2)$$

CM works well for simple meshes with no more than two concentration areas. The CM method will force the seed to be close to these areas, allowing them to be reached early in the BF-traversal process. Additionally, the CM will be displaced to the most concentrated area, which will be computed first. More complex meshes cause the CM method to fail, specially when the vertices do not concentrate around points such as the wing front and rear in Figure 2.15, but around lines or curves. In addition, CM requires that each vertex has a position vector or coordinates, which is typical in scientific simulations but not necessary in more general graphs.

Bipartition and N-partition

Bipartition seed selection is based on the bipartition algorithm used to search in a sorted array. Figure 2.16 shows its pseudocode: the algorithm generates a temporary sequence and applies bipartition, searching through the vertex sequence for an appropriate seed.

A general bipartition search algorithm must operate on ordered sequences, since it is based on key comparisons of type “greater than” or “less than” to decide where to look next, to the right or to the left of the sequence. Likewise, the bipartition seed selection algorithm requires an initial temporary sort, as shown in line 2 of the algorithm. The key to be found is always the minimum value of the maximum window, and can only be known by computing its BF traversal, which is very costly. The bipartition algorithm reduces the number of required traversals from V to $\log_2(V)$, where V is the number of vertices in the mesh.

2.2. Locality optimization with mesh sorting

```

1:  $seed \leftarrow \text{random}(\text{size}(\text{mesh}))$ 
2:  $\text{ordMesh} \leftarrow \text{BF\_order}(seed)$ 
3:  $\text{startPt} \leftarrow 0$ 
4:  $\text{endPt} \leftarrow \text{size}(\text{ordMesh})$ 
5:
6: while  $\text{startPt} < \text{endPt}$  do
7:    $\text{midPt} \leftarrow (\text{startPt} + \text{endPt})/2$ 
8:
9:   {Get max window for start and end points as seed}
10:   $\text{ordMeshAux} \leftarrow \text{BF\_order}(\text{startPt})$ 
11:   $\text{startWindow} \leftarrow \text{get\_max\_window}(\text{ordMeshAux})$ 
12:   $\text{ordMeshAux} \leftarrow \text{BF\_order}(\text{endPt})$ 
13:   $\text{endWindow} \leftarrow \text{get\_max\_window}(\text{ordMeshAux})$ 
14:
15:  {Pick new interval to look for seed}
16:  if  $\text{startWindow} < \text{endWindow}$  then
17:     $\text{endPt} \leftarrow \text{midPt}$ 
18:  else
19:     $\text{startPt} \leftarrow \text{midPt}$ 
20:  end if
21: end while
22: return  $\text{startPt}$ 

```

Figure 2.16: Bipartition seed selection

Figure 2.17a shows the initial stages of the partitioning algorithm on the NACA 0012 mesh. The starting search interval is initialized to the entire mesh, taking points 1, 2 and 3 as probes. The first iteration executes a complete breadth-first traversal at each probe, measuring the maximum vertex window. Because point 1 obtains better results than point 2, the interval between point 1 and 3 is selected. A new search is started with points 1, 3 and the middle point 4 as the probe set. The process continues until the search interval comprises only one point, which is selected as the best seed.

Figure 2.17b shows a zoomed view into the last stages of bipartitioning in the NACA NS mesh. As the search intervals get smaller, the vertex windows between neighbours become closer. At these stages, it is easier for the algorithm to make the right decisions, since the information is very local. The slope is easy to follow, and bipartition will make downhill movements until it finds a local minimum. On the contrary, during the first stages of the algorithm it is easy to take the wrong choices, as the probe points are not close to each other and thus do not take information of the vicinity into account.

Because the quality of the temporary sequence is low, bipartition occasionally selects bad seeds. In order to minimize the chances to select a bad seed, an N-partition scheme may be used. Instead of partitioning the initial sequence in two,

(a) Taking the complete mesh as a start.

(b) Taking a small subset of the vertices as a start.

Figure 2.17: Execution of the bipartition algorithm on the NACA 0012 mesh.

the N-partition algorithm divides it up into N chunks. As there are more probing points where the maximum window is calculated, the seed will be selected with better knowledge of the search space. However, N-partition requires to execute $\log_N(M)$ BF traversals. Since $\log_N(M) > \log_{N'}(M) \forall N > N'$, we conclude that a high N in a N-partition scheme gives more accuracy than a low N at the cost of a higher computing time.

Figure 2.18: Maximum window size obtained by seed selection algorithms.

Finally, it is possible to save calculations at the start of the seed selection. In Figure 2.17b, we can identify a pattern that repeats periodically. If we choose the search interval as small as the period of that pattern, the search space is reduced compared with the complete mesh. In addition, the N-partition algorithm will be guided correctly to the best local seed. Furthermore, as the local minima do not vary greatly along the mesh (Figure 2.17a), the selection of the particular interval among the complete mesh is unimportant as long as it covers the identified pattern period.

Performance results

Figures 2.18 and 2.19 compare several approaches to seed selection. Obviously, brute force always finds the best seed, but it takes several orders of magnitude longer than the *CM* method to execute. On the other hand, selecting a default seed such as vertex 0 gives potentially bad values and the maximum window size is not predictable.

The *CM* method, while keeping the execution time low, selects a good seed compared to the optimal selection, and bounds the maximum window size to a small percentage of the complete mesh. However, its correctness depends on the type of mesh: if there are multiple vertex concentration areas, or if they cluster

Figure 2.19: Execution time of seed selection algorithms.

around a line or more complex shapes, *CM* will select seeds farther from the optimal.

N-partition finds better solutions than CM and, because it does not rely on geometry information (i.e. coordinates), it works for all meshes. However, higher computing times are required, since the calculation of maximum vertex windows requires to execute graph traversals for each probe point. Generally, we would consider N-partition only if the following processing stage is long. Also, if the mesh sorting application is independent of the HPC code, it will be possible to sort the mesh once and use it for simulation several times. In the particular case of our CFD code, the mesh is processed iteratively, where the number of iterations ranges between the hundreds and the tens of thousands depending on the convergence requirements of the partial differential equations. In such case, using bipartition or N-partition is a good choice that does not increase the execution time sensibly, but obtains vertex windows close to its lower bound.

2.3 A streaming methodology for pipelined kernels

The previous Section proposed a mesh sorting method to improve data locality so that the transfers between memory levels are minimized. In this Section, sorting is combined with other techniques in a full methodology that improves the memory bottleneck in FPGA-accelerated HPC algorithms processing unstructured meshes.

Regardless of whether the computing device is a CPU or a FPGA, the memory hierarchy is organized around two physical levels. In the case of a CPU, these are the off-chip main memory, and the on-chip caches. Modern CPU caches also have up to three sub-levels. Similarly, FPGA accelerator boards have a main memory as the off-chip level, plus a wide variety of memory resources inside the FPGA as the on-chip levels. While, for CPUs, these physical levels are arranged in a standard memory hierarchy with traditional cache mechanisms (block transfers with different block sizes, associativity, etc.), the memory resources of FPGAs allow greater flexibility. We propose a hybrid hierarchy combining sequential and random accesses at different levels which, in combination with a sorted mesh and further compression techniques, reduce memory traffic significantly.

The methodology is based on the combined application of BFsort, coding and compression techniques. Sorting allows efficient stream transfer between the memory and the FPGA, improving data locality and avoiding redundant data requests. Coding achieves a compact representation of the mesh connectivity. Differential compression reduces the size of the mesh data. This complete methodology has been validated by results on a hardware system that has been implemented in our research group.

2.3.1 Hybrid sequential/random access

Figure 2.20 compares a pure random-access cache system with our custom memory system. Caches work with block transfers, also a form of size-limited burst accesses. For example, a request for vertex 3 will trigger the transfer of block {2-6}. Further requests for vertices 2 to 6 will not generate a cache miss. However, cache controllers require complex logic to implement techniques such as the replacement policies or the associativity. Sequential mode is more efficient in the available FPGA memory controllers, specially in the typical situation where memory controllers are implemented as soft cores that use part of the FPGA resources.

Because the application (partial differential equations) and the dataset structure (meshes) are well known, it is possible to stream the mesh in a predefined

Figure 2.20: A traditional cache system (left) compared to a custom hybrid sequential/random system (right) on an FPGA.

order of computation. This is the sequential part of the hybrid approach. Then, the kernel can randomly access the data inside the FPGA, as long as the data requests are satisfied locally. This is, data requests must be addressed to the small portion of the mesh that resides on-chip. Because the mesh is transferred sequentially to the lower memory level, an attempt to randomly access missing parts of the mesh results in an indefinite kernel stall.

To ensure that the computational kernel is served all data requests, each vertex is kept inside until all the computations involving its data are finished. The interval that a vertex is kept at the lower memory levels is given by its vertex window. Hence, BFsort will greatly reduce the required size of the on-chip memory by reducing the maximum vertex window.

2.3.2 Coding and compression with BFsort

BFsort aims at improving space locality by creating a processing sequence that places neighbour vertices as close as possible to each other. BFsort is applied as a preprocessing stage; it can even be part of the mesh generation process.

Frame coding packs the description of the vertices and edges in a single compact frame that removes the redundant connectivity information of the mesh described as vertex and edge lists. It applies to transfers involving the complete mesh. A differential compression scheme is also applied to both connectivity and algorithmic data associated to vertices and edges. It will be shown that the pro-

posed encoding and data compression schemes benefit from the improved data locality that results from the sorting stage.

The coding and compression techniques should be simple yet effective. Simple, because the hardware implementation must operate at the maximum clock frequency possible in conjunction with pipelined kernels, and also due to the limited hardware resources inside the FPGA. Effective, because the resulting compression factors determine the maximum throughput (i.e. vertices/second) of the decoder, which is bound in the first place by the maximum bandwidth of the DDR memories.

Connectivity coding

When the mesh description is based on edge and vertex lists, redundancies relate to the appearance of vertex IDs in the edge list and edge IDs in the vertex list. Here we assume processing in terms of edges, where each edge stores two vertex IDs, one for each endpoint. In such case, the vertices do not store the edge IDs. Consequently, each vertex ID will appear as many times as edges connect to it. If the list of edges is arranged in a way that all edges sharing a common endpoint are grouped together, it is possible to combine both lists in a single frame with no redundant vertex IDs. Upon arrival at the FPGA, the frame is decoded and both lists are recovered.

This proposed frame is organized as a sequence of blocks, arranged in subsequences starting with a single block representing a *central* vertex and followed by a variable number of edge blocks. Each edge contains the vertex ID of one endpoint, while the preceding central contains the second endpoint ID. For example, the sequence of IDs $\{\mathbf{n_0} \ n_1 \ n_2 \ n_3\}$ describe central vertex $\mathbf{n_0}$ and edges n_0n_1 , n_0n_2 and n_0n_3 . A flag must be included to distinguish between central vertex and edge blocks. Central vertices are written in bold.

Additionally, only the first appearance of any vertex ID in the frame is followed by the algorithmic data associated to the corresponding vertex. This is possible because the next uses fall necessarily within the window of that vertex, so the data will already be loaded onto the FPGA. We represent new IDs as underlined. These are marked with another flag. Finally, edge data are always included after the ID of the edge block. In directed graphs, edge data include a third flag to indicate direction.

Figure 2.21a shows an example of a small mesh, sorted according to the original labeling of vertices, and the frame describing it. In this case, we follow a tie resolution strategy based on increasing IDs as given by the original sequence. Each edge is included only in the subsequence where its first endpoint appears as central (i.e. it does not appear when the central vertex corresponds to its second endpoint).

Figure 2.21: Frame coding without ID compression and with ID compression.

Connectivity information can be compressed to reduce the size of the frame as in the second frame of Figure 2.21a. On the one hand, central vertices appear in order so their IDs are not needed, as they can be restored with a counter. Only the central flags are required (marked \$). On the other hand, non-central IDs use a simple difference compression with respect to the previous ID. A *consecutive* flag (marked +) is used when adjacent IDs are consecutive. This connectivity compression reduces the number of IDs in the edge blocks from 38 to 14.

In general, vertex labeling is decided during mesh generation. It is not necessarily random (e.g. IDs in Figure 2.21a are assigned approximately from top to bottom), but it is certainly unpredictable. By applying BFSort to the mesh in Figure 2.21a with vertex 8 as seed, the labels are reassigned as in Figure 2.21b. The first frame corresponds to the proposed coding strategy without ID compression. In the second frame, the original 38 IDs are reduced to 7 by sorting the mesh appropriately.

Data compression

In addition to connectivity information, the frame includes algorithmic data associated to vertices and edges. A simple yet efficient approach is to apply differential compression between values corresponding to the same parameter of consecutive vertices or edges. Each value may be transferred in the compressed or the original form depending on a threshold. By analyzing the values of the variables during execution for a subset of the data sets, we set a threshold for the differences and their corresponding reduced representation. The threshold value is a trade-

off between having a small representation and allowing a high number of values to be represented. Thresholds are set at compile time. Dynamic thresholds are impractical so far because they require costly mesh preprocessing and dynamic management of the memory interface modules. Each variable requires a flag in order to switch between both representations.

Differential compression works best if consecutive vertices or edges in the frame hold similar data values in a per variable basis [KSY10][EFF00]. The proposed BFSort algorithm places adjacent vertices in the mesh consecutively in the frame, thus favoring data similarity. In addition, different tie-resolution strategies (line 12 in Figure 2.8) affect the order of the vertices and thus the efficiency of differential compression. Among the possible tie-resolution strategies, those that also preserve the adjacency between the neighbours, such as clockwise or counterclockwise resolution, improve the compression of some parameters (e.g. vertex data dependent on the vertex location), but not necessarily all of them (e.g. edge data dependent on the edge direction). The effect of random and clockwise tie-resolution strategies on the connectivity and algorithmic data is analyzed in Section 2.4.2.

2.3.3 Implementation details

The architecture of the processing element (PE) is composed of a host and an accelerator, as shown in Figure 2.22. The host contains one general-purpose processor and its main memory. The accelerator board contains one or more FPGAs and enough on-board memory to store the mesh processed by the PE. The current implementation is restricted to a single PE.

Amdahl's law implies that a significant part of the algorithm must be executed in the accelerator. The host applies the compaction techniques to the mesh, sends it to the accelerator memory at the beginning of the execution (1), collects the updated mesh at the end (4) and further restores it to the original format. At each iteration of the iterative computational kernel, the on-board memory pushes the complete streamed mesh into the FPGA memory, saving the results back (2). As explained in Section 2.3.1, the FPGA cannot request a random memory position to the on-board memory, but only randomly access the portion of the mesh that resides in the FPGA at any given moment of the execution.

The board used as demonstrator for this work has three DDR memories. We take advantage of this fact by storing read-write (RW) data in one memory and writing the results into another. After each iteration, the read and write memories are swapped (3). We use the third DDR memory to store read-only (RO) data. This technique allows a $3\times$ faster memory access, since the three memories can work in parallel. However, the programmer must manually specify which data are part of the RO and RW data. This is discussed later in this Section.

Figure 2.22: Target architecture showing the data movements between host and accelerator.

Figure 2.23: Interface between the memory and the kernel for the read-write part of the mesh. At each iteration, the read and write memories are swapped.

2.3. A streaming methodology for pipelined kernels

The hardware coding system is divided into four modules: *block decoder*, *data decompressor*, *data compressor*, and *block encoder*. Figure 2.23 shows the top-level architecture of the read-write system. The RO system is identical except that there is no encoder nor compressor, since the RO data are not written back to memory. The decoder reads and writes encoded memory words. Since the mesh is processed in terms of edges, the decompressor must feed the computational kernel with one edge per cycle. The blocks are separated by dual-clock FIFOs to decouple the clock domains of each module and allow fine performance tuning. All the blocks are synchronized by a custom handshake mechanism that makes use of the *full* and *empty* flags of the FIFOs and synchronizes the pipelined kernel with the memory interface.

Mitra and Chiueh [MC02] implement a similar interface with a custom mesh encoding. They further reduce on-chip memory size by prefetching only a small fraction of the window, at the cost of duplicating transfers. The prefetching points can be calculated at preprocessing time, so it is assumed that the extra transfer overhead is hidden by the prefetching mechanism. However, the iterative nature of mesh-based scientific kernels, which changes the data across iterations, would force us to have a slower and more complex output encoder if we implemented prefetching on our system. Output data would have to be kept inside the FPGA until it was added to all the prefetching points in the mesh stream, greatly increasing the on-chip storage requirements.

Analytical model

The following model is used to evaluate the performance of the coding and compression system. The performance is given by the number of edges per second that it is capable of serving to the kernel. The system must serve the edges at least at the same rate as the memory. If BW_A is the effective bandwidth of the memory (arrow A in Figure 2.23) and S_A is the average block size in the frame, then Equation 2.3 gives the number of blocks served to the decoder per second.

$$\frac{BW_A}{S_A} = |Blk|_A \text{ (blocks/s)} \quad (2.3)$$

Only the edge blocks translate into an edge served to the kernel at interface E. The central blocks are necessary for edge reconstruction purposes, but are cleanly discarded at the decoder. Therefore, the speed in edges/s at interface E is equal to the speed in blocks/s at interface A (Equation 2.3) weighted by the proportion of edge blocks (non-centrals):

$$|Blk|_A \times \frac{E}{E + V} = |Edg|_E \text{ (edges/s)} \quad (2.4)$$

δ is the ratio of edges to vertices in the mesh (E/V), and a simple algebraic manipulation gives the following expression:

$$\frac{E}{E+V} = \frac{\delta}{\delta+1} \quad (2.5)$$

Substituting Equations 2.5 and 2.3 into Equation 2.4 results in Equation 2.6.

$$|Edg|_E \text{ (edges/s)} = \frac{BW_A}{S_A} \times \frac{\delta}{\delta+1} \quad (2.6)$$

Assuming that the decompressor outputs one edge per cycle, the minimum frequency of the decompressor is $|Edg|_E$ (Hz).

Split-frame optimization

Figure 2.22 showed multiple memories for read-only and read-write data. This effectively multiplies the available bandwidth to/from the FPGA, but requires splitting the data into two frames. The following paragraphs explain how to split the mesh according to the previous analytical model.

With several memories, balancing is central to obtaining maximum speedup. Figure 2.24 shows the format of a streamed mesh, as explained in Section 2.3.2, split into RO and RW frames. Note that the left frame contains the connectivity as subsequences of central block plus edges while the right frame only contains vertices. The processing sequence is given by the frame on the left. The CTRL field determines the type of block. An all-zero value represents a central vertex, while the remaining values represent different combinations of vertex and edge data, as well as absolute/differential representations of the variables.

Two streaming memories are balanced if it takes the same amount of time to transfer their frames. For example, for two frames where one is edge-based (subindex e) and another is vertex-based (subindex v),

$$\frac{F_e}{BW_e} = \alpha \frac{F_v}{BW_v} \quad (2.7)$$

where F is the total size of the frame (either based on edges or vertices) and α is the imbalance factor, meaning that the edge-based frame requires α the time to transfer the vertex-based frame. Obviously, $\alpha = 1$ implies two perfectly balanced frames.

Because F_e and F_v vary widely along meshes, it is easier to use δ instead: if V is the number of vertices, E is the number of edges and $\delta = E/V$, then $F_v = VS_v$ and $F_e = (V+E)S_e$. Dividing both sides by V , we obtain the following expression:

$$(1 + \delta) \frac{S_e}{BW_e} = \alpha \frac{S_v}{BW_v} \quad (2.8)$$

where S_e and S_v are the average block sizes of each frame. If we assume memories with the same bus widths and frequencies, the imbalance factor can be calculated from (2.9).

$$\alpha = (1 + \delta) \frac{S_e}{S_v} \quad (2.9)$$

In case that both frames were vertex- or edge-based, the $(1 + \delta)$ factor would not apply. In sum, memory balancing requires evaluating two aspects:

- The average size of each variable, accounting for differential compression effects. Each variable is assigned to one of the frames, increasing its size and affecting the average block size S .
- Vertex- and edge-based frames. If the kernel processes by edges, at least one frame must be edge-based in order to give the processing sequence and the connectivity. Vertex data may be placed in either vertex-based frames, or edge-based frames if they are attached to the first edge incident to the vertex. A $(1 + \delta)$ factor must be applied between vertex- and edge-based frames to compensate for the different number of blocks.

Reference implementation

The methodology has been implemented in a machine composed of an Intel Core i7 at 2'66 GHz with a Gidel ProcStarIII FPGA-based accelerator board attached via PCI express. The FPGAs are Altera Stratix III EP3SE 260 speed grade 3. Three DDR2 memories are available on-board: two SODIMMs with 4 GB each at 24 Gb/s and one SODIMM with 512 MB at 16 Gb/s [Gid12].

The hardware implementation has been developed within our research group to evaluate the results of this methodology in the context of an industrial project. As it is not work undertaken by the author, its details are not included here. We refer the reader to the complete publication [BCL⁺14] for a more complete description of the hardware blocks.

However, we show the frequency and area results obtained in order to apply the aforementioned analytical model. We use the two-dimensional CFD meshes; 3D CFD and other algorithm datasets require similar interfaces, although the exact balancing, frequency and area numbers might change, as well as the kernel implementation. The RO frame uses one 186'7 MHz memory and the RW frame uses two memories at 125 MHz in the configuration shown in Figure 2.22. The frequency and area results of the reference implementation for the RO and RW memory interfaces are shown in Tables 2.2 and 2.3, respectively.

Figure 2.24: Left: edge-based read-only frame. Right: vertex-based read-write frame. Dark blocks represent vertices; light blocks represent edges in the connectivity.

The edge normals and the vertex volumes are stored in the RO frame. The vertex coordinates, velocities, scalar densities and density $\times$ energy, are stored in the RW frame. The average RO block size is 76 bits and the memory throughput is $BW_A = 24$ Gb/s. We assume that $\delta = 3$, which is true for all our 2D meshes. Applying Equation 2.8 with these values gives an α value of 1.07, which means that the RO frame takes 7% extra time to transfer than RW. Since it is not possible to achieve perfect balancing, the real speedup of the split-frame implementation with respect to the single frame is 1.93 instead of the theoretical 2.

	Decoder	Decompressor
Logic Usage	< 1%	< 1%
Comb. LUTs	1036	819
Memory LUTs	20	40
Registers	1492	2200
Block mem. bits	3904	1672
Max. clock freq.	302 MHz	304 MHz

Table 2.2: Frequency and area results, RO frame

2.4. Performance results

	Encoder	Decoder	Compressor	Decompressor
Logic Usage	3%	3%	< 1%	1%
Comb. LUTs	3326	3597	1005	954
Memory LUTs	80	80	500	260
Registers	4836	4768	1646	2772
Block mem. bits	70544	4968	1635	1032
Max. clock freq.	295 MHz	318 MHz	299 MHz	293 MHz

Table 2.3: Frequency and area results, RW frame

Replacing the values in Equation 2.3 gives a required frequency of 316 MHz. In such case, reducing the size of the mesh to 0.35 times its original size, the performance of the memory system improves by a factor of 2.85 with respect to an unoptimized streaming memory system. According to Table 2.2, the RO decoder obtains a frequency of 302 MHz, which is 14 MHz or 5% below the required frequency. This reduces the speedup down to 2.7. Together with the split-frame implementation, this makes up to a theoretical 5.2 speedup in the memory performance with respect to a random-access cache system. The real speedup is higher, since random accesses are more expensive in current controllers than sequential ones.

The RW frame uses the 16 Gb/s memories and has an average block size of 141 bits. The same calculations give a required frequency of 113 MHz. Since the maximum frequencies of the encoder and compressor more than double this requirement, the RW frame does not affect performance with the current setup.

2.4 Performance results

This Section gives two performance evaluations of the proposed techniques. First, Section 2.4.1 shows the results of mesh sorting on a CPU with caches. These results confirm that, with random access as offered by traditional cache systems, the processor takes advantage of implicit prefetching each time that a cache line is requested, minimizing cache misses. This implicit prefetching is affected by space locality, and therefore the proposed sorting mechanism improves the overall performance of the scientific simulation.

Section 2.4.2 completes the formal analysis of the model in Section 2.3.3 with measurements of compression in the test meshes. Apart from showing that the methodology is useful, these results show that the combined application of sorting, coding and compression obtain better results than each of these techniques applied separately.

2.4.1 Data locality improvements on a multicore

A detailed profiling has been carried out with traditional time analysis on a CPU and Cachegrind [Cac], counting L1 and L2 cache misses generated by every function of the code. A simple test application has been coded; this converts the mesh into three position arrays, each storing one coordinate of the vertices, and one adjacency array, storing the connections between vertices. The computation involves, for each vertex, adding up all of its coordinates and also the coordinates of its neighbours. This computation stage, although extremely simple, generates nearest-neighbour dependencies, the same memory access patterns as many common scientific codes.

Two instances of the program have been run. While the first instance computes the vertices in the sequence given by the input file, the second instance preprocesses the data structures, using the adjacency array to breadth-first traverse the mesh, and rearranging the data structures. Measurements for the second code already include the cost of rearranging the data structures, which involves a considerable amount of memory operations.

Mesh	# vertices	Execution time (s)		
		Original	Vertex BF	S
NACA1	3070	0.080	0.060	1.33
NACA2	7908	0.145	0.139	1.04
NACA3	25651	0.795	0.791	1.01
HILIFT	54165	1.968	1.709	1.15
VFINE	494128	2.854	1.108	2.58

Table 2.4: Timings and speedups of the test code with two vertex sequences

Mesh	# accesses	% D-L1 misses		% L2 misses	
		Orig.	BF	Orig.	BF
NACA1	$1,4 \times 10^8$	5.7	5.1	0.1	0.2
NACA2	$3,6 \times 10^8$	5.9	5.0	0.0	0.2
NACA3	$9,5 \times 10^8$	6.9	7.5	5.1	5.1
HILIFT	$2,2 \times 10^9$	16.1	7.3	5.0	5.3
VFINE	$6,1 \times 10^9$	4.9	1.8	3.2	1.4

Table 2.5: Data-L1 and L2 cache misses, measured as a percentage of the total accesses to the memory system.

Table 2.4 shows the timings for both codes on an Intel Core2 Quad Q8400 at 2.66 GHz. There is little to no improvement for the smaller datasets, partly because of the extra cost of sorting the data in the main memory. However,

the bigger mesh obtains a considerable speedup of 2.58. Not only the processing sequence has changed, but also the whole data structures have been physically reordered so the cache block mechanisms work well. The breadth-first sort is expected to have a greater impact on bigger datasets, since these will not only use more memory, but also higher levels of the hierarchy that slow down the executions more easily than the lower levels. The speedup depends on the quality of the original sequence.

Table 2.5 compares the cache performance for both test codes as measured with Cachegrind. Except for NACA3, all meshes improve the L1-cache performance with BFsrt. For the L2 cache, only VFINE achieves a considerable improvement. Accounting for the fact that the BF code includes the cost of rearranging the memory structures, and by the effect of Amdahl’s law, we claim that BFsrt will perform better with codes that iterate many times over the mesh, as is usually the case in scientific codes.

2.4.2 Streaming methodology on an FPGA accelerator

Compaction of the connectivity data

The connectivity data are around 30% of the total data in a CFD mesh. Figure 2.25 shows the results of applying four different combinations of the techniques described in Section 2.3.2. We use the compression ratio, as defined by Salomon [Sal04], in order to measure the difference in size between the original and optimized streams:

$$CR = \frac{\text{new size}}{\text{original size}}$$

According to this definition, if a compressed mesh occupies 60% of its original size, the compression ratio is 0.6.

The first strategy encodes the mesh from a random sequence, using a general-purpose width of 32 bits for the vertex IDs. Then, the same encoding is applied with a breadth-first sequence. The last two versions further reduce the ID size to the minimum. Identifying the minimum bit-width required for correct representation is done as part of a preprocessing stage.

The coding strategy alone with a random sort obtains a CR of 0.57 on average. BFsrt reduces it to an average 0.28. This improvement comes exclusively from removing consecutive IDs of adjacent vertices. With compressed IDs, the CR is further reduced down to 0.21. However, just by improving the locality of IDs with BFsrt, the required bit width is further reduced in five bits. This, together with the increased amount of removed consecutive IDs, achieves a CR of 0.10.

Figure 2.25: Compression ratio (CR) of the encoded connectivity data with different sorting methods and with or without quantization.

Compression of the algorithmic data

In the CFD meshes under consideration, the algorithmic data associated to the vertices comprise velocities, volume, coordinates and two densities. The edges contain their normals.

The algorithmic data are not affected by the encoding strategy, since the encoded representation keeps one variable of each type per vertex or edge, the same as the original mesh. However, by improving space locality, the values may be close to those of previous vertices or edges, due to the smooth changes of physical variables mentioned in Section 2.3.2. For example, the velocities and the pressure values in neighbours are likely to be very similar. This does not necessarily apply to all variables. Table 2.6 illustrates the varying impact of differential compression in the algorithmic data of the CFD application. The importance of compressing the algorithmic data depends on several factors:

- The ratio of algorithmic to connectivity data.
- The physical nature of each variable, which affects the efficiency of differential compression.
- The range of values that each variable may take, which allows defining suitable thresholds.

2.4. Performance results

	original (%)	compressed (%)
Vertex data		
coordinates	7.18	2.97
volume	2.39	1.09
densities	4.79	1.76
velocities	7.18	3.10
Edge data		
normals	47.07	22.32

Table 2.6: Differential compression variation in variables from the CFD application. Sizes given as percentage of the original mesh.

	SD	FC	DFC
coordinate X	32	27	15
coordinate Y	32	28	20
coordinate Z	32	28	20
normal X	64	33	25
normal Y	64	33	25
normal Z	64	33	25
volume	64	36	20
density	32	32	24
density $\times$ energy	32	29	21
velocity X	32	32	24
velocity Y	32	32	24
velocity Z	32	32	24

SD. Mixed single/double precision

FC. Full-custom precision

DFC. Additional width for two-level compression

Table 2.7: Bit-widths for different quantization choices

Applications developed for fixed architectures (e.g. CPUs, GPGPUs) are restricted to a few types of data formats, such as single- or double-precision floating point. However, FPGAs allow the programmer to fine-tune the data widths. Table 2.7 introduces three choices for CPU and FPGA scenarios where compression has been applied. The original CPU version declares all the variables in double precision. An improvement from the original code is presented in column SD, where some variables are reduced from double to single precision.

A second improvement is obtained by applying quantization techniques that determine the exact fixed-point data widths required to keep the precision and overall stability of the algorithm. The quantization results have been obtained by means of the fast quantization techniques presented by Caffarena et al. [CCLF10] and Sedano et al. [SLC12], and are shown under column FC. These quantization values are given as an input to the compressor and are not central to this work. Quantization is the only compaction technique that implies precision loss in the results. The rest of them (differential compression, coding scheme, etc.) are loss-less compression techniques. In the case of our CFD algorithm, the quantization process has been driven by precision requirements imposed by the end users of the application to certain error measurement variables: 10^{-4} in the drag and 10^{-3} in the lift variables. Since these are global mesh variables, we have further imposed a maximum error of 10^{-2} at each solution variable per vertex at each iteration. In general, modern quantization techniques make possible to reduce the data size up to the precision loss allowed by the end user. The quantized and accelerated code is then validated against the original floating-point implementation. Finally, column DFC presents the quantization results for the two-level differential compression explained in Section 2.3.2.

Figure 2.26 compares the algorithmic data compression compared to the original code with full double precision. By using a hybrid single/double precision solution, the compression ratio is 0.78 in average. With full-custom precision as given in Table 2.7 FC, this ratio is reduced to an average 0.5. The two-level custom precision with differential compression further reduces the compression ratio in 8% for the 2D meshes and 3% in the case of 3D meshes. Surprisingly, BFSort only improves the differential compression an additional 2% for 2D meshes, and 1% for 3D meshes. This limited improvement is due to the wide dynamic ranges of the variables in these datasets.

Overall compression

The total compression is the sum of connectivity and algorithmic data accounting for the presence of each type of data in the mesh, plus minor low-level effects such as the padding in the streamed mesh.

Figure 2.27 shows the compression with and without sorting as generated by

2.4. Performance results

Figure 2.26: Compression ratio of the algorithmic data

Figure 2.27: Overall compression results in percentage of the original mesh. The first bar represents the original mesh, while the second bar represents the compressed mesh after applying the discussed techniques except for sorting. The last bar includes sorting.

the preprocessing stage. The average compression ratio is 0.34. All meshes stay close to the average, although 3D meshes tend to deviate more than 2D meshes. Comparing the second and third bars of each dataset shows that, additionally to the intermediate buffer optimization discussed in Section 2.2.2, sorting improves the performance of other compaction techniques in 3.3% and 1.8%.

With a fixed quantization of 24 bits for the algorithmic data, Isenburg et al. [IG03] obtain an average CR of 0.11. The datasets presented in this work obtain worse results in terms of required bits per vertex, but the gap is smaller in CR since the connectivity is more dense and the size in terms of vertices is less significant.

Tie resolution also has an effect on compression. We have measured differences of 2% between clockwise and random strategies. The compression results of both choices in the CFD NACA 0012 data set are given in Table 2.8.

	Random	Clockwise
Connectivity	0.13	0.10
Algorithm	0.41	0.41
TOTAL	0.33	0.31

Table 2.8: Compression ratio with two tie resolution strategies

Overhead analysis

The previous techniques introduce two main sources of overhead. The first is the extra time required by the preprocessing stage, which sorts and creates the mesh streams. According to Table 2.1, the most demanding mesh takes around 10 seconds to preprocess. The processing stage in CFD codes has a variable length, since it depends on convergence requirements that greatly differ along meshes, and thus it is hard to give the overhead in percentage. However, 10 seconds is negligible as compared to standard runs, where the algorithm iterates over the mesh thousands of times with wallclock times in the range of hours at best. The second source of overhead is a result of the hardware implementation. This uses FPGA resources that cannot be used for the computational kernel. Tables 2.2 and 2.3 demonstrate that the used resources are very low compared to the available resources (3% or below).

Another possible source of overhead is the initial transfer of the streams to the on-board memory from the host. This is not generated by the application of the proposed techniques, but by using any kind of accelerators. However, accounting for the current PCIe transfer speed, this overhead is also negligible compared to the simulation runtimes.

2.5 Conclusions

In this Chapter, we have introduced the problem of memory performance in scientific codes that process unstructured meshes. We have also proposed several optimizations aimed at general-purpose processors as well as FPGA-based hardware accelerators.

First, we have presented a method to sort the mesh and improve reference locality. The reduced memory footprint obtained allows us to remove cache misses. Our experiments show that this method can reduce the memory usage to a small percentage of the mesh, as opposed to the unpredictable behaviour of the traditional memory-oblivious computing sequences. Any algorithm that operates on meshes will benefit from a BF-sorted mesh, as long as the dependencies between vertices are similar to nearest-neighbour. The effect of changing the processing sequence and the order of the mesh in memory has also been analyzed.

Also, seed selection has been identified as a problem worth solving in order to optimize memory performance. We approach the problem through two algorithms: center of mass, which offers a fast seed selection, and N-partition, which gives more accurate results, particularly with complex meshes.

Next, we propose a methodology to approach the memory bottleneck in FPGA-based accelerators. Standard compaction techniques are combined with a mesh coding strategy that removes redundant connectivity information. Furthermore, sorting the mesh greatly improves spatial locality and the effects of coding and, to a lesser extent, compression. The methodology is applied to a heterogeneous system composed of a host CPU and an FPGA-based PCIe accelerator board. Together with a streaming approach and a sorting method, the traffic between the FPGA and the board memory is greatly reduced. This comes at the affordable cost of a minimal preprocessing stage at the host CPU and a small coding system to interface between the FPGA computational kernel and the board memory.

The results show that an efficient sequence is critical for the streaming approach to work. From all the tested sorting methods, BFSort achieves the lowest memory footprint inside the FPGA. The sequence generated by BFSort also improves the efficiency of the coding strategy by a factor of two. This compression is sustained regardless of the target algorithm. Furthermore, it is also independent of the memory and FPGA technology, thus being able to sustain the improvements as technology advances.

Chapter 3

Refactoring of sequential codes for distributed systems

Porting a piece of code to a heterogeneous distributed architecture involves partitioning the code and mapping each partition to a processor. Here, a *processor* is defined as a hardware component with computational capabilities, such as a general-purpose processor, a GPGPU or an FPGA. Each processor is expected to perform differently, and different parts of the code are supposed to be mapped to the processor where maximum performance can be achieved, while keeping the communications between processors as low as possible. After mapping, a refactoring stage rewrites the code and inserts communications so that the behaviour of the distributed code is equivalent to the original.

Extracting parallelism (partitioning, mapping) is a separate problem from modifying the code to run in a distributed memory machine (refactoring). When refactoring is done by hand, the programmer uses a technique known as *function outlining* (or *procedurization*) that separates partitions into different functions and redesigns the control and data flow. In some parallel systems, functions are called between processors in a client-server fashion using a remote invocation framework. Some examples are ONC-RPC [Blo92], GridRPC [SNM⁺02], or CUDA and OpenCL kernels [SK10][GHK⁺12]. Control flow between functions is restricted to jumping to the callee (server) and returning control back to the caller (client). In contrast, compilers work with Control Flow Graphs (CFGs), which are divided into BBs. A BB is a chunk of code with single entry and exit points. If one instruction is executed, all other instructions in the BB are guaranteed to be executed.¹ Control flow to/from a BB is less restrictive than to/from a function.

¹We assume that exceptions are not allowed or can be handled in a different way. This is a reasonable assumption in high-performance codes, as exceptions are generally costly in terms of performance and therefore are typically avoided.

Function-based control flow restricts the applicability of automatic optimizations, specially when the original code contains large functions that require distribution among several processors. Although it is possible to redesign the control flow from an intelligent compiler, automatically dealing with non-canonical loops and improper regions becomes problematic. Section 3.3 further elaborates on function- and block-based control flow, explaining their properties and differences. In sum, a control flow mechanism that simulates the behaviour of BBs is likely to be more efficient than existing RPCs, both in terms of performance of the executables and compile time.

This section introduces techniques to address the refactoring of sequential applications to distribute them in a parallel system. We present an algorithm for the compiler to partition CFGs. Subsequently, we introduce the concept of RBBs, an improvement over the traditional RPCs adapted to the requirements of the internal representation of modern compilers in Static Single Assignment (SSA) form. RBBs allow the compiler to functionally partition the program into a concurrent application based on the parallelism information, with minimal effort and without seriously compromising performance. The contributions of this Chapter can be summarized as follows:

- RBBs, an alternative control flow mechanism to that of traditional RPCs, with flexible and peer-to-peer distributed control flow, suitable for automatic code refactoring.
- An algorithm for the compiler to split codes, represented as CFGs, into smaller portions of code based on partitioning information, which can be transformed into RBBs.
- A communications Application Programming Interface (API) and its CPU library implementation that allows RBBs to communicate between each other.
- Results that show that the proposed system can be used to generate parallel executables from sequential code with efficient communications to pass control flow information.

Section 3.1 gives a brief overview of previous related work and enumerates the key differences of our proposal. Section 3.2 briefly introduces a complete compilation toolchain that sets the context of our work. Section 3.3 discusses function and basic block control flow in the context of distributed systems and presents an algorithm to refactor the sequential code in separate modules that target different processors. Section 3.4 introduces the concept of RBBs, a concept similar to RPCs that is suited for compiler transformations and allows passing control flow

in a distributed system. Section 3.5 shows proof that RBBs obtain good communications performance, evaluates the efficiency of the current implementation and presents a case study with a standard benchmark. Finally, in Section 3.6 we summarize the work and contributions, and propose future work.

3.1 Related work

3.1.1 Code refactoring

Code refactoring has been used in the past for multiple reasons such as simplifying legacy code or improving performance. Lakhota and Deprez [LD98] restructure large, non-cohesive functions into smaller, cohesive ones to reduce code complexity. An algorithm is presented, *Tuck*, that works on the CFG of a program. Starting from a set of seed statements inside the function, the transformations generate functions where each seed statement is the entry of a smaller function. The algorithm has some similarities with the algorithm that we will present later: for example, we also identify starting points for our functions. However, we focus on CFGs in SSA form, and the function is split in terms of BBs rather than single statements. We also cluster the blocks together in smaller functions as to minimize control communication rather than simplifying the code, which should result in less functions being generated.

Zhao and Amaral [ZA07] analyze the performance of oversized functions that cannot be easily inlined, identifying portions of *cold code* that do not affect performance significantly and extracting them away from the *hot code* and into small cold functions. The result is a set of smaller hot functions that are called more often than the cold ones, and that are suitable for inlining after the refactoring. The search for hot and cold code is done over conditional constructs such as if/then/else or switch statements, as these are good candidates to find cold code from the branches that are not taken often. Therefore, the outlining stage avoids having to deal with loops and improper regions. Our refactoring algorithm allows any general outlining scheme as long as the BBs do not need to be split.

Liao et al. [LQVP10] also use outlining to break a program into small kernels for individual optimization. The work is based in the ROSE source-to-source compiler, which we will use in Chapter 4). The algorithm has several steps in common with ours, but the transformations are done over the Abstract Syntax Tree (AST) of a high-level programming language. Our algorithm works with the lower-level control and data flow graphs of LLVM's Intermediate Representation. This allows us to disregard some high-level details, such as classes or public/private encapsulation, which are resolved already by the front-end of the compiler.

3.1.2 Distributed control flow

Passing control between processes in distributed systems has been extensively studied. The seminal work by Birrell and Nelson [BN84] present the first RPCs implementation that defines call semantics such as argument passing rules, binding between local and remote functions and data packing/unpacking. A similar approach for object-oriented software is introduced by Wollrath et al. [WRW96]. A sophisticated version of RPC, Remote Method Invocation (RMI), is used to communicate objects, allowing the creation of object stubs and even implementing distributed reference-counting garbage collection.

Chang et al. [CCE99] show one of the first RPC implementations for Multiple Program Multiple Data (MPMD) distributed systems. Instead of implementing the system on top of message-passing, they use *Active Messages* (AM). Because AM is restricted to Single Program Multiple Data (SPMD) programs, a minimal multithreaded runtime system that extends AM with MPMD capabilities.

Seymour et al. [SNM⁺02] apply the concept of RPCs to Grid Computing. This is achieved by adding asynchronous parallel tasking features to standard RPCs. Asynchronous tasks are also important for HPC, since this allows processes to continue working if there are no dependencies between tasks, instead of blocking to wait for the called remote function.

Other popular RPC variants are Ninf-G [TNS⁺03] or Thrift [SAK07]. The approaches discussed so far are intended for manual development of distributed applications. Several steps are automated, such as the generation of stubs for remote process invocation or parameter marshalling and unmarshalling. However, using these frameworks requires manual insertion of library calls and analysis of the data transfers between processes, usually by means of the remote function parameters.

Bertrand et al. [BBB⁺06] propose a data and control distribution for a particular problem called MxN, intended for manual development. They define a small set of *Parallel Remote Method Invocation* (PRMI) types, decomposing the data structures differently across processors depending on the type. Each type of invocation depends on the communication requirements of the decomposed application, and having multiple types allows them to choose more efficient ones when possible, allowing for more general communication styles with a performance handicap if using the efficient ones is not possible. Four challenges of PRMI are also presented, namely Delivery of arguments, Process participation, Concurrency and Parallel consistency. In our work, we handle the first one in the compiler, leaving the remaining three to the multi-threaded communication library.

Cooper et al. [CDD⁺10] extend the C language with an *offload* keyword that can be used to annotate functions for execution in different processors. This

work is focused in the Cell BE processor [PAB⁺05], which features two different types of cores working together, and is therefore a heterogeneous system. At compile time, sections to be migrated are identified, as well as other sections that must be implicitly migrated such as functions called from the offloaded functions themselves. The compiler also manages data movement between host and accelerator, and insertion of start-up and clear-down code.

Another relevant contribution in the field of programming languages is the X10 programming language [CGS⁺05] and its concept of Asynchronous PGAS (APGAS). Its main contribution is the concept of *activities* and the introduction of an `async` keyword to create them. *Activities* are asynchronous threads laid out in a thread hierarchy so that parent threads can wait on a child but not vice-versa. This avoids dead-locking of different threads waiting on each other to finish. Since the language is based on the concept of PGAS languages introduced in Section 1.2.4, data communications are inserted by the compiler as long as the programmer uses the PGAS language extensions to address data structures. This language still requires a considerable knowledge and effort from the programmer, but the concept of asynchronous threads is heavily used in the solutions proposed in this Chapter. The X10 language is addressed at homogeneous systems. Our proposal is more automated in the sense that no language constructs must be manually added to partition the specification, and adds data prefetching primitives to replace the semi-automated data communications offered by PGAS languages. Additionally, it can be retargeted to multiple architectures and heterogeneous systems.

3.2 High-level view of the compiler toolchain

The refactoring tool is part of a compiler built on top of LLVM [LA04], a well-known, production-ready compiler. The modified compiler is organised around a set of passes, which can be of two types: analysis or transform.

Figure 3.1 shows the complete compilation flow. The input to the compiler is a high-level specification composed of many modules. In C, a module comprises one source file plus included header files. Each module is processed by the front-end and translated into the LLVM intermediate representation (LLVM IR). Since our transformations operate on the IR, we can handle not only C but other procedural languages such as Fortran by using the existing front-ends to translate the original code.

An initial set of sequential optimizations is performed afterwards, so that the resulting code is tidier, simpler and more efficient than the original. All the processed modules are linked together in a single module that contains the complete sequential code apart from external libraries.

3.2. High-level view of the compiler toolchain

Figure 3.1: Proposed compilation toolchain.

The hardware architecture is given in a *Resource and Service Description* (RSD) file [BRV99]. This file defines the nodes, interconnect and memory structures of the target distributed system and is parsed into a BGL graph [SLL01]. The architecture graph also contains parameters such as the clock frequency of each processor, the bandwidth of each communication link, or the size of each memory space. The RSD can be extended to different processor types and parameters, which simplifies the development of estimation models for new processors or the improvement of existing ones. A sample RSD file is shown in Figure 3.2. There are three possible node types: “Comm”, “Memory” and “Device”. “Memory” and “Device” nodes can only have edges to “Comm” nodes. Communications are modeled as nodes instead of edges because the latter only allow point-to-point link modelling. The “Device” type models different processor subtypes. Figure 3.3 shows a graph representation of the system.

```

ROOTNODE compute{

  CONST NCPU = 2;

  NODE RAM{
    TYPE = "Memory";
    SIZE = 4;
    PORT to_FSB;
  };

  NODE FSB{
    TYPE = "Comm";
    SUBTYPE = "Bus";
    FREQ = 1.066;
    PORT to_RAM;
    for icpu = 0; icpu < NCPU; icpu = icpu + 1 do
      PORT to_CPU$(icpu);
    od
  };

  for icpu = 0; icpu < NCPU; icpu = icpu + 1 do

    NODE CPU_$(icpu){
      TYPE = "Device";
      SUBTYPE = "CPU";
      NCORE = 4;
      FREQ = 2.4;
      PORT to_FSB;
    };

    EDGE FSB_CPU_$(icpu){
      NODE CPU_$(icpu) PORT to_FSB
        <=> NODE FSB PORT to_CPU$(icpu);
    };
  od

  EDGE FSB_RAM{
    NODE RAM PORT to_FSB
      <=> NODE FSB PORT to_RAM;
  };
};

```

Figure 3.2: RSD description and graph representation of a machine with two quad-core processors, 2.4 GHz and 4 GB main memory.

3.2. High-level view of the compiler toolchain

Figure 3.3: Graph representing the architecture defined by the RSD file shown in Figure 3.2.

The linked module is input to the estimation stage, which also reads the hardware specification, and optional profiling information obtained by running data and functional profiling passes using the LLVM interpreter, `lli`. The profiling pass can collect runtime information, such as the times that a particular branch is taken or the data size, for particular training datasets. A general configurable profiler is available as part of the LLVM infrastructure.

An initial estimation of the execution time and other metrics, based on the architecture graph and the profile information, must be passed to the partitioner/mapper. During partitioning, movements of code blocks between processors are simulated at different granularity levels, updating the estimations. At the end, the partitioner assigns each code portion to a partition, and the mapper assigns each partition to a processor. This stage of the compiler is not currently implemented, but partitioning and mapping are two well-known codesign problems.

So far, the linked module remains unchanged. It is the refactoring stage that effectively creates one new module for each processor, distributes the code, and inserts additional instructions so that the modules can be executed together as a single application. This Chapter focuses on the refactoring stage of the compiler.

Finally, each output module is processed by its corresponding back-end, which outputs binary code for a specific processor. At present, back-ends exist for various processor architectures such as x86 or ARM. Also, experimental back-ends exist for various accelerators such as GPUs.

The following sections present the concepts and implementation details of a refactoring stage based on RBBs. At present, some stages of the abovementioned compilation flow are not fully automated. Hereafter, we assume that the mapping information is supplied to the refactoring stage.

3.3 Block outlining

The partitioning information is the input to the refactoring stage. At this point, each section of the code is marked for execution in one device. The code generator must refactor the code into one module per device, and the resulting modules must be able to link together and execute with the same results as the original code. The refactoring approach explained in the following paragraphs was introduced in [BCS⁺12].

The finest grain that our outliner can handle is the BB. In the control flow graph (CFG) of a function, each BB is represented by a vertex, and the directed edges between vertices represent control flow between BBs. Also, many control flow constructs exist, such as function calls, exceptions, loops, switches or gotos but, at the assembly level, control flow is always implemented by some kind of jump.

We want to turn BB-level control flow, as given by the CFG, into function-level control flow, represented by the function-call graph. Figure 3.4 shows the transformation steps. First, constant uses are tracked along the code, duplicating them in the module where user BBs are mapped. Next, global variables are distributed between the modules according to the partitioning information. Subsequently, each function is split up if necessary. Sections 3.3.1 and 3.3.2 elaborate on the splitting process. Finally, each module is written to a file, ready for back-end compilation.

- 1: duplicate constants
- 2: distribute globals
- {Process each function in the original code}
- 3: **for** every original function *f* **do**
- 4: *initiatorList* $\leftarrow$ find_initiators(*f*)
- 5: create_new_functions(*f*, *initiatorList*)
- 6: **end for**
- 7: write modules

Figure 3.4: Code generation steps

The aim of our refactoring algorithm is to make control transfers between multiple devices easy. The traditional view is to consider devices other than general-purpose CPUs as accelerators. In this way, CPUs can handle code sections with complex control flow, while the accelerators can focus on pure computation. In our approach, every device is equally important: the model of communication is more similar to peer-to-peer networks than to client-server ones. This approach makes sharing of computational resources easier in a heterogeneous platform, as each resource is just a processor of a certain type.

As a comparison between Offload [CDD⁺10] and our refactoring tool, if a section A of the target code is marked as suitable for acceleration, Offload moves it from the main program to a separate function. If another section B is called from A, Offload automatically moves it to the same accelerated function. This decision removes the necessity of managing complex control flow between CPU and accelerator, but also forces two sections to coexist in the same device even if each section would execute more efficiently on different devices. In contrast, our approach allows different mappings for A and B based on performance estimations. It achieves so by splitting up the original functions into several smaller functions, so that remote function calls can be requested between devices.

3.3.1 Initiator search

By definition, functions have a single entry-point. When a function is split up, each of the new functions has an entry BB that corresponds to a BB in the original function which may be the original entry point or not. We call these BBs the *initiators* of the original function.

New functions are initially generated by a control transfer between devices. In other words, if the partitioning results in a BB “a” mapped to device “A”, and this BB branches to another BB “b” mapped to device “B”, then “b” is an initiator. Initiators such as “b”, directly motivated by a control transfer between devices, are called *direct initiators*. An initiator may also be *indirect*, in case that it is required by previous initiators mapped to the same device.

An example is given in Figure 3.5, where the partitioning in the left shows bold and regular BBs; the former are assigned to processor A, while the latter are assigned to B. The center figure depicts four direct initiators (those with bold solid frames) and one indirect initiator (with a bold dotted frame). The entry block of the original function, **entry**, is always a direct initiator. Secondly, every BB with at least one predecessor in another device is also a direct initiator. In this example, **for.cond.preheader** and **for.cond18.loopexit** are direct initiators. These are different entry points into processor B, thus the need to refactor them into different functions. The dotted initiator **for.body20** is indirect: although its two predecessors are mapped to the same device, they have a different parent initiator, thus belonging to a different function. Hence, the dotted BB must result in a new function to honor the requirement of a single entry block per function.

Figure 3.6 shows the algorithm to find the initiators. This algorithm annotates the BBs so that the next stages of the code generation can assign the BBs to the new functions, as well as the required variables to be used inside. Also, it is possible to construct the function prototypes from the initiator information. The algorithm first looks for the direct initiators. Subsequently, a BFS is started from each of them, marking each basic block with the parent initiator.

Figure 3.5: CFG of a function at different stages of the refactoring. The figure on the left shows the device partitioning, where the BBs in the highlighted boxes are mapped to one device and the BBs in the regular boxes are mapped to another. The center figure shows the resulting initiators, and the right figure shows the newly-partitioned functions.

The algorithmic complexity is high. Finding direct initiators first requires processing all the BBs of a function, although the number of predecessors for each of them is typically small (<10). Also, finding indirect initiators requires one partial BFS traversal per initiator, ending at the next initiator, and one additional BFS traversal for each indirect initiator encountered. However, this is only a problem in the case of very deep functions with complex control flow and a fragmented partitioning that results in many initiators, which rarely happens in HPC codes. It is the task of the partitioner, not the code generator, to prevent such complex scenarios.

3.3.2 Updating the control structures

Once the initiators are found, the new functions must be generated accordingly. One function is created for each initiator and inserted into the module corresponding to the device where its BBs are mapped. Then, the BBs are moved from the original to the new function. Subsequently, the new function prototypes must be created: if any of the moved BBs requires data computed at a predecessor BB that is under a different initiator, this information must be added as a function parameter.

If a former branch crosses the boundaries of two functions, it must be turned into a function call. In order to be able to recover the parameter values that must

3.3. Block outlining

Ensure: *initList* is empty
 {Find direct initiators}
1: **for** all BBs in *f* **do**
2: *BB* $\leftarrow$ next BB in *f*
3: **for** all predecessors of *BB* **do**
4: *predecessor* $\leftarrow$ next predecessor of *BB*
5: **if** mapping(*BB*) $\neq$ mapping(*predecessor*)
 & *BB* not in *initList* **then**
6: insert *BB* into *initList*
7: break inner loop
8: **end if**
9: **end for**
10: **end for**
 {Entry block is always initiator but has no predecessors}
11: *entryBlk* $\leftarrow$ get_entry_block(*f*)
12: insert *entryBlk* into *initList*
 {Find indirect initiators}
13: *q* $\leftarrow$ *initList*
14: **while** *q* $\neq \emptyset$ **do**
15: *init* $\leftarrow$ pop(*q*)
16: root(*init*) $\leftarrow$ *init*
 {First BFS: start breadth-first search from the initiator.}
17: **for** all descendents of *init* in BFS order **do**
18: *child* $\leftarrow$ next_descendent(*init*)
19: **if** *child* is initiator **then**
20: break this BFS branch
21: **else if** root(*child*) = $\emptyset$ **then**
22: root(*child*) $\leftarrow$ *init*
23: **else** {another initiator previously visited *child*}
24: insert *child* into *initList*
25: push *child* into *q*
 {Old root must be revisited to update children}
26: push root(*child*) into *q*
 {Second BFS: erase root for all children of *child*}
27: **for** all descendents of *child* in BFS order **do**
28: *child2* $\leftarrow$ next_descendent of *child*
29: **if** *child2* is initiator **then**
30: break this BFS branch
31: **else**
32: root(*child2*) $\leftarrow \emptyset$
33: **end if**
34: **end for**
35: **end if**
36: **end for**
37: **end while**

Figure 3.6: The initiator search finds the entry blocks for future functions and marks BBs with its root initiator.

(a) The phi node in BB "2" is illegal, as there is no BB "entry" in the new **bb2()** function and neither are some of the variables used.

(b) A new entry block is created and the phi node is updated accordingly. Observe that the mapping "4"->c disappears in the Phi node.

Figure 3.7: Updating a phi node

be passed in a call to a new function, we maintain mappings from the original values to the newly created function parameters. Sometimes, at the start of a BB, there are special instructions called *Phi nodes* that select variables depending on the parent BB that jumped into the current BB. A new function may have outdated Phi nodes, where the possible values are now function parameters instead of values in previous BBs. In addition, Phi nodes in an entry block are illegal. This is shown in Figure 3.7.

We overcome these problems by inserting an empty BB as a new entry block, and updating the Phi nodes so that, if all of their values have been turned into one function parameter, the Phi node disappears and the assignment is done automatically. This is the case when all the BBs branching to the initiator are outside the new function. On the other hand, if there was a loop in the CFG, a BB may still branch to the initiator, and the Phi node is still needed to select the value between the one coming from the branching BB and the parameter.

The resulting Phi node will have one value for each possible BB that branches to the initiator from inside the new function, and a single value for all the branches from outside the function. Note that the purpose of a Phi node is to distinguish between values coming from different branches; when the new function is created, the branches from external BBs will be turned into a function call. Each external caller will call the new function with a different parameter value, hence the need for only one value in the Phi node to represent the parameter, instead of requiring one value for each external caller.

3.3.3 Data structures

Data partitioning is essential to obtain performance improvements in heterogeneous systems, all the more since the aim is to support distributed memory. For the sake of simplicity, we have assumed shared memory in this work in order to test the code generator for correctness.

Constants are copied to the modules where they are used. This is easy to implement by using the use-def chains provided by LLVM. For now, non-constant global variables are mapped to the initial device but, since the modules are linked together after partitioning (see Section 3.2), they can be made visible to other modules by marking them as external (similarly to the `extern` keyword in C). Internal data structures are passed by means of pointers as function parameters.

Our goal is to automatically partition data structures, so we can assign portions of them to modules at compile time. Previous works such as [BSY⁺10] use centralized arbiters for data management, but we consider these solutions to be poorly scalable to distributed memory machines. Another problem is that of pointers with distributed memory, where memory spaces are different on each device and thus pointer addresses are not preserved across the execution environment. Cooper et al. [CDD⁺10] further explain the problem and introduce the concept of *outer pointers* to automatically manage data transfers between devices. Automated data partitioning techniques for distributed systems are discussed in Chapter 4.

3.4 Remote Block Branches (RBBs)

The previously discussed outlining mechanism splits functions and places them in separate modules depending on the target processor. In this Section, we introduce the transformations and the runtime support required to make control flow between these functions behave in a similar manner to local control flow. In the same way that RPCs are the distributed equivalent to functions, we introduce a distributed equivalent to BBs that we call RBBs.

The original RPCs implement a client-server communication mechanism, where the client must wait for the return of the server. Newer implementations such as Seymour [SNM⁺02] or Tanaka [TNS⁺03] allow for asynchronous communications so that the client may keep working while waiting for the server to finish. However, at some point, the client must sync with the server in order to receive its return value and check that the server function has finished. This limitation does not meet the requirements of BB control flow, since the server should not return control but pass it to another superblock. Since the behaviour of the functions generated by outlining is different than that of regular functions in terms of control flow, we refer to them as superblocks.

RBBs follow a peer-to-peer approach where the superblocks control the program for a limited amount of time. All the information is passed between the superblocks by means of either the RBB parameters or a parallel data flow system. The first is used to transfer virtual registers of the LLVM IR, which is in SSA form. Since a virtual register is only assigned a value once, all the parameters of a RBB are read-only and may be passed by value.

In the case of references such as dynamically allocated memory or arrays, these cannot be passed by value (this is, the pointer value) since the memory spaces associated to different processes are not equivalent. We use prefetching techniques instead, where a superblock that produces data passes it to the superblocks that consume that data. Prefetching primitives also serve as synchronization points between producer and consumer superblocks. At present, prefetching must be inserted by the programmer, but it is our goal to automate it in the future with the help of pointer analysis, which is quickly evolving in the LLVM community. Chapter 4 discusses a more specific approach for the subset of programs discussed in Chapter 2.

When a basic block passes control to a remote superblock, it issues a RBB call and passes the virtual registers required by the following blocks as parameters. A key contribution of RBBs is that the calls are purely non-blocking: right after the RBB is issued, the call returns. No further synchronization is required: no `RBB_Wait()`, `RBB_Probe()`, etc. It is possible to implement such a system with standard RPCs; for instance, the Linux glibc (based on the Open Network Computing RPC (ONC-RPC) and consequently on [BN84]) allows the user to set timeouts in the client. If the timeout is zero, the call is nonblocking. Bloomer [Blo92] refers to this technique as “support for one-way batch and broadcast”. However, the timeouts must be set server-wide, rather than locally for each remote function, and there is no tool for full automation of distributed programs that combines RPCs with compiler transformation passes. In addition, the performance is lower than VMRPC [CSS⁺13], and subsequently at least an order of magnitude lower than the RBB implementation proposed in this Thesis as per the results shown in Section 3.5.

3.4. Remote Block Branches (RBBs)

(a) Initial code

(b) Outlined code with RBBs

Figure 3.8: Transformation of branch instructions into a distributed application with Remote Block Branches.

Figure 3.8 shows a program with three basic blocks that have been outlined and where RBBs have been inserted. When a RBB is issued, the caller specifies the identifier of the superblock to be activated and a list of the parameters corresponding to the virtual registers to be transferred. For example, the value of register `a` is handed over to `bb2{}` because it is used inside block 2, and is also passed to `bb3{}` because successive calls to `bb2{}` must receive its value again. In this example, all superblocks die after issuing a RBB because this is the last instruction. For the rest of the paper, we use curly braces to name superblocks similarly to the parentheses used in function names (e.g. `a{}` is a superblock, while `a()` is a function). If no parentheses or curly braces are appended to the name, the element is a plain basic block.

Deep CFGs often have long-lived virtual registers having to be passed along. LLVM maintains the list of all the uses of every register in structures called *use-def* and *def-use* chains. Some optimization passes analyze these chains (what is called a reaching definitions analysis) and move assignments of virtual registers close to their uses, thus reducing its life and therefore reducing the number of parameters of the RBBs. However, Section 3.5 demonstrates that the performance of the current RBB implementation does not degrade significantly when the number of parameters increases. When a superblock is callable both locally and remotely, the code can be replicated so that the local version is accessed using the traditional branching mechanism and the remote version is accessed by means of RBBs. This avoids the overheads introduced by the RBB implementation when invoking local code.

Chang et al. [CCE99] identify five primary issues when designing RPCs. The first is method name resolution. RPCs require costly distributed databases to register methods, and lookup tables to hide the latency of name resolution. Since, in the case of our compiler, we know the possible calling sites at compile time, the compiler is able to fill in static lookup tables at each processor's module. These tables allow for name resolution with very little overhead.

The second issue is argument marshalling. Our refactoring stage inserts all the parameter marshalling by analyzing the prototypes of functions and superblocks. In contrast to dynamic methods used in the typical RPC distributions, this approach results in lower runtime overhead with only a slight increase in the executable size.

Data transfer is the third issue. RPCs receive all the required data by means of their parameters. OpenCL and CUDA, which use evolved remote function mechanisms, also allow data prefetching that must be inserted by the application programmer. This is solved in part with the transfer of virtual registers as the RBB parameters. Additionally, the communication API has prefetching primitives that allow handling dynamic memory and static data.

Control transfer, the fourth issue, is the center of the RBB implementation

itself, non-blocking and distributed. The last issue is message reception and dispatch. Traditional RPCs implement message transfer by means of polling threads or interrupts. The current RBB implementation defers message handling to the lower communication layer, currently MPI [The12]. Fault tolerance and security also depend on MPI.

The RBB implementation is divided in two components: the compiler and the communications library. In order to maintain high performance and compiler automation, we place most of the implementation inside the compiler, this is, static decisions, instead of the runtime system. As a side effect, this also results in LLVM IR platform-independent code. Then, each module is linked against a platform-specific communications library before back-end compilation.

3.4.1 Compiler passes

The refactoring stage is divided into three LLVM passes called RefactoringPass, CommMarkerPass and PartitionWriterPass. These are aided by two helper classes, PartitionInfo and CommunicationInfo, which store the temporary results between passes.

RefactoringPass does superblock outlining according to the mappings supplied as an input. First, it finds the initiators, as explained in Section 3.3. Then, it creates the superblocks from these and transforms the old branches into explicit superblock calls. Afterwards, since the LLVM IR is an SSA language, the previous steps result in outdated Phi nodes that are simplified or fixed appropriately. Finally, global variables are distributed and constants are copied to all the user modules. The output of this stage is the original code partitioned into a number of LLVM modules.

CommMarkerPass inserts the calls to the communication API. Specifically, startup and shutdown calls are inserted at the main function of each module. RBBs are inserted where each caller requests a remote function or passes control to a remote superblock. The compiler detects whether a return is required (this is, when the called superblock corresponds to the entry point of a function in the original code) and inserts return sends and receives. For each remote calling site, the pass creates a client stub with parameter marshalling code. Similarly, for each server function corresponding to a superblock, a server stub is generated with un-marshalling code. As the stubs are being generated, pointers to them are inserted into a static table at the server module, and the compiler identifies them with a number together with the process ID of the owning process. This allows for a very lightweight method resolution. Other approaches require a distributed database for registration and method lookup; here, the problem is solved at compile time. CommMarkerPass also inserts the program terminator, a unique program exit block, and redirects all previous exit points to it. Although LLVM does not re-

quire to have a unique exit block for the program, the RBB does require this in order to be able to orchestrate a clean shutdown. `sendShutdownACK()` is inserted into this unique block.

Function prototypes, server and client stubs, parameter marshalling and insertion of communication primitives are all done automatically by the compiler. This is achieved by traversing the use-def and def-use chains offered by the LLVM IR, allowing the refactoring pass to analyze the access patterns to virtual registers and elicit the required data transfers between superblocks. Other popular libraries and frameworks require, at least, to specify the prototypes of the functions, either by means of a protocol compiler such as `rpcgen` (e.g. ONC-RPC) or by manually outlining portions of code (e.g. CUDA, OpenCL). At present, the RBB system requires prefetching calls to be inserted by hand.

Finally, `PartitionWriterPass` instantiates a standard LLVM `PrintModulePass` for each module and writes out the code in LLVM IR format.

3.4.2 The runtime library

The communication API is a small set of functions inserted by the compiler into the target code. Each target processor architecture must have its own implementation of the API.

In order to give support for functional partitioning in multiple processes, the communication system must support Multiple Program, Multiple Data (MPMD) execution. Many of the potentially suitable communication standards, such as Active Messages [vECGS92] or Gasnet [Bon02] do not support MPMD. MPI, on the contrary, supports MPMD, has implementations for most of the existing architectures and is sufficiently high-level so that programmability is not an issue. Building on top of lower-level interfaces such as Infiniband [Pen02] is more efficient yet less portable. We have tested the communication library with OpenMPI [Ope] and MPICH [MPI] in a multicore system, and with MVAPICH [MVA] in a 16-core Infiniband cluster.

The implementation is intended to run in conditions where each process owns its processor completely. The processors run stripped-down kernels where multitasking is minimal and there is almost no contention for local resources, which is the typical case of a HPC cluster. The reason for this is performance; the generated executables will run correctly in other scenarios, but the performance will suffer due to the multiple threads required by the RBB runtime system. The implementation runs several threads inside each process as follows:

- Main thread: This corresponds to the entry point of each process. Among all the processes running in different processors, only one serves as the real

3.4. Remote Block Branches (RBBs)

entry point of the program. The other processes only call the main thread for startup and shutdown purposes.

- **Listener:** This thread is forked when the main thread calls the startup function. It listens for a termination signal, RBB requests and data prefetch messages coming from other processes. When a RBB request arrives, it receives the parameters and invokes the corresponding handler.
- **RBB handlers:** Each handler manages a superblock. A handler starts with a server stub that unmarshalls the parameters, then calls the local superblock. Handlers are invoked from the listener thread with the help of the compiler-generated static tables for name resolution.

OpenMPI, MPICH and MVAPICH all have full multithreading support. We use the Pthreads library [BFN13] and compile the MPI distributions with thread support. We use Pthreads due to its flexibility and portability compared to other multithreading environments such as OpenMP [CJdP08] or Intel Threading Building Blocks [Rei10].

The standard communication API contains the following minimal functionality:

- **startup:** Initializes the runtime system.
- **shutdown:** Allows clean shutdown of the application and auxiliary runtime systems.
- **sendShutdownACK:** Triggers shutdown mechanism at the unique exit block.
- **RBB:** Request for Remote Block Branch.
- **sendReturn:** Send function return to a remote caller. This is inserted at the superblocks that correspond to exit points of the original functions.
- **recvReturn:** Wait for the return of a remote function. This is inserted at some point after a remote call has been made to a superblock that corresponds to the entry point of any of the original functions.

In addition to the RBB calls, we add support for data prefetching with `prefetch`, `getPrefetch`, `freePrefetch` and a higher level `prefetch2DArray`, the latter of which allows prefetching parts of a 2D array. The combination of these prefetching functions allows data producer and consumer relationships between the superblocks: `prefetch` and `getPrefetch` are used for data transfer between the producer and the consumer and operate in a non-blocking manner,

Figure 3.9: Startup, RBB call and shutdown process. A closed lock appears when a thread is put into sleep. An open lock represents the wakeup condition.

while `freePrefetch` signals the time when a received prefetch will not be used anymore. The latter can be used as a data-driven synchronization mechanism, since it manipulates a series of bits that validate or invalidate the reception of new prefetches.

The resulting application is multi-process and multithreaded, and executes seamlessly in a distributed environment. Figure 3.9 illustrates the execution model for the case of an application partitioned into two processes. All processes call `startup` to initialize MPI. A private communicator is then created, where each process is assigned a new rank as its process ID. The ranks are preset by the compiler as a parameter to the `startup` calls when these are inserted. Finally, the listener thread is forked. The main thread must be able to wait on the listener's finish (a *joinable* thread, in Pthreads jargon, as opposed to a *detached* thread).

The main thread of the entry process continues executing the application. At some point, a RBB will be issued to another process. The listener of the second process captures the request, looks for the function pointer of the requested superblock in the constant function table and calls it.

Only one of the superblocks in the program results in the end of the program. Before exiting, this superblock calls `sendShutdownACK`, which sends a signal to the listener of its own process. Once this listener exits the listening loop, it sends an MPI shutdown message to all other listeners and terminates. The termination signal implies that no other superblock is being executed. At some point, the main thread of each process calls `shutdown`. This function waits until the listener thread finishes, then shuts down MPI, tidies up all other structures, and terminates the process.

Using MPI together with Pthreads requires a careful analysis of the communication patterns between threads inside the same process. We have used

ParaProf and TAU [SM06] to profile and trace the tests and find hot spots. The ideal situation in the RBB system would be that the listener in a process is only run when there are no pending handlers (colloquially speaking, when there is no useful work). Isolated scheduling within a process is possible according to the Pthreads standard by starting the threads with `PTHREAD_SCOPE_PROCESS`. Unfortunately, Linux does not support this part of the standard, thus throwing a *not supported* error if attempted. Another possibility would be to use the default `PTHREAD_SCOPE_SYSTEM` and give higher priorities to the handlers, but this affects the general OS scheduling and can lead to starvation in kernel threads. For that reason, Linux only allows changing the thread priorities with root access, which is unreasonable in a shared HPC cluster, as it would introduce critical security issues.

For that reason, we have implemented a custom thread scheduling system embedded in the communication library. By using Pthreads condition variables, the RBB handlers always have a higher priority than the listener, except when they must wait for some events. This improves the performance of the library by more than an order of magnitude. Other operating systems, such as FreeBSD, fully support Pthreads scheduling, in particular `PTHREAD_SCOPE_PROCESS`, and therefore should not require the custom scheduler, although we have not tested this.

3.5 Results

The experiments with refactoring and the RBB system are split into three subsections. Subsection 3.5.1 tests the refactoring algorithm by using vectorization on a multicore system. At this point, no RBBs are used. Subsection 3.5.2 performs an initial evaluation of the RBB system with a set of synthetic benchmarks based on typical control flow constructs. This Section contains a general evaluation of the whole system, as well as a more detailed analysis of the communication library. Finally, Section 3.5.3 evaluates the system with a matrix multiplication kernel that is typically used as HPC benchmark.

3.5.1 Results of refactoring with vectorization on a multicore

The correctness of the code generation has been checked with various test codes, both control- and data-intensive. The transformations required to insert function calls result in a code whose behavior is identical to the original in all the tested benchmarks. On the other hand, using the LLVM tools to measure execution time on passes, the result is that between 40% and 57% of the compilation time,

not including profiling, is spent on code generation, being the absolute numbers between 5 and 30 milliseconds depending on the benchmarks. These are reasonable numbers; the aim of the heterogeneous compilation tool is to reduce the development cycle in terms of months.

This subsection does not attempt to prove the feasibility of speeding up the application; this is achieved by using the autovectorization capabilities of *gcc/-dragonegg*. In order to investigate the overhead introduced by the extra function calls, the toolchain has been tested in the following scenario. Assume that the target architecture contains two processors, one of them with support for vector execution and another one with a faster memory access but no vector capabilities. Slowing down the memory access of the vector processor is necessary to force the partitioner assign blocks to both devices; otherwise, all the code ends up in the SIMD device. For the purpose of this experiment, we have developed an estimator that feeds the partitioner with the partitioning decisions. As the real architecture does not have different memory features, the estimator is not accurate, but serves the purpose of testing the code generator. In other words, we are simulating a heterogeneous architecture by using a homogeneous one and modeling parts of it differently.

The first stage is to link all the source files together into a single IR file. This file is then passed through *dragonegg*, and a vectorized IR file is obtained. After vectorization, the partitioner processes the vectorized IR and outputs two LLVM modules, labeled “CPU.part” and “CPU_SIMD.part”. Although the whole IR has been vectorized by *dragonegg*, it is possible to select which modules to vectorize by using the regular x86 back-end or a SIMD-enabled one. Since all the used back-ends generate code for 64-bit x86 machines, the resulting assembled modules can be linked together into a single executable, where only some sections have been vectorized. There is no automated insertion of communications at this point, hence it is not possible to generate one executable per device. Also note that, in this experiment, communication between outlined functions is free apart from the function call overhead, since there is only one processor executing both modules.

The results of the experiment with four benchmarks are shown in Figure 3.10. The first benchmark, labeled *ctrl*, is a test program with heavy-control and data-intensive sections. *arrayops* executes a series of array operations similar to those encountered in linear algebra. The remaining benchmarks are computational kernels extracted from PolyBench [Pou]: *adi* is a stencil computations code, and *floyd-warshall* finds shortest paths in a weighted graph. Each of them has been compiled with five configurations. These are the original code (1), the original code vectorized by *dragonegg* (2), and the partitioned code in three variants: non-vectorized (3), fully vectorized (4), and partially vectorized (5), where only “CPU_SIMD.part” is vectorized. All the partitioned variants use the same estimator, but they are compiled with different back-ends as explained above. The

Figure 3.10: Performance of the partitioned codes with different combinations of partitioning and vectorization.

measurements are taken from a round of 15 executions for each case, and taking the median in order to minimize unpredictability.

The benchmarks experience a variety of speedups between 1.08 and 1.9 by the effect of vectorization. Also, two of the benchmarks improve execution time by 5% between the original (1) and the non-vectorized partitioned code (3). The same applies to the vectorized original (2) and the vectorized partitioned code (4). This implies that the generated code is more efficient, even when we initially suspected that the function calls would add some overhead. An in-depth examination of the generated assembly code unveiled that the back-end optimizations benefit from Phi node simplification and the addition of BBs, which together give more margin for instruction scheduling. In the case of *arrayops*, the code refactoring allowed the LLVM hoisting pass to move four instructions away from a loop with a high loop count, resulting in millions of instructions saved at run time.

In the case of the *floyd-warshall* benchmark, partial vectorization results (5) indicate that the partitioner does not map all the vectorizable code to the SIMD processor: it performs as bad as the non-vectorized code (3). This is partly due to the poor estimation provided.

3.5.2 RBBs on synthetic benchmarks

We have presented an automated distributed control flow compiler based on RBBs with two components: a set of compiler passes and a communication runtime

library. This Section evaluates the performance of both sides. First, we measure the improvement of the RBB refactoring as compared to a traditional RPC-style refactoring. Next, we compare the performance of the RBB communication library against the MVAPICH library in order to find the overhead of our RBB implementation.

The platform used for benchmarking is a three-node cluster managed by a Rocks Mamba 6.0 cluster operating system [PKB03]. One node acts as the front-end for interactive work, and the remaining two nodes make up the compute back-end. Each back-end node has two quad-core Intel Xeon processors at 1.86 GHz with access to 32 GB of shared memory. The available interconnects are 1GbE connecting both nodes to the front-end of the cluster and one 40 Gb Infiniband network private to the back-end. The computing power adds up to 64 GB of distributed memory and 16 computing cores.

Overall performance of the RBB system

Figures 3.11, 3.12 and 3.13 show the CFG and the timeline of the three synthetic benchmarks used for evaluation (seq, nested, improper). Each of the figures contains a first subfigure that shows the original CFG; the next subfigures represent the timeline, where the execution starts at the top and ends at the bottom. Although these benchmarks have been coded by hand with the purpose of benchmarking the RBBs, they are representative of the control flow structures present in common HPC applications, and they are considered the minimal building blocks of control flow [Muc97]).

Figure 3.11: **seq** benchmark: three basic blocks in sequence.

(a) Original CFG

(b) RPC style and primitives

(c) RBB style, RPC primitives

(d) Full RBB style and primitives

Figure 3.12: **nested** benchmark: two nested loops.

Figure 3.13: **improper** benchmark: an improper region consisting in a single loop with two possible entry and exit points.

Each benchmark has been distributed with three different techniques. The first is a pure master-slave RPC approach, where the master process assumes the control of the application and calls other processes remotely. This is widely used in systems with hardware accelerators, where the host processor assumes control flow management and requests slave functions to the accelerators.

The second is a hybrid technique where the CFGs are rearranged in a similar way to our RBB refactoring pass. However, the communications are still implemented via RPC. The results of these implementations give an idea of the improvement of the peer-to-peer approach against a master-slave one. Note in Figure 3.13 that improper regions must be removed by replicating blocks 2 and 3. This is a standard procedure explained in [Muc97]. The `if...else` constructs in the Figure select one of the two overlapped functions (e.g. `f4()` and `f5()` in Figure 3.13c).

The last technique is a pure peer-to-peer, non-blocking RBB implementation, obtained from C source code compilation using the LLVM compiler and RBB passes, and linking against the RBB communication library. As opposed to the first two techniques, where the benchmarks have been manually implemented, the benchmarks implemented with the full RBB approach are generated by the compiler.

Table 3.1 gives the timings of these benchmarks for the case of intra-node communications, where all the processes are mapped to the same compute node of the target architecture, and the case where communications are conducted over a system-wide network. An additional benchmark (`seq + vregs`) is included, where the first benchmark is extended so that data from virtual registers is communicated between superblocks, resulting in 256 bytes of parameters to be marshalled and transmitted with each RBB. For each experiment, five measures have been taken and the median used as the final number reported.

The intra-node hybrid RPC-RBB versions of the benchmarks save, respectively, 3%, 9%, 51% and 1% of the RPC execution time. The 9% improvement of the sequential program with virtual register transfers is motivated by the fact that, sometimes, the registers have to be transferred deep down the CFG path. In the case of a pure master-slave approach, they are transferred at each RPC return to the master and again to each slave in case that the next remote superblock (or remote function) requires them. In the case of a hybrid approach, all the RPCs corresponding to remote superblocks have empty returns, unlike remote functions. This saves communication costs.

The 51% improvement of the nested benchmark is explained by the extra RPC required at each inner iteration of the master-slave program. As the basic blocks 2 and 3 are located in the same partition, the peer-to-peer approach simulates the control flow making the call to `f5()` a tail call, so that the return of `f3()` results in the program arriving at the `for` loop located in `f2()`. Since `f3()` is called

	Intra-node			Over the network		
	RPC	RPC-RBB	RBB	RPC	RPC-RBB	RBB
seq	0.844	0.821	0.041	1.642	1.573	0.126
seq + vregs	0.917	0.839	0.052	1.837	1.633	0.188
nested	0.835	0.411	0.041	1.536	0.689	0.132
improper	0.420	0.416	0.057	0.759	0.727	0.142

Table 3.1: Benchmark timings (in seconds) with three implementation choices and different locality of communications.

intensively and does not have to be called in the hybrid version, the performance is greatly improved. These results prove that a peer-to-peer approach is generally not slower than a master-slave one and, in some cases, it is far more efficient.

The extra speedups of the full RBB versions range between 7x and 20x. Apart from peer-to-peer control flow, the MPI-based RBB library takes advantage of the QDR Infiniband available and an implementation with very low overhead on top of MPI. This results in lower latency compared to the 1GbE network used by the glibc RPC mechanism. The results of the benchmarks distributed over different nodes are obviously handicapped by the network performance, but result in similar conclusions (speedups between 5x and 13x).

Detailed performance of the communication library

The communication library has been implemented over MPI due to its unparalleled advantages in terms of performance, portability, flexibility (at the cost of ease of use) and variety of implementations. This comparative study seeks to demonstrate the scalability of the communication library with the size of the system.

Figure 3.14 compares the startup and shutdown time of the RBB system against `MPI_Init()` and `MPI_Shutdown()`. The overhead in the first 2 to 8 processes is close to zero. The step between 8 and 9 processes is due to the use of two nodes (with 8 cores per node), among which a network must be used that is slower than the shared memory. Most of the extra activities required by the RBB system, such as creating the extra communicator and starting up the listener threads, do not incur significant overhead. However, from 9 processes onwards, the RBB system starts deviating from MVAPICH. This is because, at shutdown, the listener thread of the terminator process must send one termination message to the listeners of all other processes. This broadcasting behaviour could be improved by efficiently implementing the notifications in a tree (although this might already be part of the MPI implementation of `MPI_Bcast()`), or clustered by groups of processes residing in a single node. This has not been yet imple-

3.5. Results

mented, but an in-depth analysis should be done in case the approach is used in machines with high processor counts.

Figure 3.14: Combined startup and shutdown time of the RBB system compared to the MVAPICH MPI distribution.

Figure 3.15 shows the single RBB execution time with CFGs of different depth. We compare against two synchronous MPI calls, instead of one, because each RBB is currently implemented with two MPI messages. The results show that the extra activities required by an RBB, including memory allocation for the parameters, marshalling / unmarshalling and lookup of the requested superblock, are negligible compared to the cost of the raw MPI communications involved.

Figure 3.15: Performance of chained Remote Block Branches as generated by a loop.

Figure 3.15 runs the same experiment than that reported in the original VM-RPC paper [CSS⁺13]. The latency obtained with our RBB implementation for

Figure 3.16: RBB performance variation with different parameter sizes.

an intra-node null RPC is $2.137 \mu\text{s}$. This is a speedup of 7x over the VMRPC measures. Also, a RBB with a 256-byte message takes $2.470 \mu\text{s}$. We can apply a simple data transfer model: $t = L + S/B$, where t is the message transfer time, L is the latency, S is the message size and B is the bandwidth in MB/s. From this model and the measures for a 100-byte RBB and a null RBB, we obtain the maximum throughput, which is 769 MB/s. The best results of VMRPC reach 70 MB/s, this is, RBBs obtain a speedup above 10x. In VMRPC, several software layers increase the communication overhead.

Finally, Figure 3.16 shows the variation of RBB performance related to the variation of the RBB parameters, this is, the size of virtual registers to be transferred. Up to 250 bytes of parameters, the performance of the RBB does not degrade significantly. It is not usual that real CFGs propagate the use of their virtual registers deep along the basic block chain. Therefore, the parameter list of the superblocks is likely to remain low enough so that the cost of an RBB is affordable. The scalability of the implemented communication library is bound to the scalability of MPI, which is known to scale reasonably well to thousands of processors.

3.5.3 2-dimensional matrix multiplication

Together with data prefetching, the RBB system is able to implement functional and data parallelism in distributed machines. By calling RBBs concurrently, many superblocks can work in parallel if there are no data dependencies. If dependencies are present, the prefetching mechanism automatically synchronizes the threads based on the data flow. Also, data parallelism is easy to generate by replicating superblocks and distributing data among them with prefetching primitives.

This section applies RBBs to the 2-dimensional matrix multiplication kernel shown in Figure 3.17, which is part of the PolyBench [Pou] benchmark suite. The kernel calculates the formula $D = \alpha \times A \times B \times C + \beta \times D$, where we define A, B, C and D as square matrices to simplify the analysis of the results. In order to analyze performance depending on the computational load, we have added the loop in line 5, which performs the calculation a parameterized number of times `WL_MULTIPLIER`.

```

1 for (i = 0; i < ni; i++){
2   for (j = 0; j < nj; j++){

3     tmp[i][j] = 0;

4     for (k = 0; k < nk; ++k){
5       for (wl = 0; wl < WL_MULTIPLIER; wl++){
6         wlTmp = alpha * A[i][k] * B[k][j];

7         tmp[i][j] += wlTmp;
8       }
9     }

10    D[i][j] *= beta;

11    for (k = 0; k < nj; ++k)
12      D[i][j] += tmp[i][k] * C[k][j];
13  }
14 }

```

Figure 3.17: 2mm kernel in C.

The kernel has two main loops (line 1 and 8) with subsequent nested loops. The compiler has been instructed to map the first loop into one processor, and the second into another processor. This decision generates two remote RBB calls: one from the main function to the first loop, and another from the end of the first loop to the header of the second one. RBBs allow automatic start of both superblocks almost at the same time. Since the `tmp[][]` array imposes data dependencies, prefetching primitives have been added manually. The result is that the second superblock consumes the values of `tmp` that the first superblock produces in a pipelined manner. This is only one of many types of parallelism

that can be exploited with RBBs. Thus, an important question is whether the RBB system with prefetching is able to hide the extra communication costs in the distributed program.

Order	Original (s)	Partitioned (s)	Speedup (%)
10	4.6×10^{-5}	6.0×10^{-3}	0.08
100	0.018	0.052	0.33
1000	44.061	58.706	0.75

Table 3.2: Timings of the 2mm original and partitioned code with different data sizes, and calculated overhead.

The results in Table 3.2 show the wallclock times of the original (sequential) and partitioned programs for different data sizes, and the resulting speedup of the latter against the former. Speedups below 1 represent an overhead. Although the three data sizes result in a slowdown, this is because we have introduced no data parallelism, this is, we are measuring the overhead of the RBB system. These results are due to the fact that the analyzed kernel has a very low computational load compared to the data to be transferred between superblocks. However, it is clear that a higher data payload results in the RBB/prefetch system being more efficient, this is, hiding the transfer latencies more effectively.

Figure 3.18 shows the speedup of the partitioned and parallelized program against the original code as a function of the workload multiplier. Since the 2mm kernel has a very low computational load as is, the experiments with the workload multiplier set to 1 are dominated by the communications, leading to slowdowns. For the small dataset, it is necessary to raise the multiplier to 2000 for a speedup of 1 (no slowdown), while the bigger datasets only require a factor between 10 and 20 to achieve the same result. This is, the prefetching and parallelization in this example counteracts its communication cost for large datasets and a kernel with enough computational load. This is usually the case, for example, in high performance simulation codes, where taking advantage of Gustafson’s law is critical.

While the lines for the 10×10 and 100×100 matrices have a similar shape, the 1000×1000 matrix shows a more gradual slope. This is due to the fact that the datasets in the first two cases fit in the 2048 KB L2 cache (the 100×100 matrices add up to 400 MB), whereas the bigger case takes 40 MB of main memory space. This closes the gap between communication and memory access times, thus reducing the differences between the original code, only with memory access, and the parallelized and partitioned one, which performs both memory access and communications.

Figure 3.18: Performance of the 2mm kernel with RBBs

3.6 Conclusions

This Chapter has introduced a refactoring stage within a compiler toolchain to improve automated partitioning in heterogeneous HPC environments. In particular, we present a novel code generation technique based on function outlining. The approach is general enough to suit a wide variety of devices such as GPUs or FPGAs, provided that an appropriate LLVM backend is available. In some cases, the compiler generates more efficient code due to outlining small blocks of code, counteracting the negative effects of the increase in function calls.

We combine these LLVM analysis and transformation passes over the CFG with a high-performance communication library. The current implementation of the communication library is based on MPI, but other implementations should be possible. Regardless of the speeds obtained by Infiniband, building on top of MPI also ensures portability, base performance and scalability. The resulting applications are distributed over the target architecture from the partitioning information.

The RBB concept borrows some of its ideas from RPCs, a long-time standard. However, the results obtained indicate that RBBs are more efficient for automated generation of distributed applications based on a compiler. While virtually all existing work targets shared-memory systems, our work proposes a general solution for distributed memory, effectively hiding the typical latency of interprocess communication when the data size and kernel computational intensity are above a certain size.

3.6.1 Future work

We have focused in the code generation and therefore the toolchain is not complete. In order to create a fully-working toolchain, integrating the partitioning

and RBB approach with a functional partitioning step is the major missing step. Many of such algorithms have already been published in the field of codesign in embedded systems.

RBBs are a low-level concept upon which many types of parallelism can be built. Implementing a streaming and pipelining approach on top of RBBs would show the benefits for systems based on FPGAs.

3.6. *Conclusions*

Chapter 4

Automated pipelining with the inspector-executor paradigm

The previous Chapter analyzed control transfer between different parts of an application that were assigned to different processors. A refactoring algorithm and a communication library were introduced so that communications for distributed control flow could be automatically inserted by a parallelizing compiler. This Chapter investigates the challenge of automated insertion of communications, this time from a data transfer standpoint. The following Sections investigate how to split and communicate the data between processes that run different parts of the code.

HPC systems with distributed memory typically perform communications between their processors by means of *message passing*, which is usually implemented in parallel applications by means of a communication library. In order to parallelize an existing application, calls to the library must be manually inserted by the developer, resulting in a tedious refactoring process. Sometimes, even a complete redesign of the application is required. No parallelism pattern is suitable for all applications, and in the past years there have been numerous efforts to classify forms of parallelism. One of the most famous attempts is the work by Asanovic et al. [ADK⁺09], sometimes known as the *Berkeley dwarfs* or *motifs*. In this work, a study of classic scientific and industrial applications is conducted, resulting in classifications according to the data structures used and the nature of the problem to be solved. Other sub-classifications have appeared, such as the *Algorithmic Species* by Nugteren et al. [NCC13], which categorize codes with affine loops according to the polyhedral model.

Automated extraction of parallelism is a complex problem and many solutions have been attempted in the past. A general solution might even require implementing artificial intelligence techniques in the parallelization tools, in order to identify parallel patterns in the target code. However, as usual, constraining the

```

1: {Time iteration loop}
2: while not end_condition() do
3:   {Loop traversing a mesh by vertices}
4:   for firstVertex to lastVertex do
5:     calculate1()
6:   end for
7:   {Loop traversing a mesh by edges}
8:   for firstEdge to lastEdge do
9:     calculate2()
10:  end for
11:  {Other loops traversing the mesh}
12:  ...
13: end while

```

Figure 4.1: Pseudocode structure of the codes that can be considered for parallelization.

scope to a small category of codes allows to apply more specific techniques. For example, polyhedral tools such as Polly [GGL12] or Graphite [PCB⁺06] are capable of automatically finding a suitable loop schedule for parallelization, where different iterations of the same loop can run concurrently. In this case, the approach requires that the application contains affine loops, and other parts of the code are left unoptimized.

We restrict the work in this Chapter to codes using irregular data structures that implement algorithms as shown in Figure 4.1. A relevant example of such codes is the CFD code analyzed in Chapter 2. The layout of these algorithms are as follows:

- An outer or “main” loop iterates until some condition is met.
- Several inner loops iterate over the data structure of interest.

The described structure is very common in simulations dealing with unstructured meshes and, generally, in many other scientific simulations. In the simplest case, the condition of the main loop is a count for a certain number of iterations. In the case of physical simulations such as CFD codes, however, the exit condition is usually a function that calculates whether the simulation has reached a stable state. The outer loop is also colloquially referred to as the *time loop*, and each iteration is referred to as a *timestep*. Note that this does not necessarily represent time physically, but rather the evolution of the simulation along a timeline. In the case of CFD for aeronautics, the data structure traversed by the inner loops is the unstructured mesh that represents the fluid surrounding an airfoil, such as a wing or the fuselage of an airplane.

A classic approach to parallelizing the code above is to first partition the data structures with a partitioning software tool such as METIS [LPS⁺15][KK98] or Zoltan [CBD⁺07]. The code is then replicated in several processes, and each data partition is assigned to one of such processes. Depending on the algorithm of interest, the boundaries between partitions may introduce communication requirements, for example from a process that writes to data in the boundaries to processes that only read that data. Another option is to assign the inner loops to different processes so that they can operate in a pipelined fashion.

While data partitioning is very common in HPC, pipelining different loops into several processors has been seldom investigated. Being able to assign different parts of the code to different processes is of vital importance in heterogeneous systems, especially for those with FPGAs: in many cases, the area constraints in these accelerators make impossible to map the whole computational kernel into one of them. Also, the relatively low memory performance of these devices implies that the memory-intensive loops may perform better in a general-purpose processor.

This Chapter analyzes the problem of automatically assigning the loops to different processors and inserting the required data communications so that they operate in a pipelined fashion. A pipeline communication library has been designed and developed so that it can be automatically inserted at compile time, even for codes that use unstructured datasets, which can only be fully analyzed at runtime. To the best of our knowledge, this is the first attempt to automate parallelization of sequences of loops by means of a pipeline. As such, and due to the complexity of the task, the main goal of this work is to demonstrate the feasibility of the approach, thus starting a research line in the parallelization of sequences of loops through pipelining.

The rest of the Chapter is organized as follows. Section 4.1 introduces previous works that are foundational to this Chapter. Section 4.2 introduces the communications library, as well as the modifications that have been performed in an existing compiler to allow parallelization in two dimensions: code and data. Section 4.3 analyzes the performance of a scientific code after such parallelization. Finally, Section 4.4 presents some concluding remarks taken from the experience of building the library and the results obtained.

4.1 Background and Related Work

The first relevant investigations about parallelizing compilers were undertaken around the decade of 1990 [HAA⁺96][BCG⁺95][WDS⁺95][AFMP95][Tse89]. Nowadays, the complexity of manual optimizations and the increasing parallel opportunities in heterogeneous, multi-core, massively-parallel supercomputers have moti-

vated a growing body of research in automated parallelization tools. A survey of the most important compiler automation works for procedural languages has already been presented in Section 1.2.4. A relevant example for this Chapter is the work by Scandolo et al. [SKH09], where the concept of *Synchronized Pipelining* is introduced to analyze the dependencies between different loops and determine which iterations of both loops can be run in parallel. Similarly to this Chapter, the approach is targeted at the parallelization of sequences of loops. Although the implementation given assumes shared memory and, contrary to our work, it does not seem to consider non-affine accesses, the mathematical background allows parallelization of sequences of loops in a wide range of applications. It is also able to handle nested loops, which we do not consider.

The Scala programming language [Scaa] introduces the concept of parallel collections to allow a certain degree of parallelization without user intervention. The language allows applying typical parallel operations such as *map*, *fold* or *filter* to these data structures, resulting in an automated parallelization. Graph for Scala [Scab] is a library targetting general graphs that provides parallel collections for graphs. Although this is a step forward in the search of automated parallelization for graphs, the functionality out of the box is limited, requiring manual intervention to parallelize custom codes based on unstructured meshes, hence the difficulty of comparison between this approach and the present work.

4.1.1 Skeletons

Because the data dependencies generated by unstructured meshes cannot be fully characterized at compile time, the compiler can only do partial optimizations on them. It is necessary to add further steps at run time in order to obtain a full picture of the data dependencies. One popular way to analyze properties of programs is by means of *skeletons*. A skeleton is generated from a piece of code by stripping parts of it in order to make it simpler and faster to run. Skeletons are useful when it is necessary to study certain properties of the code that can only be measured at run time; therefore, those parts of the code that do not affect such properties are removed, resulting in a lightweight version of the application. We use the concept of skeletons extensively throughout this Chapter.

Skeletons are not new in Computer Science. Sodhi et al. [SSX08] use skeletons to estimate application performance in grid computing. In this case, they define five types of operations that affect performance: CPU, memory, I/O, communication and synchronization and, finally, application phase transitions. Similarly to our approach, the skeleton is generated automatically. However, instead of a compiler, an ad-hoc generator is used that collects traces of the original program with the help of a profiling library, and then selects the operations of interest. This skeleton can then be used to estimate the performance of the program when

the platform (e.g. network conditions, CPU workload) is not fixed.

Ketterlin and Clauss [KC11] optimize memory tracing by using skeletons. By keeping only the operations that affect memory, they obtain a lightweight skeleton that can be run side by side with the original code. This avoids having to instrument the original code, obtaining an average speedup of 1.7 with respect to an instrumented version of the code throughout a set of 28 benchmarks from the SPEC2006 benchmark suite. Their approach is similar to ours in that the skeleton contains the memory operations, which we need to characterize the dataset dependencies and generate the communications between loops assigned to different processes.

The work by Diniz [Din03] is an application of the concept to HPC. A list of performance models is proposed first, and a compiler maps each BB in the application to one of the models from that list, generating the skeleton of the application by replacing the BBs with instances of the models. The performance data obtained by running the skeleton can then be used to estimate the performance of the application with real datasets without having to run the real application, signaling possible bottlenecks.

4.1.2 The ROSE compiler

We will be using the ROSE compiler as a starting point to implement the parallelization tool. ROSE [LQLM12][RDLQ12][Qui00] is a compiler infrastructure designed to apply source-to-source transformations. Several cutting-edge parallel optimization tools have already been implemented using ROSE, which makes it an ideal candidate for our tool. For example, Liao et al. [LYS⁺13] presents one of the first implementations of the accelerator directives from the OpenMP standard. Similarly, Sottile et al. [SRW⁺13] introduce a compiler that identifies data-parallel structures in Fortran and parallelizes them into accelerators. Liao et al. [LQWP09] propose an automated approach to parallelize codes for multicore processors. While these three works tackle the problem of automated parallelization, they all target shared memory systems.

4.1.3 Inspector-Executor

The inspector-executor paradigm (or I/E) is a technique to create parallel implementations of algorithms that was introduced by Saltz et al. [SCMB90]. It uses a runtime preprocessing stage, the *inspector*, prior to the actual computation of the algorithm, called the *executor*. The two parts of the code were originally called *preprocessor* and *executer*, and had to be manually implemented for each application. This approach has been used in the HPC community ever since to

4.1. Background and Related Work

Figure 4.2: Creation of inspector and executor

calculate data partitions and communications between processes for loops where the data dependencies are only entirely defined at runtime.

Inspectors are typically created by duplicating the sequential algorithm, and stripping the lines of code that do not result in accesses to data from the copy. This was called a *template* in the original work, and we extensively refer to it as *skeleton* due to its similarity to the works discussed in Section 4.1.1. Data access patterns can be analysed at run-time by the skeleton at a preprocessing stage. This later allows efficient data partitioning and therefore pre-calculating the chunks of data that the executor will need to communicate between processes.

Figure 4.2 illustrates the process of creating the inspector and executor. The first step is to copy the body of the kernel of interest and identify which parts of the code access the relevant data. In this case, the first block accesses array `a` in the order defined by the loop induction variable. The second block also accesses `a`, but in this case the traversal is given by a function whose result, to illustrate the case, depends on two other arrays that will be received as input to the program. At this stage, the parts of the code that do not affect memory access are removed, such as the computation of `r`. In the context of this work, a line of code that *affects* memory is a line that generates either a read or a write to memory, or a calculation used later to decide the location of a read or write.

The second stage in the Figure illustrates the analysis of the inspector and the computation of the accesses. This allows the executor to index array `a` by accessing another array instead of calling a function that depends on two more arrays. The first block in the inspector is also removed, since the compiler can detect that the array is indexed with the induction variable and does not require modifying the executor. This is a simple example; the real strength of the I/E approach is more visible when analysing complex implications of memory accesses such as communications, and will become clearer in the following paragraphs.

The I/E paradigm is only suitable for loops whose data dependencies are completely known before the execution of the main algorithm. This is, the dependencies must not change during the computations to be parallelized, but on pre-computations or data structures that are fixed once the kernel code starts. Nevertheless, this is a step forward with respect to static optimizations, such as the ones implemented by the programmer or those attempted by the compiler by merely inspecting the structure of the code. The discussed characteristics of the I/E approach make it a perfect fit for simulations that use unstructured meshes.

The work by Strout et al. [SLC⁺16] has a few similarities with our proposal. The polyhedral model has been already introduced in Section 1.2.4. These frameworks analyze the data accesses in a program and represent them as a polyhedron, which can later be used to reorganize the data accesses while honoring the dependencies between them. However, because the polyhedral model is restricted to affine data accesses and loops, it is not possible to apply it to graphs in general. The reason is that graphs are typically traversed by means of indirect accesses to an array that depend on the contents of a second array (i.e. $A[B[x]]$ instead of just $A[x]$). This type of accesses are not affine, and therefore this approach cannot be applied to the applications analyzed in our work.

In order to solve that problem, Strout et al. take advantage of the inspector/executor paradigm by generating a preprocessing stage with data accesses that will be fed into an executor that performs polyhedral reorderings at run time. The resulting proposal is known as the Sparse Polyhedral Framework. Similarly to this work, we use an inspector/executor approach to analyze the data access patterns of unstructured graphs. However, instead of using the information collected by the inspector to reorder the data accesses, we use it to find an efficient parallel partitioning and distribute the program across several processors.

4.1.4 IEC

The Inspector-Executor Compiler, or IEC, is an automation tool to parallelize programs for distributed memory systems using the Inspector-Executor paradigm. The target code must be annotated by the application developer with simple pragmas specifying the arrays that participate in the parallel computations. IEC then statically analyzes the code at compile time, and automatically generates the inspector in order to defer decisions to run time if necessary. It also wraps the original code into an executor, and inserts calls to communication directives. The tool was presented by Ravishankar et al. [REP⁺12] and is the basis of the work in this Chapter.

In Chapter 2, we introduced several techniques to optimize data access in codes that use unstructured meshes. Suppose a simple example where the mesh

is represented as an array of vertices and an adjacency matrix giving the connections between the vertices. At any given iteration, a loop in the code visits a particular vertex or edge, and performs computations that require data from its neighbour vertices, or endpoints in the case of edges. This requires looking into the connectivity matrix, which is typically given as an input to the program and therefore cannot be analysed at compile time. A more complex example would be the case where the loop traverses the mesh in a breadth-first manner, for example if we apply the optimizations mentioned in Chapter 2. In that case, each time that the loop advances to the next iteration, the connectivity matrix must be accessed in order to select one of the neighbours of a previous vertex. IEC is able to handle these cases by creating a skeleton of the application in the inspector.

Types of memory accesses to analyze

Memory accesses are typically triggered from the code when variables are read or written. Variables can be of many types but, for the purpose of this work, we restrict the analysis to arrays, as these are very common in scientific codes. Other types of variables, such as basic datatype variables or structures, are simpler to handle in a distributed context.

Arrays are categorised into two groups:

- *Indirection arrays* are used to access other arrays. The connectivity matrix in the previous example is one of such arrays, since its contents are used as indexes to access the vertex array. These arrays cannot change during the executor, otherwise the information calculated by the inspector becomes obsolete.
- The rest are referred to as *data arrays*. They contain data used for calculations and their contents may be modified during the processing, such as the vertex array in the previous example.

As said, IEC does not allow the contents of indirection arrays to change during the algorithm execution. For example, if the target algorithm changes the connectivity matrix, it will change the mesh topology, and the inspector will not be able to analyse the access patterns in advance.

Some codes dynamically modify the topology of the mesh during the execution of the simulation. For example, a mesh that represents the chassis of a car in a crash test simulation will deform, and extra vertices may need to be added in certain regions for improved accuracy. In theory, it would be possible to generate an inspector that traverses the mesh multiple times in order to analyse the changes in the mesh. In the worst case, the mesh would be traversed as many times as the original algorithm. This would increase the algorithmic complexity of the

inspector to the same as the target algorithm, rendering it useless in terms of performance.

Generation of the inspector

Typical automated parallelization tools target *affine loops*, where the loop bounds, conditionals and expressions used to access arrays are affine functions of iterators and parameters. When IEC analyzes the code at compile time, it searches for *partitionable loops*. These are a superset of affine loops where values stored in indirection arrays can also be part of the functions used to access arrays. Skeletons of these loops are added to the inspector, with expressions that result in data accesses being replaced by calls to a library that populates a hypergraph. This hypergraph is then partitioned, and each partition is assigned to a process.

Figure 4.3 illustrates the generation of the skeleton for an inspector with a small pseudocode application that works on a graph. The outer loop checks on an exit condition; scientific codes typically check that the algorithm reached a stable state, or bail out if a certain number of iterations is reached. The inner loop traverses the vertices of the mesh and, for each vertex, it does some calculations where its neighbours are involved, hence the accesses to the adjacency matrix.

The inspector skeleton is almost a copy of the original code, with some exceptions as the example in Figure 4.2 anticipated. First, the calls to the *compute**() functions have been removed. This is to avoid unnecessary work that does not affect accesses to data structures. Second, the outer time iteration loop is not necessary: the inspector assumes that the data access layout does not change across time iterations, and therefore analysing the mesh once per inner loop suffices. Third, `register_iter(loop_id, iteration)` creates initial data structures to represent this iteration of the loop as a vertex in a hypergraph. Observe that the inner loop has a unique identifier (which is more relevant when the code has several inner loops); this identifier allows later stages of the inspector and executor to retrieve the appropriate analysis information for each loop. Similarly, each iteration is assigned an identifier that is unique among other iterations in the same loop. Fourth, `mark_used(array, index, loop_id, iteration)` populates this hypergraph: each array index is represented by a hypergraph net which is connected by this function to the vertex (loop iteration) that it is used in. At the end of the inspector execution, each vertex may be connected to an arbitrary number of neighbours, and therefore the net used to connect them links all the iterations in all inner loops that use this array index.

```

1: {Time iteration loop}
2: while not end_condition() do
3:   {Traverse the mesh in random order}
4:    $j \leftarrow 0$ 
5:   while  $j < \text{size}(\text{vertices})$  do
6:     for  $j2 \leftarrow 0$  to  $\text{size}(\text{adjMatrix})-1$  do
7:        $n \leftarrow \text{adjMatrix}[j2]$ 
8:       compute( $\text{vertices}[j]$ ,  $\text{vertices}[n]$ )
9:     end for
10:     $j \leftarrow j + 1$ 
11:  end while
12: end while

```

(a) Original code

```

1:  $j \leftarrow 0$ 
2: {This loop was assigned ID 0 at compile time}
3: while  $j < \text{size}(\text{vertices})$  do
4:   register_iter(0,  $j$ )
5:   for  $j2 \leftarrow 0$  to  $\text{size}(\text{adjMatrix})-1$  do
6:     mark_used( $\text{adjMatrix}$ ,  $j$ ,  $j2$ , 0,  $j$ )
7:      $n \leftarrow \text{adjMatrix}[j2]$ 
8:     mark_used( $\text{vertices}$ ,  $j$ , 0,  $j$ )
9:     mark_used( $\text{vertices}$ ,  $n$ , 0,  $j$ )
10:  end for
11:   $j \leftarrow j + 1$ 
12: end while

```

(b) Inspector

Figure 4.3: Pseudocode example of inspector generation

Hypergraph partitioning

With all the information gathered by running the inspector, a standard hypergraph partitioning algorithm can later be used to:

- Optimize load balancing by assigning the same amount of vertices to each processor.
- Minimize communications by assigning iterations linked by the same nets (array indexes) to the same processor.

Figure 4.4 gives an example hypergraph for a code snippet with two arrays and two loops. The iterations in the first and second loop are represented by nodes i_x and j_y respectively. The nets represent each index in arrays **a** and **b**. Since loop **i** has three iterations, the hypergraph shows vertices i_0 , i_1 and i_2 . Similarly, the six iterations of loop **j** are represented by the six vertices j_0 to j_5 .

Now, $a[0]$ is used in iteration i_0 , j_0 and j_3 ; $a[1]$ is used in iteration i_1 , j_1 and j_4 ; and so on.

Suppose that the application runs with two processes, and that the partitioner decides to assign i_0 and i_1 to the first process and i_2 to the second. In order to minimize the nets that cross partitions (i.e. communications), the partitioning of loop j should assign j_0 , j_1 and j_3 to the first process. j_2 and j_5 should definitely be assigned to the second process, and j_4 shall be assigned to either of them, but for the sake of load balancing it will be assigned to the second. At the end, process 1 is assigned five iterations and process 2, four. In practice, iterations of distinct loops may affect load balancing differently.

```

a[3]
b[6]
for i in 0 to 2
  a[i] = 2*i
  b[i] = 0
  b[3+i] = 6
for j in 0 to 5
  b = b[j] + a[j % 3]

```

Figure 4.4: Example of an IEC hypergraph

A key observation is that, in IEC, the iteration space of each loop is split among all processors. For codes where several loops are partitionable, they are all computed by all processors. This is the reason why a single hypergraph is used even in cases where several inner loops are available for parallelization: an array may be used in more than one loop, and the partitioning choices have an effect on the amount of communications required when the processes finish one loop and start with the next one. Also, because processes communicate between the end of a loop and the start of the next loop, they must be all synchronized by means of a barrier at the end of each loop. As a side-effect, this guarantees that communications take place only at the start or end of the loops. In IEC, the barrier is implicit, as it uses MPI blocking collective communications that prevent processes to continue until all of them have the data required.

In sum, the IEC inspector performs the following actions:

1. A global rank is assigned to each process.
2. The inspector is created and initialized with the above information, together with global information such as the number of loops, data and indirection arrays.

3. Array objects (both data and indirection) are given an identifier, initialized with the relevant information such as size, and initially block-partitioned among processes.
4. Loops are initialized with information about the used arrays. Each data array is marked as read-only or read-write.
5. Hypergraphs are created by running the loop skeletons.
6. The hypergraph is partitioned. IEC partitions one single hypergraph that holds information about the uses of all data arrays in all loops.
7. Sections of the arrays are assigned to each process.

4.2 Modifications to IEC for pipelining

The fact that all processes participate in the computation of every partitionable loop is a problem for heterogeneous systems. Because processors have different computational capabilities, it is not clear that load-balancing the dataset equally will yield best results. Also, there may be cases where a processor is not suitable for computing some of the loops. For example, an FPGA may not have enough resources to contain kernels for all the loops. Furthermore, the implicit barrier at the end of the loops implies that, if load-balancing is not carefully tuned, some processors will be idle while others struggle with their computational load.

Another challenge in IEC is the scalability of the solution. A high number of processors, such as when executing in a high-performance cluster, adjusts to the diminishing returns law: the data partitions become smaller, and the frontiers between partitions become bigger in comparison. Therefore, the ratio of computation to communication decreases, and the solution is less efficient.

This Section introduces the implementation of a solution based on IEC that also splits the set of partitionable loops into different processor teams. The teams are arranged in a pipeline so that computations in one loop can be started as soon as the data required from previous loops is ready for each iteration, without having to wait for the predecessor loops to complete. We refer to this solution as *IEC/pipeline* throughout the text in order to distinguish it from the original IEC.

The IEC/pipeline approach should be, in theory, friendlier to heterogeneous systems than the original IEC implementation. Firstly, it allows more flexibility when mapping parts of the code to processors. For example, FPGAs yield best results when dealing with long pipelines with a small amount of input and output data. On the other side, traditional CPUs can handle a high number of

input/output data thanks to their cache memories, but do not perform well with large loops due to limited fine-grain parallelism and the register spills required by small register banks. In such cases, IEC/pipeline would allow allocating different loops to the most adequate set of processors. Additionally, standard loop splitting techniques could be used to obtain a finer granularity with more loops. Therefore, the code can be arranged in a way that is most efficient for the set of available processors.

Second, IEC/pipeline removes the synchronization points at the end of the loops, allowing groups of processes to run in a loosely-coupled manner. As soon as the data are ready in one loop, it is transferred to the processors computing the next loop, instead of waiting for the loop to finish in all processors.

Third, partitioning both the dataset and the inner loops allows increasing the size of data partitions with respect to IEC on the same number of processors, since each loop is computed by a smaller group of processors. This reduces the borders between partitions, thus reducing halo-swapping between processors assigned to the same loop. In fact, it can be entirely removed if the iterations in an inner loop only need data from the previous loops. This is a typical case in mesh-based scientific codes, where the inner loops process the data at time iteration N from the data at iteration $N-1$. In other words, assuming that the inner loops traverse the dataset, vertices and edges require data from neighbouring vertices from the previous outer iteration, but not from the current one.

There is a potential pitfall however. In IEC, only a small part of the data needs to be communicated between processors: the borders of the data partitions. Other parts of the arrays are local to the processors; these will process each loop in the same portion of data. In the case of a pipeline approach, teams of processors now need to communicate all the data throughout the pipeline. If a loop uses data that a previous loop writes to, it needs to receive the complete chunk of the array, since the receiver processor does not have any part of the array. In sum, the communications between teams may completely shadow the performance gains obtained by splitting the set of loops. A smart implementation of the communications is vital to avoiding this.

The library has been implemented using message passing, in particular MPI. This allows the compiler to target massively parallel distributed systems interconnected by a wide variety of networks such as all kinds of ethernet, Infiniband, and shared memory buffers when available.

4.2.1 Load partitioning in two dimensions

IEC/pipeline partitions the load in two dimensions: these will be called the *data* dimension and the *loop* dimension; these are not to be mistaken for the dimensions of a mesh (i.e. two or three-dimensional meshes). In the data dimension,

partitioning is done in a similar way to IE: all positions in data arrays are turned into nets in a hypergraph, which is then partitioned resulting in the arrays being partitioned as well among processes. The loop dimension covers the set of inner loops: processes are grouped into teams, and each team is assigned to a single loop. The initial proof-of-concept implementation has a simple partitioning scheme in this dimension: each loop is a unit of work that will be assigned to a single team of processes, and each team is assigned to a single loop, with the arrays involved in the loop computation partitioned among the processes of the team.

A separate but related problem is creating the teams. Here we also take a simple approach where all teams have the same number of processes. Therefore, if the code has three loops, we need at least three processes (team size = 1), and increasing in multiples of three (6 for a team size of 2, 9 for a team size of 3, etc.). The team size also needs to be taken into account when partitioning in the data dimension.

Figure 4.5 shows a diagram where loops are shown together with the data structures read or written in them. Each array in a loop may be read-only (R), write-only (W), read/write (R/W) or an indirection array (I). Indirection arrays are read-only by definition, as explained before; this requirement allows the compiler to insert an inspector with $O(N)$ complexity, where N is the size of the mesh. The diagram represents part of a seismic simulation [BBG⁺96] widely used as part of benchmarking suites that we will use later to analyze the performance of IEC/pipeline.

Each loop is a *stage* in the pipeline, which is computed by exactly one team of processes. Every team that generates data required by a later stage is called a *producer*. In this example, all teams are producers except for the last one. Conversely, every team that requires data from an earlier stage is called a *consumer*. Every team except for the first one is a consumer. For example, loop 1 consumes array *disp0* from loop 0, this is, loop 1 is a consumer of loop 0, and loop 0 is a producer for loop 1.

At the end of each iteration of the outer loop, there is a special case where the last team acts as a producer to the first team, although the data produced by the last team at iteration i turns into the data received by the first team at iteration $i + 1$ of the outer loop. We generally use the term *producer* and *consumer* only when it applies to communications within the same outer loop iteration.

4.2.2 Data dependencies

The data dependencies between loops define the communications between teams. Data dependencies are commonly categorized into three types: read-after-write (RAW) or true dependencies, write-after-read (WAR) or anti-dependencies, and

Figure 4.5: Partial diagram of a seismic simulation

write-after-write (WAW) or output dependencies. RAW dependencies between two loops effectively mean that there is a producer/consumer relationship between the writer and the reader loop. Therefore, they translate directly into a communication need in the pipeline.

The two other dependencies are innocuous in the context of a pipeline and can be completely disregarded. In a situation with a WAR dependency, a processor W writes to some array and is placed in the pipeline after processor R, which reads from the same array. W does not affect the values read by R since it is placed later in the pipeline, and R obviously does not need W's values since it is placed before. The reasoning is similar for WAW dependencies. WAR and WAW dependencies are relevant in the context of scheduling, as for example when generating kernels targetting an FPGA that comprise a single loop, when fusing loops together, or in problems related to resource sharing such as register allocation. However, they do not translate into communications across pipeline teams and therefore are irrelevant in the context of this thesis.

Figure 4.6: Two approaches to pipeline dependency communication

In the general case, there may be data dependencies between loops that are not placed one immediately after the other. This is illustrated in Figure 4.6, where an array *a* is written in the first loop and read in the third loop. There are two options to handle this requirement in the communications library. The first one would be to allow teams to send data to all consumer teams. This is the case of the diagram in the center. The second would be to propagate data from producers to consumers along the pipeline; this is, if process *M* produces data for process *M* + *N* (where *N* is the separation between both processes), *M* would send data to process *M* + 1, which would in turn send it to *M* + 2, and so on until the data reaches process *M* + *N*. This is depicted in the diagram to the

right, where a *shadow* dependency on array `a` has been added to the second loop.

The first option implies that the communication library must be able to synchronise sends and receives between multiple processes, which requires a more loosely-coupled form of communication that is not trivial to optimize. The latter, in contrast, would allow restricting communications in processes to two other processes: the immediate predecessor and successor in the pipeline. Sadly, the fact that data must go through multiple teams between producer and consumer could increase the level of communications by $O(\text{pipeline length})$ in the worst case. This disadvantage is the reason why it has been decided to implement the communications library so that processes can communicate with all its consumers. This will be discussed more in-depth in the following sections.

4.2.3 Changes to the inspector

The modifications to IEC/pipeline mostly consist of a library of classes and functions that can be easily inserted by the source-to-source compiler into the application code, in the same way that the original IEC does. The aim of this initial demonstrator is to investigate the feasibility of the pipelining approach in a homogeneous cluster, for which a simple partitioning suffices: the processes are split equally among as many teams as loops.

In IEC/pipeline, each process is assigned a team at the beginning of the inspector. All processes join a global MPI communicator `global_iec_communicator`, and processes in each team join an additional `team_communicator`. The original IEC has a function `set_inspector()` that creates and initializes a singleton inspector. The initialization comprises the setup of the global communicator, registration of data arrays into `global_data` objects, and allocation of buffers for the communications. IEC/pipeline additionally registers loops individually in `global_loop` objects and sets up the team communicators. Each loop holds information about the arrays that are used in it, which of them are read, write, or both, a flag indicating if the loop is the last one in the pipeline that writes to each of the arrays, and an equivalent flag to indicate if the loop reads the arrays without any previous loop writing to them. The last two flags are later used to decide which teams need to feed data to the next iteration of the outer loop, and which ones need to receive data from the previous iteration. The loop structures are initialized and given a loop ID.

Once the inspector, loops, and data and indirection arrays have been registered and initialized, the skeletons for each loop are run. These are similar to the ones in IEC: calls to `add_vertex()` and `add_pin_to_net()` initialize the hypergraph. The difference with IEC is that each team creates its own hypergraph. A team only runs the skeletons for the loop that they own, this is, the one that will be run in the executor, and for the consumer and producer loops. This allows

each team to request information to consumers and producers in order to know exactly which data needs to be communicated and when. After this, the hypergraphs are partitioned and chunks of each array are assigned to the processes in each team where they are used.

The information gathered from the partitioning can then be used to calculate all the information about communications. At this point, the library code knows which parts of the arrays are needed in which processor, and at what iteration of the inner loop. A call to `pipe_calculate_info()` follows. This function calculates what indexes are needed for each array that is read, which ones must be sent in the case of arrays that are written, and who are the producers and consumers of the current process.

`pipe_calculate_info()` also calculates the data that must be fed back into other processes for the next iteration of the outer loop. For that, two groups of arrays are populated at compile time and given to each team: `lastWriteX` and `readNoPreviousWriteZ`, where X,Z are loop numbers. The rationale behind this calculation is as follows: if a team X is the last one to write to an array A, its processes will need to pass data to the user teams for the next outer-loop iteration. This is recorded as `lastWriteX[A] = 1`. Similarly, if a team Y reads array A, it will need to obtain the information from somewhere. If a previous team Z (in the same iteration) writes to array A, then Y will receive A from the processes in loop Z. However, if there is no previous team that writes to A, team Z will need to receive A from team X in the previous iteration. When a team reads an array and there is no previous writer team, this information is recorded as `readNoPreviousWriteZ[A] = 1`. It is then trivial to match consumer and producer teams for outer-iteration communications. With this and the information about the array partitioning, the inspector can calculate which processes are senders or receivers, which parts of the array must be communicated, and to/from which processes they need to communicate at the end of each outer iteration.

The communications between teams in the same outer iteration vary with each inner iteration, as each inner loop iterates through the data structures and therefore each iteration uses different parts of the arrays. This requires a considerable amount of data to be calculated by the inspector. Conversely, the communications between outer iterations are identical across iterations, as the outer loop iterates through the whole arrays, and the partitioning and data structures are immutable.

The inspector generated by IEC/pipeline is as follows:

1. A global rank, team ID and team rank are assigned to each process.
2. The inspector is created and initialized with the above information, together with global information such as the number of loops, data and indirection arrays.

3. Array objects (both data and indirection) are given an identifier, initialized with the relevant information such as size, and initially block-partitioned among processes.
4. Loops are initialized with information about the used arrays. Where IEC only needed to know if the array was read-only or read-write, IEC/pipeline keeps separate information about read and write properties, as it affects the layout of RAW dependencies (an array can be write-only and therefore it will be only a producer for that array).
5. Hypergraphs are created by running the loop skeletons.
6. Hypergraphs are partitioned. Although IEC/pipeline has as many hypergraphs as loops, they are smaller because they only need local information to the loop instead of the global knowledge required in IEC.
7. Sections of the arrays are assigned to each process.
8. Data required for pipeline communications is calculated.

Stages 3, 5 and 7 are identical to IEC. Stages 1 and 2 require some additional information related to the pipeline approach, such as team IDs. The remaining stages are specific to IEC/pipeline and provide the required data to calculate the inter-loop communications required during the executor.

4.2.4 Changes to the executor

IEC inserts a call to `setup_executor()` before the actual computational kernel. Each process receives the parts of the data required for their computations according to the partitioning. Additionally, IEC/pipeline allocates send and receive buffers for the pipeline communications. In order to perform the pipeline transfers, the compiler will have inserted calls to `pipe_send()` and `pipe_receive()` at the start and end of each iteration of the inner loops respectively.

Collective communications, performance and deadlocks

Whenever a group of processes communicate similar data or communicate at the same time, collective communications can be used. Collectives are typically faster than using point-to-point communications: they save startup time and, depending on the implementation, often feature improved message scheduling. In addition, some systems have dedicated tree networks for collective communications, such as the BlueGene/L [AAA⁺02]. IEC/pipeline might appear as an ideal case for collective operations. However, as we discussed in Section 4.2.2,

4.2. Modifications to IEC for pipelining

all producers and consumers communicate between each other. In this context, collective communications are not the best choice.

It is common to have pipeline producers that need to multi-cast the same data to multiple consumers, which could be easily achieved with a call to `MPI_Bcast()` on an MPI communicator comprising the sender and the receivers. One broadcast would be required for each array being transferred, since the set of consumer processes may be different for each of them. Similarly, since consumers receive data from multiple producers, one broadcast call would be required for each received array. Assuming blocking communications, each broadcast must succeed in order for the next one to start. However, the order in which a producer sends the data may not be the order in which the consumers need it. Furthermore, given a single producer, each consumer may need the data in a different order, for example if one of them uses an access array to index the data array, and another one indexes it by means of the loop induction variable. To make things worse, each consumer will most likely depend on an entirely different set of producers. This situation is ideal for dead-locking situations and performance degradation.

Figure 4.7: A blocked loop in the pipeline as a consequence of a dependency in an unrelated loop

An example is depicted in Figure 4.7. Loop 1 broadcasts some data to the first iteration in loop 2, and the second iteration of loop 3. Because the broadcast must wait until all consumers receive the data, this broadcast keeps waiting because loop 3 cannot proceed until it gets the data from loop 2. Now, because the

broadcast has not finished, loop 2 will not initiate the broadcast and the whole program dead-locks.

There are non-blocking versions of collectives in MPI. If we use a non-blocking broadcast, loop 2 successfully receives the data from loop 2, it initiates the broadcast and eventually loop 3 receives the first broadcast from loop 1. However, if both broadcasts in loop 1 are in different iterations, and we restrict the length of the non-blocking broadcast to a single iteration (i.e. we wait for the communication to finish at the end of the iteration), loop 4 will be blocked for some time until loop 3 finishes iteration 2. In other words, loop 4 incidentally blocks because of loop 3 even when there is no dependency between them. Limiting the length of the non-blocking broadcast to a single iteration is a sensible restriction, since otherwise the library should maintain a considerable number of handles for open connections (a group of `MPI_Status()`, in MPI jargon).

The problem with broadcasts, and collective communications in general, is that they impose an artificial synchronization point among their participants: all processes block until the whole communication is finished, even when there is no real dependency between the consumers.

Loosely-coupled communications

The solution to deadlocks is to avoid these artificial constraints by using non-blocking point-to-point communications. We use the term *loosely-coupled communications* here to stress the fact that producers may send data at different times than expected by the consumers, and these need to be able to correctly identify them and update the appropriate local data structures. In order to achieve this, the compiler inserts calls to the IEC/pipeline communication library, making this loose coupling transparent to the application code.

At the end of each iteration, `pipe_send()` searches in the arrays for the data that have a final value after the current iteration, and packs a message to each consumer. This package is sent through a call to `MPI_Issend()`. After sending to all the consumers, the producer calls `MPI_Waitall()` to make sure that all processes received the data.

Similarly, each consumer calls `pipe_receive()` before starting an iteration. This function first checks if the consumer has received all the data required to start the computations in the current iteration by calling `pipe_ready()`. In order to do this check, it keeps track of how many messages have been received from each producer, and compares it with a “safe” value that changes for each producer and inner-loop iteration. This value is calculated by the inspector based on an analysis of the producers, and therefore every consumer knows when it is safe to proceed with the calculations when the required data have been received. For example, if loop A is about to start iteration 7 and the safe value for producer B

is 3, this means that A must have received at least 3 messages from B in order to start the iteration safely.

If the consumer is “not ready”, it calls `MPI_Waitany()` to wait for messages from any producer. Each message carries information about the sender so that the receiver can look up the mapping between received data and local data structures. These tables contain, for each sender, information about the order in which indexes in arrays are received, and are calculated by the inspector. Once the mappings are known, the data are copied from the MPI message into the local data structures. Afterwards, a new `MPI_Irecv()` is issued to the producer that the data were received from. Finally, the producer checks again whether it is ready to start with the iteration. The cycle is repeated until all data required for the iteration have been received.

This form of communication implies that library calls in the consumers are responsible for the correct synchronization, whereas the producers will send data as soon as it is ready. The inspector populates tables for each array holding information about when data are ready. This information is computed based on the read-write information of the array: an array position is ready at the last iteration of the inner loop that writes to it.

Communication between iterations of the outer loop

At the end of each outer-loop iteration, there is a global synchronization point. Processes that wrote arrays for the last time communicate the results to the first teams that read them. Since these communications do not have the problems discussed in the previous Sections, they have been implemented with collective communications. Specifically, a call to `MPI_Alltoallv()` has been used in order to trigger an all-to-all communication where each process can define different sizes for data to be sent or received, and for different processes.

Required data structures

This is a summary of the data structures related to the pipeline communications that form the interface between the inspector and the executor. These are populated by the inspector using the partitioning information, and used by the executor to perform the pipeline communications and synchronize the process execution.

For each data structure, we first explain its purpose, then we give the C++ prototype, and finally we annotate it with the meaning of each field.

usedArrays: Each inner loop holds, for each iteration, the identifiers of the arrays that are read. `pipe_receive()` iterates through this structure, and uses

the identifiers to query each array (represented by a `local_array` object) about the data that needs to be received from the producers.

```
std::map<int, std::set<int> > usedArrays
std::map<iteration, std::set<loop_ids> > usedArrays
```

computedArrays: Similarly, each inner loop holds, for each iteration, the identifiers of the arrays that are written. `pipe_send()` iterates through this structure, and uses the identifiers to query each array about the data that needs to be sent to the consumers.

```
std::map<int, std::set<int> > computedArrays
std::map<iteration, std::set<loop_ids> > computedArrays
```

pipeSendIndexes: Local arrays are related to a loop, and therefore to a team. Each local array holds, for each iteration, all the indexes that need to be transferred to each of the consumers. `pipe_send()` queries the `computedArrays` for this information.

```
std::map<int, std::map<int, std::vector<int> > > pipeSendIndexes
std::map<iteration, std::map<consumer, std::vector<indexes> > >
pipeSendIndexes
```

pipeRecvIndexes: Similarly, each local array holds, for each iteration, all the indexes that need to be received from each of the producers in order to start with the next iteration. `pipe_receive()` queries the `usedArrays` for this information.

```
std::map<int, std::map<int, std::vector<int> > > pipeSendIndexes
std::map<iteration, std::map<producer, std::vector<indexes> > >
pipeSendIndexes
```

safeIter: Each loop holds a vector indexed by local iteration number which contains, for each producer, the minimum number of messages that the loop needs to receive in order to be certain that all the data required for the iteration have been received.

```
std::vector<std::map<int, int> > safeIter
std::vector<std::map<producer, min_messages_received> > safeIter
```

4.3 Performance investigation

The performance of the IEC/pipeline communication library has been tested against a real-world earthquake scientific simulation software called *quake* [BBG⁺96]. This application has some characteristics that make it ideal for the performance

evaluation: similarly to the CFD code used in the first part of the thesis, it is based on unstructured meshes. Also, the computational kernel is a single outer loop that iterates over a number of timesteps, with 15 inner loops that iterate over the mesh data structure. The reason for using this benchmark instead of the CFD code analyzed in Chapter 2 is mere convenience: it is included in the original IEC as one of the demonstration codes, and should allow easy comparison of the results between both approaches. *quake* is also part of the known benchmark suite SPEC2000 [SPE] and, although the complete suite is a commercial product, the benchmark itself is publicly available.

The experiments have been taken in two machines. The first one is a multicore system with shared memory; the second is an HPC cluster. The multicore system has an Intel Core i5-4200U processor with four cores and 16 GB of main memory. Since the limitations of this system for HPC are obvious, it has been used only for debugging and gathering of profile data.

The second system is *Magerit* [Mag], the HPC cluster run by the Supercomputing and Visualization Center of Madrid (CeSViMa). The computer is in fact split into two clusters, one based in the IBM Power7 architecture and another based in the Intel Xeon one. The results shown throughout this Section are taken in the Intel cluster, which contains 82 octa-core Intel Xeon E5-2670 processors. Each processor has a peak performance of 332 GFLOPS, which makes a total peak performance of 13'64 TFLOPS. The system has 2'624 TB of main memory, thus allowing 32 GB of RAM shared among the eight cores of each processor. Each processor is a separate node, and different nodes are connected through an Infiniband QDR network. This network can be accessed by the parallel applications through message passing, specifically OpenMPI 1.6.5.

4.3.1 Performance results of the pipelined *quake*

This Section presents the performance results in Magerit, and compares them to the original sequential code. *quake* has been re-implemented following the inspector-executor model with the help of the compiler and manually adding the calls to the pipelining communication library. The current state of the compiler only allows processes to be part of one team, this is, each process is assigned to a single inner loop. So far, there is only one process per loop, therefore the partitioning is in one dimension only. This implies that 15 processes will be used.

Three meshes have been used for the experiments: test, small and large. Table 4.1 shows the sizes of the meshes, and Table 4.2 gives the runtime results of both sequential and pipelined versions. The times shown are only for the application kernel in order to discard less meaningful startup and shutdown serial time, such as reading the input file. In order to avoid pathological cases, each number is the median of five runs. Such pathological cases would involve, for

Mesh	Vertices	Edges	δ
test	20	26	1.23
small	7294	35025	4.80
large	30169	151173	5.01

Table 4.1: Characteristics of the datasets used with *quake*

example, ideal or too disadvantageous mappings between processes and processors in the cluster, or a run being unreasonably affected by the operating system evicting the parallel processes from the processors.

Mesh	Original (s)	Pipelined (s)	Time increase
test	3.88×10^{-4}	8.88×10^{-3}	22.9x
small	0.17	3.08	18.1x
large	61	1619	26.5x

Table 4.2: Execution time comparison between the original and the pipelined *quake*

There could be various reasons for the slowdown in the parallel version. Since the comparison is relatively similar for all meshes, we will undertake a more complete performance study on the smallest dataset for simplicity. Figures 4.8 and 4.9 show the profiling of the sequential and pipelined codes with **perf stat**.

Figure 4.8 shows that the sequential code has no CPU migrations and a small amount of context switches. A context switch occurs when the currently executing process is evicted from the processor by the system scheduler in favour of another process that is waiting. These events are highly disrupting, as they require saving the registers in the CPU, loading the registers for the next process, and jumping to the next instruction from the new process, which often has negative performance side-effects in the level-1 caches. A CPU migration occurs after the context switch of a process, where such process is later assigned to a different processor than the one it was previously being executed on. When a process is context-switched and CPU-migrated, the performance penalty resulting from the local caches is higher than the case where it is not migrated, due to the local cache not having any of the instructions from that process in the instruction cache. The processors used have separate instruction and data caches, and a similar phenomenon occurs with the data cache.

Figure 4.9 reveals that the pipelined version has about 891 times more context switches than the sequential version, which explains why it also has so many CPU migrations. Since the pipelined code takes about 23 times longer than the

4.3. Performance investigation

```
[IEC]: OriginalTime:0.000374

Performance counter stats for 'orig-quake test.in':

    1,628602 task-clock          #    0,141 CPUs utilized
           7 context-switches  #    0,004 M/sec
          0 CPU-migrations      #    0,000 M/sec
        202 page-faults        #    0,124 M/sec
    4851271 cycles              #    2,979 GHz
    2826307 stalled-cycles-fee #   58,26% frontend cycles idle
    1982447 stalled-cycles-be  #   40,86% backend  cycles idle
    5486165 instructions        #    1,13  insns per cycle
                                   #    0,52  stalled cycles/insn
       733173 branches          #   450,185 M/sec
        14545 branch-misses    #    1,98% of all branches

0,011564538 seconds time elapsed
```

Figure 4.8: **perf** analysis of the sequential code

```
[IEC]: ExecutorTime:0.008880

Performance counter stats for './iec-quake_block test.in':

    149,981809 task-clock          #    0,089 CPUs utilized
         6237 context-switches    #    0,042 M/sec
          68 CPU-migrations        #    0,000 M/sec
        3430 page-faults          #    0,023 M/sec
    405697244 cycles              #    2,705 GHz
    301365340 stalled-cycles-fee #   74,28% frontend cycles idle
    264438201 stalled-cycles-be  #   65,18% backend  cycles idle
    228071078 instructions        #    0,56  insns per cycle
                                   #    1,32  stalled cycles/insn
       51463569 branches          #   343,132 M/sec
        865378 branch-misses     #    1,68% of all branches

1,680210206 seconds time elapsed
```

Figure 4.9: **perf** analysis of the pipelined code

sequential code, it would explain to some extent why there are more context switches, for example due to the kernel processes. However, the high numbers are an indication that the *quake* concurrent processes are becoming blocked very often and therefore the scheduler decides to evict them from the CPU. This is backed by the high percentage of cycles that the CPU frontend and backend are idle in the pipelined version. This is most likely caused by the overhead introduced by the communications between processes in the pipeline but, as *perf* is designed to measure single processes, it is hard to analyze the exact behaviour of parallel applications.

4.3.2 Parallel profiling of the pipelined code

More detailed information about the parallel program has been gathered using the TAU parallel profiler [SM06][TAU]. This profiling utility can instrument the executables by using binary instrumentation, where recompilation is not required, or by using an automatic instrumentation library at compile time. It also allows manual instrumentation of the code. Due to technical problems, the following experiments have not been taken in the cluster but in the multicore system.

Figure 4.10 shows the results of a profiled run of the pipelined code with the test input dataset. As before, the profile only captures the computational kernel. It can be immediately noticed that the vast majority of the time is spent in the MPI communication primitives. For reference:

- `MPI_Waitany()` is called by consumers waiting for data from the producers.
- `MPI_Alltoallv()` is used by all processes to receive data in the borders between partitions before starting the kernel execution (commonly known as *halo swapping*), and at the end of each iteration of the outer loop of the kernel to communicate the data back to the first processes for the next iteration.
- `MPI_Waitall()` is used by the producers to wait for all the sends to the consumers to complete at the end of each inner loop's iteration.

The figure clearly shows two groups of processes; 0, 5 and 10 do not use `MPI_Waitany()`, while the rest do. This is due to the fact that, in this computational kernel, the loops assigned to the processes in the first group are not consumers, only producers. An interesting observation is that the effect of this function increases with the node identifier: node 1 and 2 spend considerably less time in this function than the later nodes. This is because the node IDs are orderly assigned to the inner loops in the kernel. Therefore, node 14 is only a

4.3. Performance investigation

Figure 4.10: Distribution of execution time per thread among different functions

consumer and therefore will be most of the time waiting for results from its producers. This effect comes from the fact that there is a synchronization point after each iteration of the external loop in the form of an `MPI_Barrier()`. The reason for placing this barrier is to limit the amount of information that we must hold in order to keep processors in sync. In the absence of this barrier, processors with a lower computational load could overtake other processors and start computing later iterations. It should be possible to implement this with a less aggressive approach to allow processes to continue working without overtaking others by more than one iteration but it is not currently implemented. Such implementation would reduce the impact of `MPI_Waitany()` in the profile (and possibly `MPI_Waitall()`), thus improving performance.

Figures 4.11, 4.12 and 4.13 show the distribution of execution time for different functions in processes 0, 7 and 14. For each process, only the ten most time-consuming functions are shown. The figures show that the values are exclusive, meaning that the time measured for each function excludes the time spent in other functions called from the measured function.

As in Figure 4.10, these figures show that the vast majority of the time is spent in MPI communication primitives. At the early stages of the design, one of the possible bottlenecks that were expected to happen was that accessing the control data structures created by the Inspector could affect the performance. These data structures include those needed to know the array indexes that need to be sent at each iteration for each consumer, and likewise for the indexes that must be received by the producers. They also include the synchronization variables that decide when a process has received enough data to carry on with the next iteration. Accessing these data structures do not seem to have a considerable impact according to the profile data.

Barring the MPI primitives, the next functions to have an effect on timing are

Figure 4.11: Detailed profile in process 0

Figure 4.12: Detailed profile in process 7

Figure 4.13: Detailed profile in process 14

`pipe_send()`, `pipe_receive()`, `pipe_populate_send_buf()` and `pipe_update()`. The third function takes care of packing the data to be sent to consumers from a specific local array into a send buffer. Similarly, the last function updates the local arrays with data received on a receive buffer from producers. The first two functions encapsulate the last two, applying them for all arrays to be sent and taking care of the actual communications required with MPI primitives. The profiles show that the time taken is between five and ten times smaller than the performance of the original code, and therefore do not have a significant impact in the overall performance of the library.

4.4 Conclusions

In this Chapter, the feasibility of parallelizing the execution of sequences of loops to operate in a pipelined fashion has been demonstrated. This starts a new line of research that will allow the automatic partitioning of the computational load of HPC applications, not only in the data dimension, but also in the loop dimension. This line has a promising future as it implies additional flexibility when mapping processes to processors in the most efficient way, and also when using FPGAs in the heterogeneous system as they are most effective with pipelined architectures but have limitations in terms of resources.

We have presented a communication library to allow this style of pipelining in scientific simulations with unstructured datasets. The design of the library uses the inspector-executor approach to analyze the dataset dependencies at runtime efficiently. The approach has been tested in an HPC cluster with a widely used scientific application based on unstructured meshes, and the conclusion is that the implementation of the library and the integration with the Inspector/Executor approach do not have a significant impact on the performance of parallel simulations.

However, the previous performance analysis in Section 4.3 has shown important limitations in the basic implementation of the proposed approach, mainly due to the time spent in communications. The evidence collected suggests that this approach could be improved by the following actions:

- Apply the approach to a code with longer inner loops. The kernel in *quake* is split across 15 loops with varying computational loads. Some of the loops only initialize arrays, whereas other loops carry computationally expensive operations, but at the end none of the loops seem to be particularly intensive in terms of CPU time. An application with longer loops would increase the computation-to-communication ratio, therefore minimizing the cost of pipeline transfers between the loops and taking more

advantage of IEC/pipeline. This is similar to the effect of the dataset size when partitioning the data instead of the code, which was first pointed out by Gustafson [Gus88] in response to the pessimistic view of Amdahl’s law with regards to parallel processing.

- Pack the messages of several iterations of the inner loops into one. It may be the case that the messages transmitted at the end of each inner loop’s iteration are small, therefore being affected by the network latency rather than bandwidth. In theory, packing the messages for 10 iterations could divide the communication cost by a factor of 10. In practice, this factor will be smaller due to the network’s throughput limitations.
- Apply loop fusion to remove some loops, decreasing the pipeline depth while increasing the computational load of existing loops. The optimal number of loops should be determined empirically but, at the moment, it is clear that *quake* has too many loops compared to the overall amount of code inside its kernel to make pipelining viable. Loop fusion is a standard optimization that can be carried by any modern compiler, but this optimization is driven by heuristic functions that would need specific tuning for IEC/pipeline.
- Remove the barrier at the end of each iteration of the external loop, as already explained earlier in this section. This would allow processes that have already finished the current iteration to start with the next one, minimizing the waiting issues experienced by the processes situated at the end of the pipeline. However, this action would have a greater effect in algorithms where the inner loops are better load-balanced between each other.

Future work should focus on implementing these performance improvements. Also, making the compiler insert the calls to the IEC/pipeline library automatically would allow us to apply this approach to other benchmarks and real applications with minimal cost. Finally, integrating IEC/pipeline with already existing sequential optimizations such as loop fusion or splitting, together with adding heuristics to aid load balancing, would allow the compiler to automatically rearrange the code across the loops and perform load balancing in the code itself, similarly to how it is already done with data partitioners.

An important remark is that, while the compiler does not currently insert the calls to the IEC/pipeline library, this is not a technical challenge. The compiler inserts calls to the original IEC library, and the new library only requires data already being computed by IEC, and some additional data that can be easily obtained at compile time. One example is the data dependencies between loops which, in LLVM and other popular compilers, can be done by means of def-use chains and existing pointer aliasing analyses.

4.4. *Conclusions*

Chapter 5

Conclusions and future work

This thesis has investigated the topic of optimization of scientific codes based on unstructured datasets in heterogeneous systems. It has analyzed three problems that were found to be hindering the appearance of automated approaches. The thesis is better understood in the global context of the projects that it is part of. These projects attempt to result in a series of tools that allow efficient execution of unstructured datasets in HPC clusters with FPGA accelerator boards. There is little previous work in this subject, partly due to the high computing power, memory and communication demands of this type of applications, and the tremendous effort required to implement them in reconfigurable hardware compared to CPUs or GPGPU. Because of this, there are still multiple problems to be solved. This work has investigated three of them, and given solutions that could be easily integrated in a complete parallelization tool.

5.1 Summary of the conclusions of the Thesis

Each chapter of the thesis has ended with a section that collected in detail the conclusions derived from the work described in the chapter. Here, a summary of the major conclusions of the thesis is provided.

Chapter 2 presented a series of techniques to improve memory transfers between the FPGA and other parts of the system, mainly general-purpose CPUs. The limited bandwidth of the FPGAs makes these techniques very valuable in order to achieve maximum performance in the kernels assigned to the reconfigurable hardware, as they allow stream processing of the mesh with sequential memory access, which maximizes performance in FPGAs. We have also surveyed in great detail the problem of seed selection in traversal and sorting algorithms, and its effect on memory locality. The application of seed selection algorithms allows us to obtain an extra edge on similar bandwidth reduction algorithms.

A combination of compression and encoding techniques has been proposed for general unstructured meshes. We have also developed a method to analyze the performance of the hardware required to implement these approaches. Experimental evidence showed that the proposed techniques can reduce the bandwidth needed to approximately 30% of that required before the optimizations.

Chapter 3 dealt with the problem of automated functional partitioning. In order to automate the development of concurrent programs distributed across different parts of a HPC system, the compiler must know how to group parts of the code into partitions, and map these into different processing elements. We have proposed an algorithm that allows the compiler to cluster different parts of the code according to the processing element they have been assigned to, and insert the communications required to mimic the control flow of the original application. The algorithm has been implemented on top of LLVM, a production-grade optimizing compiler, and it is capable of automatically distributing the code of an application across multiple processors while resulting in the same runtime behaviour. It works on the internal representation used by the compiler, making it applicable to multiple programming languages and processor architectures. Additionally, a communication library has been developed. The experiments showed that the library introduces very little overhead compared to existing communication approaches such as various implementations of Remote Procedure Calls.

Chapter 4 brought the problem of parallelization automation to a level closer to the mesh-based applications discussed in Chapter 2. Again, it uses a compiler to partition a program and distribute it across a parallel architecture, in this case the ROSE compiler infrastructure, and more specifically IEC, which already implements data partitioning techniques in order to automate parallelization. The computational kernels of scientific simulations based on unstructured meshes typically consist of an outer loop with a sequence of loops inside. The compiler is capable of inserting a *skeleton* of the code before the kernel so that data dependencies can be analyzed at runtime before the kernel, using the *inspector-executor* paradigm. These dependencies can be automatically translated into communications between loops in the kernel by means of a communication library, so that different loops can be put in a pipeline. Code partitioning in this manner is compatible with the more traditional data partitioning, catering for a greater flexibility. Although the obtained results showed that the overhead introduced by this approach heavily penalize overall performance, the work proves that the approach is feasible and obtains the same functional behaviour. Also, the little overhead inserted by the inspector and the executor suggests that the single major problem is the communication overhead. Furthermore, a series of improvements to the communication library have been proposed. The initial estimation of the performance uplift that could be obtained with these improvements suggests that the approach could be viable.

In Chapter 1, we enumerated four challenges: memory bandwidth limitations, irregular data accesses, code transformations for distributed systems and automated insertion of communications. These challenges glue the different chapters together. As an example, the CFD application used throughout the text has long loops with computationally expensive sections in some of them. These expensive loops could be put in a pipeline using the techniques proposed in Chapter 4. Additionally, if only part of the loop is computationally intensive or more suitable for an FPGA, or the loop could not fit in the FPGA, it could be further split by using the methods suggested in Chapter 3. At the same time, the data transfer requirements of the code assigned to the FPGA can be further optimized using the streaming techniques introduced in Chapter 2. We claim that these techniques put together will improve the current state of the art when evaluated against the four challenges mentioned above. Integrating all these techniques into a single compiler infrastructure would require a considerable effort. A complete toolchain would require solving the problem of cost estimation for different partitioning solutions at a minimum. Therefore, it is out of the scope of this work.

5.2 Future lines of work

While the ultimate objective of the work is to create a fully-automated toolchain for heterogeneous systems with FPGA accelerators, this objective is too ambitious for a single thesis. There are still too many problems to be solved, and the state of existing tools and techniques must be greatly improved in order to make this objective a reality. Furthermore, nowadays it is still unclear whether full automation of application parallelization can be achieved even if FPGA accelerators are removed from the problem and only general-purpose CPUs are considered. The survey of previous related works has shown several examples of the challenges involved.

Several proposals for future work have been given in the conclusion sections of the previous chapters. Additionally, in this section, we give a series of ideas for future work that could follow from the work already done in this thesis. These proposals have enough entity to constitute separate lines of work.

One of the problems that has been left out of this work is the extension of the streaming approach to be used in a cluster with multiple FPGAs. So far, we have considered that the complete mesh can be processed in a single FPGA. If multiple ones were to be used, the format of the streamed mesh should consider the required communications between accelerators and how these affect the vertex windows. Being able to partition the mesh would further improve the scalability of the solution to larger distributed systems and gigantic meshes where the memory inside one accelerator will not be able to hold the entire mesh.

Although we have given two different approaches to partitioning the specification in a compiler, the decision of how to create the partitions is still given manually in Chapter 3 or in a simple and inflexible way in Chapter 4. An estimation algorithm, possibly driven by heuristics, should be able to guide the partitioning to achieve better performance results and further automate the process. Due to the technical complexity of creating a suitable algorithm, its development was out of the scope of this work from the beginning. For practical reasons the two automation approaches have been implemented on top of two different compilers. One desirable line of work would be to unite them in a single toolchain, probably LLVM due to its production-grade quality, popularity, and the strong community behind it. Additionally, some ongoing projects from the Integrated Systems Laboratory (LSI), at the *Universidad Politécnica de Madrid*, are related to the work proposed in this Thesis. For example, a compiler-based approach for the compilation of computational kernels in FPGAs is being developed at the time of writing. Integrating these works under a common infrastructure would make the task of porting existing applications to systems based on FPGAs easier.

Finally, since the automation techniques presented in the later chapters have only been tested in homogeneous systems, they have been implemented with some technologies not currently applicable to FPGAs. A hardware implementation of a subset of the MPI standard would allow using the developed libraries with relatively low effort. This would not only be beneficial for the current work, but could also be applied to other applications to be ported to FPGAs.

Bibliography

- [AAA⁺02] N. R. Adiga, et al. An Overview of the BlueGene/L Supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 60–60, November 2002.
- [ABDS02] V. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *Journal of Parallel and Distributed Computing*, 62(3):393–426, March 2002.
- [ADK⁺09] K. Asanovic, et al. A view of the parallel computing landscape. *Commun. ACM*, 52(10):5667, October 2009.
- [AFMP95] F. André, M. Le Fur, Y. Maho, and J. Pazat. The Pandore data-parallel compiler and its portable runtime. In *High-Performance Computing and Networking*, Lecture Notes in Computer Science, pages 176–183. Springer Berlin Heidelberg, May 1995.
- [AFP⁺11] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proc. 19th ACM/SIGDA Int’l Symp. Field Programmable Gate Arrays*, pages 25–28, 2011.
- [AIO⁺11] T. Akamine, K. Inakagata, Y. Osana, N. Fujita, and H. Amano. An implementation of out-of-order execution system for acceleration of computational fluid dynamics on FPGAs. *ACM SIGARCH Computer Architecture News*, 39(4):50–55, 2011.
- [AIO⁺12] T. Akamine, K. Inakagata, Y. Osana, N. Fujita, and H. Amano. Reconfigurable out-of-order mechanism generator for unstructured grid computation in computational fluid dynamics. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 136–142. IEEE, 2012.

- [BBB⁺06] F. Bertrand, R. Bramley, D. E. Bernholdt, J. A. Kohl, A. Sussman, J. W. Larson, and K. B. Damevski. Data redistribution and remote method invocation for coupled components. *J. Parallel Distrib. Comput.*, 66(7):931–946, July 2006.
- [BBG⁺96] H. Bao, J. Bielik, O. Ghattas, L. F. Kallivokas, D. R. O’Hallaron, J. R. Shewchuk, and J. Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing ’96, Washington, DC, USA, 1996. IEEE Computer Society.
- [BC11] P. Barrio and C. Carreras. Mesh traversal and sorting for efficient memory usage in scientific codes. In *Proc. of the 30th IEEE Int’l Performance Computing and Communications Conf. (IPCCC)*, Nov. 2011.
- [BC12] P. Barrio and C. Carreras. Turning Control Flow Graphs into Function Call Graphs: Transformation of Partitioned Codes for Execution in Heterogeneous Architectures. In *2012 European Conference on the LLVM Compiler Infrastructure*, <http://llvm.org/devmtg/2012-04-12>, 2012.
- [BCG⁺95] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, October 1995.
- [BCK⁺14] J. Bezanson, J. Chen, S. Karpinski, V. Shah, and A. Edelman. Array Operators Using Multiple Dispatch: A Design Methodology for Array Implementations in Dynamic Languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY’14, pages 56:56–56:61, New York, NY, USA, 2014. ACM.
- [BCL⁺14] P. Barrio, C. Carreras, J. López, O. Robles, R. Jevtic, and R. Sierra. Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes. *Journal of Systems Architecture*, 60(7):579–591, August 2014.
- [BCS⁺] P. Barrio, C. Carreras, R. Sierra, J. A. López, G. Caffarena, E. Sedano, J.A. Fernández, and R. Jevtic. Systems and methods for improving the execution of computational algorithms. Patent

- number: US 9,311,433, Country: US, Owners: Airbus España and Universidad Politécnica de Madrid.
- [BCS⁺12] P. Barrio, C. Carreras, R. Sierra, T. Kenter, and C. Plessl. Turning control flow graphs into function calls: Code generation for heterogeneous architectures. In *2012 Int'l Conf. on High Performance Computing and Simulation (HPCS)*, pages 559–565, July 2012.
- [BDH⁺10] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, January 2010.
- [Ber16] Berten. GPU vs FPGA Performance Comparison. Technical report, Berten Digital Signal Processing, 05 2016.
- [BFN13] D. Buttlar, J. Farrell, and B. Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, Inc., January 2013.
- [BGA12] A. Baldassin, F. Goldstein, and R. Azevedo. A transactional runtime system for the Cell/BE architecture. *J. Parallel Distrib. Comput.*, 72(12):1535–1546, December 2012.
- [Blo92] J. Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc., 1992.
- [BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):3959, February 1984.
- [Bon02] D. Bonachea. Gasnet specification v1.1. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.
- [BRV99] M. Brune, A. Reinefeld, and J. Varnholt. A resource description environment for distributed computing systems. In *The Eighth International Symposium on High Performance Distributed Computing, 1999. Proceedings*, pages 279–286. IEEE, 1999.
- [BSY⁺10] K. Bertels, et al. HArtes: Hardware-software codesign for heterogeneous multicore platforms. *IEEE Micro*, 30(5):88–97, October 2010.
- [Cac] Cachegrind: a cache and branch-prediction profiler. <http://valgrind.org/docs/manual/cg-manual.html>.

- [CBD⁺07] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and Lee Ann Riesen. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–11, March 2007.
- [CCE99] C. Chang, G. Czajkowski, and T. Von Eicken. MRPC: a high performance RPC system for MPMD parallel computing. *Softw. Pract. Expert.*, 29(1):4366, January 1999.
- [CCLF10] G. Caffarena, C. Carreras, J. A. López, and A. Fernández. SQNR estimation of fixed-point DSP algorithms. *EURASIP J. Adv. Signal Process*, 2010:21:121:12, February 2010.
- [CCLH10] B. Cope, P.Y.K. Cheung, W. Luk, and L. Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Trans. on Computers*, 59(4):433–448, Apr. 2010.
- [CDD⁺10] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload automating code migration to heterogeneous multicore systems. In *High Perf. Embedded Archs. and Compilers*, volume 5952, pages 337–352. Springer Berlin Heidelberg, 2010.
- [CDW10] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):13:113:65, June 2010.
- [CGS⁺05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.
- [CHM11] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proc. of the 19th ACM/SIGDA Int’l Symp. on Field Programmable Gate Arrays*, pages 97–106, 2011.
- [CJdP08] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

- [CLR⁺01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.
- [CLS⁺08] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Symp. on Application Specific Processors (SASP'08)*, pages 101–107, Jun. 2008.
- [CSS⁺13] H. Chen, L. Shi, J. Sun, K. Li, and L. He. A fast RPC system for virtual machines. *IEEE Trans. Parallel Distrib. Syst.*, 24(7):1267–1276, 2013.
- [Din03] P. C. Diniz. A Compiler Approach to Performance Prediction Using Empirical-Based Modeling. In *Computational Science ICCS 2003*, pages 916–925. Springer, Berlin, Heidelberg, June 2003.
- [DMM⁺10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10*, page 6374, New York, NY, USA, 2010. ACM.
- [DMnCN12] J. Diaz, C. Muñoz Caro, and A. Nio. A Survey of Parallel Programming Models and Tools in Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, August 2012.
- [DSS07] R. McGregor D. Szczerba and G. Székely. High quality surface mesh generation for multi-physics bio-medical simulations. *Lecture Notes in Computer Science, Springer*, 4487:906–913, 2007.
- [DVCS08] H. Devos, J. Van Campenhout, and D. Stroobandt. Building an application-specific memory hierarchy on FPGA. In *2nd HiPEAC Workshop on Reconfigurable Computing*, page 53, 2008.
- [EAMEG11] E. El-Araby, S. G. Merchant, and T. El-Ghazawi. A framework for evaluating high-level design methodologies for high-performance reconfigurable computers. *IEEE Trans. on Parallel and Distributed Sys.*, 22:33–45, 2011.
- [Edw06] S.A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers*, 23(5):375–386, 2006.

- [EFF00] V. Engelson, D. Fritzson, and P. Fritzson. Lossless compression of high-volume numerical data from simulations. In *Proceedings of the Conference on Data Compression*, DCC '00, page 574, Washington, DC, USA, 2000. IEEE Computer Society.
- [EGS06] T. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
- [ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proc. of IEEE Symp. on Visualization*, pages 319–326, 1996.
- [Eve79] G. C. Everstine. A comparasion of three resequencing algorithms for the reduction of matrix profile and wavefront. *International Journal for Numerical Methods in Engineering*, 14(6):837853, 1979.
- [Fei00] U. Feige. Coping with the NP-Hardness of the graph bandwidth problem. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, SWAT '00, page 1019, London, UK, UK, 2000. Springer-Verlag.
- [FHK⁺06] K. Fatahalian, et al. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
- [GGF⁺] T. Gerhold, M. Galle, O. Friedrich, J. Evans, T. Gerhold, M. Galle, O. Friedrich, and J. Evans. Calculation of complex three-dimensional configurations employing the DLR-tau-code. In *35th Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics. DOI: 10.2514/6.1997-167.
- [GGK01] C. Gotsman, S. Gumhold, and L. Kobbelt. Simplification and compression of 3D meshes. In *European Summer School on Principles of Multiresolution in Geometric Modelling*, page 44, Munich, 2001.
- [GGL12] T. Grosser, A. Groesslinger, and C. Lengauer. Polly performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, December 2012.
- [GHK⁺12] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Newnes, December 2012.

- [Gid12] Gidel. *Gidel PROCStarIII*, 2012.
- [GR09a] C. Geuzaine and J. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):13091331, 2009.
- [GR09b] T. Gurung and J. Rossignac. SOT: compact representation for tetrahedral meshes. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, page 7988, New York, NY, USA, 2009. ACM.
- [gra16] November 2016 — graph 500. http://www.graph500.org/results_nov_2016, November 2016.
- [GS99] M. B. Gokhale and J. M. Stone. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, page 63, 1999.
- [Gus88] J. L. Gustafson. Reevaluating Amdahl’s law. *Commun. ACM*, 31(5):532–533, May 1988.
- [GY06] J. Gross and J. Yellen. *Graph Theory and its Applications*. Chapman & Hall/CRC, 2006.
- [HAA⁺96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.
- [Har94] F. Harary. *Graph Theory*. Westview Press, October 1994.
- [IG03] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graph.*, 22(3):935942, July 2003.
- [IL05] M. Isenburg and P. Lindstrom. Streaming meshes. In *IEEE Visualization, 2005. VIS 05*, pages 231 – 238, October 2005.
- [ILGS03] M. Isenburg, P. Lindstrom, S. Gumhold, and J Snoeyink. Large mesh simplification using processing sequences. In *Proc. of IEEE Symp. on Visualization*, pages 465–472, 2003.

- [JNH16] B. Janen, M. Naserddin, and M. Hübner. A Hardware/Software Co-Design Approach for Control Applications with Static Real-Time Reallocation. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 241–246, May 2016.
- [JVTB17] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg. HW/SW Co-design Toolset for Customization of Exposed Datapath Processors. In Waqar Hussain, Jari Nurmi, Jouni Isoaho, and Fabio Garzia, editors, *Computing Platforms for Software-Defined Radio*, pages 147–164. Springer International Publishing, 2017.
- [JWP33] E. N. Jacobs, K. E Ward, and R. M. Pinkerton. The characteristics of 78 related airfoil sections from tests in the variable-density wind tunnel. Technical report, 1933.
- [KC11] A. Ketterlin and P. Clauss. Efficient memory tracing by program skeletonization. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 97–106, April 2011.
- [KD09] N. Kapre and A. DeHon. Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, Cell, and multi-core processors. In *Int’l Conf. Field Programmable Logic and Applications (FPL’09)*, pages 65–72, Aug. 2009.
- [KdA07] A. Khamayseh and V. de Almeida. Adaptive hybrid mesh refinement for multiphysics applications. *J. Physics: Conf. Series*, 78, 2007.
- [KDW10] S. Kestur, J.D. Davies, and O. Williams. BLAS comparison on FPGA, CPU and GPU. In *IEEE Annual Symp. on VLSI*, pages 288–293, Jul. 2010.
- [KK98] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998.
- [Kor95] R. Korf. Space-efficient search algorithms. *ACM Computing Surveys*, 27(3):337–339, 1995.
- [KPN⁺09] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and Xiaojun Yang. FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *International Symposium on*

- Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09*, pages 149–156, July 2009.
- [KSY10] K. Katahira, K. Sano, and S. Yamamoto. FPGA-based lossless compressors of floating-point data streams to enhance memory bandwidth. In *2010 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 246–253, 2010.
- [KZC16] W. Kemmerer, W. Zuo, and D. Chen. Parallel Code-specific CPU Simulation with Dynamic Phase Convergence Modeling for HW/SW Co-design. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD '16*, pages 79:1–79:8, New York, NY, USA, 2016. ACM.
- [LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pages 75–86. IEEE, March 2004.
- [LD98] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(1112):677–689, December 1998.
- [Lee10] J. Lee. *Introduction to topological manifolds*, volume 202. Springer, 2010.
- [LPS⁺15] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis. Improving Graph Partitioning for Modern Graphs and Architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA³ '15*, pages 14:1–14:4, New York, NY, USA, 2015. ACM.
- [LQLM12] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee. ROSE::FTTransform - A source-to-source translation framework for exascale fault-tolerance research. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.
- [LQVP10] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*,

- number 5898 in Lecture Notes in Computer Science, pages 308–322. Springer Berlin Heidelberg, January 2010.
- [LQWP09] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas. Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore. In Matthias S. Mller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, Lecture Notes in Computer Science, pages 28–41. Springer Berlin Heidelberg, June 2009.
- [LS04] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):9751028, November 2004.
- [LYS⁺13] C. Liao, Y. Yan, B. R. Supinski, D. J. Quinlan, and B. Chapman. Early Experiences with the OpenMP Accelerator Model. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Mller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, Lecture Notes in Computer Science, pages 84–98. Springer Berlin Heidelberg, September 2013.
- [Mag] Magerit - nodos power7 / intel xeon. <http://www.cesvima.upm.es/services/hpc/magerit>.
- [MC98] T. Mitra and T. Chiueh. A Breadth-First Approach to Efficient Mesh Traversal. In *Proc of the 1998 SIGGRAPH Workshop*, pages 31–38, 1998.
- [MC02] T. Mitra and T. Chiueh. An FPGA Implementation of Triangle Mesh Decompression. In *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines, FCCM’02*, pages 22–31, 2002.
- [MCT96] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424453, July 1996.
- [MOS16] I. Mhadhbi, S. B. Othman, and S. B. Saoud. A Comprehensive Survey on Hardware/Software Partitioning Process in Co-Design. *International Journal of Computer Science and Information Security; Pittsburgh*, 14(3):263–279, 2016.
- [MPI] MPICH — High-performance Portable MPI. <http://www.mpich.org/>.

- [Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 7. Morgan Kaufmann, August 1997.
- [MVA] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. <http://mvapich.cse.ohio-state.edu>.
- [MWBA10] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.
- [NCC13] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic Species: A Classification of Affine Loop Nests for Parallel Programming. *ACM Trans. Archit. Code Optim.*, 9(4):40:1–40:25, January 2013.
- [NFMM13] M. Naylor, P.J. Fox, A.T. Markettos, and S.W. Moore. Managing the FPGA memory wall: Custom computing or vector processing? In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2013.
- [NNH⁺12] Z. Nagy, C. Nemes, A. Hiba, A. Kiss, A. Csik, and P. Szolgay. FPGA based acceleration of computational fluid flow simulation on unstructured mesh geometry. In *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 128–135, 2012.
- [NR98] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.
- [NS05] P. Nalabalapu and R. Sass. Bandwidth management with a reconfigurable data cache. In *Proc. of the 19th IEEE Int’l Parallel and Distributed Processing Symp.*, Apr. 2005.
- [Ope] OpenMPI: Open source high performance computing. <http://www.open-mpi.org>.
- [Ozt11] O. Ozturk. Data locality and parallelism optimization using a constraint-based approach. *J. Parallel Distrib. Comput.*, 71(2):280–287, February 2011.
- [PAB⁺05] D. Pham, et al. The design and implementation of a first-generation CELL processor. In *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, pages 184–592 Vol. 1, February 2005.

- [PCB⁺06] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. GRAPHITE: polyhedral analyses and optimizations for GCC. In *Proceedings of the 4th GCC Developers Summit*, June 2006.
- [PD11] J. Park and P. C. Diniz. Data reorganization and prefetching of pointer-based data structures. *IEEE Design and Test of Computers*, 28:38–47, 2011.
- [Pen02] O. Pentakalos. *An Introduction to the Infiniband Architecture*. O’Reilly Media, Inc., 2002.
- [PKB03] P. M. Papadopoulos, M. J. Katz, and G. Bruno. NPACI rocks: tools and techniques for easily deploying manageable linux clusters. *Concurr. & Comp.: Pract. Exper.*, 15(7-8):707725, 2003.
- [Pou] L. Pouchet. PolyBench homepage. <http://www.cse.ohio-state.edu/~pouchet/software/polybench/>.
- [Qui00] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, June 2000.
- [RDLQ12] S. Royuela, A. Durán, C. Liao, and D. J. Quinlan. Auto-scoping for OpenMP Tasks. In Barbara M. Chapman, Federico Massaioli, Matthias S. Mller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, Lecture Notes in Computer Science, pages 29–43. Springer Berlin Heidelberg, June 2012. DOI: 10.1007/978-3-642-30961-8_3.
- [Rei10] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O’Reilly Media, Inc., March 2010.
- [REP⁺12] M. Ravishankar, J. Eisenlohr, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, pages 72:1–72:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [Ros99] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Visualization and Computer Graphics*, 5(1):47–61, 1999.

- [RT98] G. Rivera and C. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 3849, New York, NY, USA, 1998. ACM.
- [SAA⁺08] R. Susukita, et al. Performance prediction of large-scale parallel system and application using macro-level simulation. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, page 20:120:9, Piscataway, NJ, USA, 2008. IEEE Press.
- [SAK07] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 2007.
- [Sal04] D. Salomon. *Data Compression: The Complete Reference*. Springer, 2004.
- [Scaa] The scala programming language. <http://www.scala-lang.org/>.
- [Scab] Graph for scala. <http://www.scala-graph.org/>.
- [Sch10] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, September 2010.
- [SCMB90] J. Saltz, K. Crowley, R. Michandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.
- [SGS10] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.
- [Shi06] K. Shimada. Current trends and issues in automatic mesh generation. *Computer-Aided Design and Appl.*, 3(6):741–758, 2006.
- [SHY14] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, March 2014.
- [SHZRM⁺16] J. C. Salinas-Hilburg, M. Zapater, J. L. Risco-Martn, J. M. Moya, and J. L. Ayala. Unsupervised Power Modeling of Co-allocated Workloads for Energy Efficiency in Data Centers. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*,

- DATE '16, pages 1345–1350, San Jose, CA, USA, 2016. EDA Consortium.
- [SK10] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, July 2010.
- [SKH09] L. Scandolo, C. Kunz, and M. Hermenegildo. Program Parallelization Using Synchronized Pipelining. In *Logic-Based Program Synthesis and Transformation*, pages 173–187. Springer, Berlin, Heidelberg, September 2009.
- [SKS⁺07] R. Sass, W. V Kritikos, A. G Schmidt, S. Beeravolu, and P. Beeraka. Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-Based petascale computing. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM 2007*, pages 127–140. IEEE, April 2007.
- [SLC12] E. Sedano, J.A. López, and C. Carreras. A fast interpolative wordlength optimization method for DSP systems. In *2012 VIII Southern Conf. on Programmable Logic (SPL)*, pages 1–6, March 2012.
- [SLC⁺16] M M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Computing*, 53:32–57, April 2016.
- [SLL01] J. G. Siek, L. Lee, and A. Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.
- [SM06] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.
- [SNM⁺02] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: a remote procedure call API for grid computing. In Manish Parashar, editor, *Grid Computing GRID 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer Berlin / Heidelberg, 2002.
- [SPE] SPEC CPU2000 benchmark. <https://www.spec.org/cpu2000>.

- [SQJ11] J. Shalf, D. Quinlan, and C. Janssen. Rethinking Hardware-Software codesign for exascale systems. *Computer*, 44(11):22–30, November 2011.
- [SRSLB⁺11] D. Sánchez-Román, G. Sutter, S. López-Buedo, I. González, F. Gómez-Arribas, and J. Aracil. An Euler solver accelerator in FPGA for computational fluid dynamics applications. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 149–154. IEEE, 2011.
- [SRW⁺13] M. J. Sottile, C. E. Rasmussen, W. N. Weseloh, R. W. Robey, D. Quinlan, and J. Overbey. ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming. *Int. J. Comput. Sci. Eng.*, 8(1):47–57, February 2013.
- [SSX08] S. Sodhi, J. Subhlok, and Q. Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.
- [TAU] Tau - tuning and analysis utilities. <https://ww.cs.uoregon.edu/research/tau/home.php>.
- [TB10] X. Tian and K. Benkrid. High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU. *ACM Trans. Reconfigurable Technol. Syst.*, 3:26:1–26:22, Nov. 2010.
- [The12] The Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), 2012.
- [TNS⁺03] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoaka. Ninf-g: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 2003.
- [top13] June 2013 — Top500 Supercomputer Sites. <https://www.top500.org/lists/2016/11/>, June 2013.
- [Tse89] P. Tseng. A parallelizing compiler for distributed-memory parallel computers. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA (USA), January 1989.
- [vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Annu. Int’l Symp. Comput. Arch.*, ISCA ’92, page 256266, New York, NY, USA, 1992. ACM.

- [WDS⁺95] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandan. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.
- [WM95] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):2024, March 1995.
- [WMH⁺15] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-Level Language Tools for Reconfigurable Computing. *Proceedings of the IEEE*, 103(3):390–408, March 2015.
- [WQZ⁺13] H. Wei, M. Qin, W. Zhang, J. Yu, D. Fan, and G. R. Gao. StreamTMC: Stream compilation for tiled multi-core architectures. *J. Parallel Distrib. Comput.*, 73(4):484–494, April 2013.
- [WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the JavaTM system. In *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, page 1717, Berkeley, CA, USA, 1996. USENIX Association.
- [XSAH10] J. Xu, N. Subramanian, A. Alessio, and S. Hauck. Impulse C vs. VHDL for Accelerating Tomographic Reconstruction. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 171–174, 2010.
- [YGR16] S. Yousuf and A. Gordon-Ross. An Automated Hardware/Software Co-Design Flow for Partially Reconfigurable FPGAs. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 30–35, July 2016.
- [ZA07] P. Zhao and J. N. Amaral. Ablego: a function outlining and partial inlining framework. *Software: Practice and Experience*, 37(5):465491, 2007.
- [ZH06] R. Zhou and E. Hansen. A breadth-first approach to memory-efficient graph search. In *Proc. of National Conf. on Artificial Intelligence*, volume 21, pages 1695–1699, 2006.
- [ZTM⁺15] M. Zapater, A. Turk, J. M. Moya, J. L. Ayala, and A. K. Coskun. Dynamic workload and cooling management in high-efficiency data centers. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, December 2015.

Glossary

APGAS Asynchronous PGAS. 65

API Application Programming Interface. 62

AST Abstract Syntax Tree. 63

BB Basic Block. xiii, 61–63, 69–75, 84, 101

BFS Breadth-First Search. 35, 70

BLAS Basic Linear Algebra Subroutines. 3

CBE Cell Broadband Engine. 3

CFD Computational Fluid Dynamics. v, vi, 3, 13, 98, 120, 131

CFG Control Flow Graph. 61–63, 73, 85

cluster Group of connected computers that can be viewed as a single system. 1

CPU Central Processing Unit. 3, 5, 14, 17, 18, 20, 42, 52, 53, 57, 60, 62, 69, 70, 100, 101, 108, 129, 131

FLOPS Floating-point Operations per Second. 1, 120

FPGA Field-Programmable Gate Array. v, vi, 1–3, 5–7, 61, 95, 99, 108, 129, 131, 132

GPGPU General-Purpose Graphics Processing Unit. vi, 1–4, 6, 8, 57, 61, 129

GPP General-Purpose Processor. 6

HPC High Performance Computing. v, vi, 1, 2, 5, 64, 82, 97, 99, 101, 126, 129, 130

LOC Lines Of Code. 2, 3

MPI Message Passing Interface. 8, 78–81, 89–91, 94, 107, 109, 113, 116–118, 123, 124, 126, 132

MPMD Multiple Program Multiple Data. 64

MPP Massively Parallel Processing. 5

NACA National Advisory Committee for Aeronautics. 13, 30

PDE Partial Differential Equation. 3, 8

PGAS Partitioned-Global Address Space. 8, 65

RBB Remote Block Branch. ii, xiii, 62, 68, 74–82, 84, 85, 88–94

RMI Remote Method Invocation. 64

RPC Remote Procedure Call. 7, 10, 62, 64, 74, 75

RSD Resource and Service Description. xii, 68

soft-core Hardware design wholly implemented using logic synthesis. 6

SPMD Single Program Multiple Data. 64

SSA Static Single Assignment. 62, 63, 75, 78

UPC Unified Parallel C. 8