

Modèle de programmation de haut niveau pour la parallélisation expicite et automatique : application aux architectures multicoeurs

Nader Khammassi

▶ **To cite this version:**

Nader Khammassi. Modèle de programmation de haut niveau pour la parallélisation expicite et automatique : application aux architectures multicoeurs. Autre [cs.OH]. Université de Bretagne Sud, 2014. Français. NNT : 2014LORIS350 . tel-01207434

HAL Id: tel-01207434
https://theses.hal.science/tel-01207434

Submitted on 30 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE / UNIVERSITÉ DE BRETAGNE SUD
sous le sceau de l'Université européenne de Bretagne

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : Informatique

École Doctorale SICMA

Présentée par
Nader KHAMMASSI

Préparée au
Laboratoire Lab-STICC UMR CNRS 6285
École Nationale Supérieure de Techniques Avancées
ENSTA Bretagne

High-Level Structured Programming Models For Explicit and Automatic Parallelization on Multicore Architectures

Thèse soutenue le 5 Décembre 2014 ,
devant la commission d'examen composée de :

M. Koen Bertels
Professor, Delft University of Technology / Rapporteur

M. Frédéric Loulergue
Professor, Université d'Orléans / Rapporteur

M. Abdoulaye Gamatié
CNRS Research Scientist / Examinateur

M. Jean-Philippe Diguet
Lab-STICC CNRS / Directeur de thèse

M. Jean-Christophe Le Lann
Lab-STICC CNRS / Co-directeur de thèse

M. Alexandre Skrzyniarz
Thalès Airborne Systems / Co-encadrant de thèse

Résumé

La prolifération des architectures multi-coeurs est source d'une pression importante pour les developpeurs, qui doivent chercher à paralléliser leurs applications de manière à profiter au mieux de ces plateformes. Malheureusement, les modèles de programmation de bas niveau amplifient les difficultés inhérentes à la conception d'applications complexes et parallèles. Il existe donc une attente pour des modèles de programmation de plus haut niveau, qui puissent simplifier la vie des programmeurs de manière significative, tout en proposant des abstractions suffisantes pour absorber l'hétérogénéité des architectures matérielles.

Contrairement à une multitude de modèles de programmation parallèle qui introduisent de nouveaux langages, annotations ou étendent des langages existants et requièrent donc des compilateurs spécialisés, nous exploitons ici le potentiel du language C++ standard et traditionnel. En particulier nous avons recours à ses capacités en terme de meta-programmation, afin de fournir au programmeur une interface de programmation parallèle simple et directe. Cette interface autorise le programmeur à exprimer le parallélisme de son application au prix d'une altération négligeable du code séquentiel initial. Un runtime intelligent se charge d'extraire toute information relative aux dépendances de données entre tâches, ainsi que celles relatives à l'ordonnancement. Nous montrons comment ce runtime est à même d'exploiter ces informations dans le but de détecter et protéger les données partagées, puis réaliser un ordonnancement prenant en compte les particularités des caches.

L'implémentation initiale de notre modèle de programmation est une librairie C++ pure appelée XPU. XPU est conÃğue dans le but de faciliter l'explicitation, par le programmeur, du parallélisme applicatif. Une seconde réalisation appelée FATMA doit être considérée comme une extension d'XPU qui permet une détection automatique des dépendances dans une séquence de tâches : il s'agit donc de parallélisation automatique, sans recours à quelque outil que se soit, excepté un compilateur C++ standard. Afin de démontrer le potentiel de notre approche, nous utilisons ces deux outils –XPU et FATMA– pour paralléliser des problèmes populaires, ainsi que des applications industrielles réelles. Nous montrons qu'en dépit de leur abstraction élevée, nos modèles de programmation présentent des performances comparables à des modèles de programmation de bas-niveau, et offrent un meilleur compromis productivité-performance.

Abstract

The continuous proliferation of multicore architectures has placed developers under great pressure to parallelize their applications accordingly with what such platforms can offer. Unfortunately, traditional low-level programming models exacerbate the difficulties of building large and complex parallel applications. High-level parallel programming models are in high-demand as they reduce the burdens on programmers significantly and provide enough abstraction to accommodate hardware heterogeneity. In this thesis, we propose a flexible parallelization methodology, and we introduce a new task-based parallel programming model designed to provide high productivity and expressiveness without sacrificing performance. Our programming model aims to ease expression of both sequential execution and several types of parallelism including task, data and pipeline parallelism at different granularity levels to form a structured homogeneous programming model.

Contrary to many parallel programming models which introduce new languages, compiler annotations or extend existing languages and thus require specialized compilers, extra-hardware or virtual machines..., we exploit the potential of the traditional standard C++ language and particularly its meta-programming capabilities to provide a light-weight and smart parallel programming interface. This programming interface enable programmer to express parallelism at the cost of a little amount of extra-code while reuse its legacy sequential code almost without any alteration. An intelligent run-time system is able to extract transparently many information on task-data dependencies and ordering. We show how the run-time system can exploit these valuable information to detect and protect shared data automatically and perform cache-aware scheduling.

The initial implementation of our programming model is a pure C++ library named "XPU" and is designed for explicit parallelism specification. A second implementation named "FATMA" extends XPU and exploits the transparent task dependencies extraction feature to provide automatic parallelization of a given sequence of tasks without need to any specific tool apart a standard C++ compiler. In order to demonstrate the potential of our approach, we use both of the explicit and automatic parallel programming models to parallelize popular problems as well as real industrial applications. We show that despite its high abstraction, our programming models provide comparable performances to lower-level programming models and offers a better productivity-performance tradeoff.

N d'ordre : 000000000

Université de Bretagne Sud

Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES

Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70

Dedication

It is with my deepest gratitude and warmest affection that I dedicate this thesis to:

My beloved mother Fatma for making me be who I am, for her constant support and prays of day and night, for her great sacrifices and unconditional love.

My father Hammadi for earning an honest living for us, for his love and continuous support.

My sweet sister Nadia for her love and constant encouragement.

My grand parents Zina and Ahmed for raising me and for everything they have done to make me happy.

"... they gave up and told me that I am going straight to the wall. So I went straight to the wall ! I hit it so hard for a long time despite pain and loneliness, I hit it until I broke through it, I broke it with an unbreakable will and I will smash any other wall that might stand in front of me, between me an my dreams, no matter how long it would take or cost.
Nothing is impossible, it is all about faith, will and patience, just want it then do it. Don't let anybody tells you who you are or can be, just show them who you are and how great you can be."

Warrior Spirit, 642

Acknowledgments

First and above all, my praises to God for giving me the patience and the perseverance to endure the long PhD journey. Only by His grace was I able to achieve the degree requirements.

I would like to thank M. Koen Bertels, Professor at Delft University of Technology and M. Frédéric Loulergue, Professor at Université d'Orléans, for reviewing this thesis and providing their valuable advises.

I would like to thank M. Abdoulaye Gamatié, CNRS Research Scientist, for being member of the jury.

I would like to express my gratitude to my supervisor Jean-Christophe Le Lann for his constant support, friendly encouragement, great advices and guidance. I thank him for being a friend with great human qualities before being a technical supervisor.

I would like to thank Jean-Philppe Diguet for accepting to be my thesis director, for his great advises and support.

I would like to thank Alexandre Skrzyniarz for his constant support since my first works at Thalès Airborne Systems then during all my thesis. I thank him for supporting my choice to pursue my PhD and allowing me to work on my researches at Thalès.

I would like also to thank Vincent Verbeque, Gilles Le Pluart for allowing me to work on this thesis at Thalès Airborne Systems.

I feel fortunate to work at ENSTA Bretagne, a great research environment. I am grateful to all my lab mates and friends.

I would like to thank my friend and office mate Mohammed Ben Hammouda for his continuous encouragement and his kindness.

A very special thanks to Joel Champeau for his friendly support.

Above all, I would like to express my gratitude to my parents and my sister for their support, great sacrifices and love.



Contents

Contents 4

I Introduction and State Of the Art 9

1 Introduction 11
1.1 Context . . . 11
1.2 Structured Parallel Programming . . . 12
1.3 Research Questions . . . 14
1.4 Contributions of this Thesis . . . 14
1.5 Outline of the Dissertation . . . 15
1.6 Terminology . . . 16
1.7 Publications . . . 17

2 State of The Art 19
2.1 Compiler-Based Parallelization . . . 19
2.2 Language-Based Parallelization . . . 21
2.3 Discussion . . . 41
2.4 Motivation . . . 42

II Explicit Parallelism Expression : XPU 45

3 XPU Methodology and Architecture 47
3.1 Parallelization Methodology . . . 47
3.2 XPU Architecture . . . 51
3.3 The Hierarchical Task Group Graph . . . 52
3.4 Overview of the XPU Programming Model . . . 54

4 Task Definition 59
4.1 Task Definition and Programmabitlity . . . 59
4.2 PThreads Callbacks . . . 59
4.3 Threading Building Blocks Task . . . 61
4.4 XPU Task . . . 61
4.5 Design and Implementation Of The XPU Task . . . 64

5 Task Parallelism 77
5.1 Task Graph Representation . . . 77
5.2 Parallelism Expression . . . 78
5.3 Automatic Shared Data Detection and Protection . . . 81
5.4 Shared Data Detection . . . 83
5.5 Shared Data Protection with Critical Sections . . . 85
5.6 Programmability Comparison with Intel TBB . . . 86

5.7 Execution Infrastructure . . . 88
5.8 Conclusion . . . 90

6 Data Parallelism **93**
6.1 Parallel Loop . . . 94
6.2 Vectorization . . . 108
6.3 Abstract Parallel Vector Interface . . . 111
6.4 Data Parallel Applications . . . 112
6.5 Conclusion . . . 132

7 Pipeline Parallelism **133**
7.1 Pipeline Execution Pattern . . . 133
7.2 Pipeline Parallelism Expression . . . 135
7.3 Internal Design . . . 138
7.4 Run-Time System . . . 141
7.5 Application: Image Processing . . . 144
7.6 Performance . . . 147
7.7 Limitations and Future Works . . . 149
7.8 Conclusion . . . 149

III Automatic Parallelization : FATMA **151**

8 Automatic Parallelization **153**
8.1 Related Works . . . 156
8.2 FATMA Contributions . . . 157
8.3 The FATMA Parallelization Process . . . 157
8.4 Tiled Cholesky Factorization . . . 158
8.5 The FATMA Programming Interface . . . 160
8.6 Dynamic Task Dependency Graph Construction . . . 168
8.7 Static and Dynamic DAG-Driven Scheduling . . . 172
8.8 Application to Tiled Linear Algebra Algorithms . . . 175
8.9 Conclusion . . . 179

IV Applications **181**

9 Radar Signal Processing Application **183**
9.1 Algorithm Overview . . . 183
9.2 Experimental Setup . . . 186
9.3 Serial Execution . . . 187
9.4 Parallelization Methodology . . . 188
9.5 Task Parallelism Extraction . . . 189
9.6 Data Parallelism . . . 190
9.7 Pipeline Parallelism . . . 194

9.8 Conclusion . . . 196

10 Polyphase Filter Bank Processing Application 199
10.1 The Quadrature Polyphase Filter Bank Technique . . . 200
10.2 Context . . . 201
10.3 Algorithm Description . . . 202
10.4 Parallelization . . . 204
10.5 Conclusion . . . 212

11 Conclusion 213

Conclusion 213
11.1 Summary . . . 213
11.2 Limitations . . . 214
11.3 Perspectives . . . 215

A C++ Template Metaprogramming 219

C++ Template Metaprogramming 219
A.1 C++ Templates . . . 219

B Platforms Used For Testing XPU and FATMA 225

Platforms Used For Testing XPU and FATMA 225
B.1 Single Multicore Processor . . . 225
B.2 Bi-Processor Platform . . . 225
B.3 Quad-Processor Platform . . . 225
B.4 The CAPARMOR Supercomputer . . . 225
B.5 The BADIANE Cluster . . . 226

Bibliography 240

List of Figures 241

List of Tables 249

Part I

Introduction and State Of the Art

1 Introduction

1.1 Context

For decades, parallel computers were synonymous of large and expensive supercomputers built by companies like IBM or CRAY. These machines were affordable only to large corporations and government laboratories. Only expert programmers with deep parallel computing knowledge were able to effectively use these systems. Low level parallel programming models were used to express parallelism and build parallel programs.

In the 1990s, three parallel programming standards grew to dominate the parallel computing landscape: the POSIX Threads (PThreads) [NBF96], the Open Multiprocessing (OpenMP) [DM98] and the Message Passing Interface (MPI) [SOW+95]. PThreads specified a set of thread and memory management primitives to build multithreaded applications. MPI and OpenMP aimed to simplify development of parallel applications by promoting portable, open standards over multiple proprietary technologies. Nevertheless, these programming models still required an indepth understanding of parallel computing.

With the rise of the Chip Multicore Processor (CMP), parallel computing hardware is becoming widely prevalent on many scales: from personal computers to embedded systems to high performance supercomputers...[BDM09] [KAG+09] [Wol09]. Unfortunately, experience and knowledge in parallel programming have not kept pace with the trend towards parallel hardware. While parallel programming is still distant from the average sequential programmers, this proliferation of multicore architectures has placed mainstream developers under a great pressure to parallelize their applications as much as possible to take advantage of these platforms.

Parallel programming using the traditional low-level thread-and-lock programming model remains a difficult task for most of the programmers since it is time consuming, error prone and requires deep knowledge and skills. Consequently, mainstream programmers are facing a complex productivity-performance trade-off where they should extract enough parallelism to justify the use of a dedicated parallel programming library or system. Moreover, parallel hardware is becoming increasingly heterogeneous: a modern workstation may include two or more multicore processors with several manycore

GPUs. In order to target such architectures, a programmer must have a deep understanding of the target hardware and should often use several disparate programming models making parallel programming even harder and resulting in poor productivity. Exploiting software parallelism on these emerging multicore architectures has become a great design challenge which outlines the need for new technologies to make multicore processors more accessible to a larger community of developers [KB09]. This need was particularly observed by the Thales Airborne Systems company which supported this thesis and allowed us to experiment our research results on real industrial applications.

1.2 Structured Parallel Programming

Due to this technological context, two major needs have emerged: on one hand, a high hardware abstraction to hide details of the underlying platform providing portability , scalability and accommodating its heterogeneity. On the other hand, programmability improvement is in high-demand as it increases productivity and minimizes programming complexity. These two requirements should be satisfied without sacrificing performance and forward scalability.

Programmability is achieved by minimizing/reducing parallel development costs in term of time, complexity and required tools in order to remain as close as possible to traditional sequential development. Parallel development overheads are mainly generated by programming paradigm-related routines such as synchronization, communication, shared memory management, and workload scheduling. These routines introduce a significant amount of extra-code related to parallel programming paradigms and not to the user application itself. Additional effects such as hard debugging and difficult performance tuning are also induced.

Skeleton-based programming, often referred to as "structured parallel programming" [Col91] [Col04], is a promising high-level approach which satisfies most of these requirements and attempts to replace the traditional low-level thread lock model with better abstraction and an easier way to express parallelism through a collection of recurrent parallel patterns [MWHL06] [AD07]. It aims mainly to provide a good trade-off between programmability, portability, reusability and performance enhancement in order to improve programmer productivity by letting him focus on algorithms instead of hardware architectures.

In this thesis, we addressed the productivity issue by designing two task-based programming models which aims to exploit the potential of standard C++ language to ease parallelism expression without sacrificing performances. The first programming model is named XPU and is designed to ease explicit parallelism expression while the second one is named FATMA (FAsT Multicore Application) and is designed to provide automatic parallelization of a large sequence of tasks with complex dependencies.

XPU provides a collection of hierarchical parallel constructs to enable the programmers to express several types of parallelism including task parallelism, data parallelism and pipeline parallelism at different levels of granularity inside a single homogeneous and structured programming model. While easing parallelism expression and improving programmer productivity in many application domains, XPU may be less suited to some applications exacerbating complex dependencies between large number of tasks making explicit parallelism expression very hard. For instance, many tiled linear algebra algorithms may generate thousands of tasks which exacerbate extremely complex dependencies. **FATMA** addresses this issue by extending XPU and providing automatic parallelization capabilities and allowing the programmer to:

1. Parallelize transparently a large sequence of tasks by generating the corresponding task dependency graph.

2. Scheduling asynchronously the parallel tasks on the available processors without violating their dependencies.

While many parallel programming models introduce new programming languages or extends existing languages and hence require specialized compilers and tools, both FATMA and XPU exploit exclusively the potential of standard C++ language and its metaprogramming capabilities to provide high programmability and better productivity. They provide an intuitive and light-weight programming interface which promotes the reuse of sequential code almost without any alteration and requires a little amount of extra-code to express parallelism. Consequently, programs using our FATMA and XPU require nothing more than a standard C++ compiler.

C++ language was selected as the most adapted language to implement our programming models not only for performance considerations but also for its powerful metaprogramming features which provide great flexibility and expressiveness and contribute significantly to simplify the exposed programming interface while improving performances through compile-time optimizations. For instance, programming languages offer different programmability-performance trade-off: for example, C language is designed to offer high performances at the cost of poor and verbose programming interface while higher level interpreted programming languages such as Python offer better programmability thanks to their friendly and powerful programming interface at the expense of lower performances. In our case, C++ was the ideal candidate since it offers the best expressiveness-performance tradeoff to implement efficiently our programming models and enabled us to provide both good programmability and performance to XPU and FATMA users.

1.3 Research Questions

XPU and FATMA treated different research questions which are axed around easing the parallelization of sequential applications (parallelism expression and parallelism extraction) :

1. Easing parallelism expression.
2. Flexibility of the parallelization methodolgy.
3. Programmability and productivity-performance tradeoff.
4. Unification of different parallel programming models : use of different types of parallelism at different levels of granularity in a single homogeneous and structured parallel programming model.
5. Automatic parallelism extraction and tasks parallelization at run-time.

1.4 Contributions of this Thesis

The contributions of this thesis can be summarizing as following:

1. Proposing a flexible and progressive parallelization methodology (MHPM) [1] that represents a program as an hierarchical task graph and allows the expression of several types of parallelism (task parallelism, data parallelism and pipeline parallelism) at different levels of granularity. XPU implements the MHPM programming model using the C++ programming language.
2. XPU exploits a traditional and popular programming language (standard C++) to provide high parallelism expressiveness without introducing any language extension or specialized compilers or tools.
3. XPU exploits C++ metaprogramming techniques to promote the reuse of legacy functions, object methods and lambda expressions as tasks without any alteration.
4. XPU uses standard C++ metaprogramming techniques to extract transparently tasks data dependencies and consumer-producer relationship between tasks.
5. XPU uses the tasks data dependencies information to automate several parallel programming routines such as shared data detection and protection against conflictual concurrent accesses (*race condition*).
6. XPU exploits the information on tasks data dependencies to perform spatial and temporal cache-aware task scheduling.

[1] **Publication** : Nader Khammassi, Jean-Christophe Le Lann, Jean-Philippe Diguet and Alexandre Skrzyniarz, "***MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology***", Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication, HPCC'12, Liverpool, UK

7. Design and implementation of a Cache-Hierarchy-Aware Task Scheduling (CHATS) algorithm [2] for parallel Loops on modern multicore architectures.

8. FATMA exploits the available information on tasks data dependencies to build automatically the task dependency graph (DAG) which specifies the consumer-producer relationships between tasks.

9. FATMA uses a DAG-driven scheduler which perform asynchronous task scheduling in a super-scalar fashion.

1.5 Outline of the Dissertation

The dissertation is organized in as follows:

Chapter 2 introduces the historical background of structured parallel programming with deterministic patterns, presents most recurrent parallel and sequential patterns, gives and overview of several skeleton frameworks and programming models which are relevant and discusses recent researches.

The **Part 2** of this thesis is dedicated to explicit parallelism expression using XPU.

Chapter 3 gives an overview of the XPU parallelization methodology, present an abstract intermediate representation of parallel programs and gives an overview of the XPU framework architecture.

Chapter 4 defines the task object which is the fundamental component of our programming model and explains how the C++ language and its metaprogramming techniques are exploited to allow easy and direct reuse of legacy sequential code as task. The internal design of the task object is described and the transparent task-data dependency feature is discussed in this chapter.

Chapter 5, **6** and **7** discuss explicit parallelism expression using XPU and present a set of hierarchical parallel execution patterns that allow specification of task, data and pipeline parallelism. The XPU programming interface is described and the internal design and implementation of each of the different execution patterns are detailed. Finally, in each of these chapters, practical applications are presented and used to evaluate our approach.

Part 3 The third part of the thesis is dedicated to automatic parallelization using the FATMA framework.

[2] **Publication** : Nader Khammassi and Jean-Christophe Le Lann, "***Design and Implementation Of A Cache Hierarchy-Aware Task Scheduling For Parallel Loops On Multicore Architectures***", Third International Conference on Parallel, Distributed Computing Technologies and Applications, PDCTA 2014, Sydney, Australia

Chapter 8 introduces the FATMA programming model and its automatic parallelization capability. This chapter explains how the transparent data dependency extraction feature of the task object can be used to build automatically and dynamically the task dependency graph. Finally, we show how we can use this task graph to drive the asynchronous execution of the parallel tasks in a superscalar fashion.

Part 4 The last part of the thesis is dedicated to practical industrial applications.

Chapter 9 and **10** presents two high performance real-time signal processing applications. These applications are two industrial case studies where XPU has been used to parallelize the sequential code. The progressive parallelization process is described and the achieved performance are discussed.

Chapter 11 concludes this thesis by summarizing the contributions of this thesis and their limitations and providing future research perspectives.

1.6 Terminology

In this section we define the some ambiguous terms which are used in the literature to express different meanings depending on the context:

1. "*Hybrid*": In the parallel programming context, this term is used in the literature to describe parallel programing models for several different meanings depending on the context: for example this term can be used to indicate the ability of a programming model to express parallelism at both the distributed memory and shared memory levels (use of OpenMP and MPI for example) while the same term is used to describe a programming model which is able to exploit heterogeneous Multicore architectures such as Multicore CPU and GPU. In our case this term is used to indicate the ability of our programming model to express sequential execution and several type of parallelism in a single homogeneous and structured/hierarchical programming model.

2. "*Skeleton*","*Parallel Construct*", "*Parallel Pattern*": we use interchangeably "*parallel pattern*", "*parallel construct*" or "*algorithmic skeleton*" to indicate a structure storing a set of tasks and specifying their execution configuration.

1.7 Publications

1. Nader Khammassi, Jean-Christophe Le Lann, Jean-Philippe Diguet and Alexandre Skrzyniarz, "***MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology***", Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication, HPCC'12, Liverpool, UK

2. Nader Khammassi and Jean-Christophe Le Lann, "***A High-Level Programming Model to Ease Pipeline Parallelism Expression On Shared Memory Multicore Architectures***", 22nd High Performance Computing Symposium, ACM HPC 2014, Tampa, FL, USA

3. Nader Khammassi and Jean-Christophe Le Lann, "***Tackling Real-Time Signal Processing Applications on Shared Memory Multicore Architectures Using XPU***", Embedded Real Time Software and Systems ERTS 2014, Toulouse, France

4. Nader Khammassi and Jean-Christophe Le Lann, "***Design and Implementation Of A Cache Hierarchy-Aware Task Scheduling For Parallel Loops On Multicore Architectures***", Third International Conference on Parallel, Distributed Computing Technologies and Applications, PDCTA 2014, Sydney, Australia

5. Book Chapter: Nader Khammassi and Jean-Christophe Le Lann, "XPU: A C++ Metaprogramming Approach To Ease Parallelism Expression: Parallelization Methodology, Internal Design and Practical Application", In "***Parallel Programming: Practical Aspects, Models and Current Limitations***", Mikhail S. Tarkov (ed.), 2014, Nova Science Publishers, Inc., Hauppauge, NY, USA. (Being Printed)

2
State of The Art

For decades, parallel hardware was synonym of distributed multiprocessing systems that were affordable only to large laboratories and corporations and thus only expert parallel programmers were able to use such platforms using a limited set of parallel software and tools. Today, multicore systems are accepted in all industry segments. The quick spreading of multicore and manycore architectures has placed programmers under great pressure to parallelize their software to be able to exploit modern parallel hardware efficiently and consequently outlined the need for new parallel software technologies. Hence, translating traditional sequential code to parallel code has become a great design challenge for software designers who attempted to ideally automate the whole parallelization process.

2.1 Compiler-Based Parallelization

2.1.1 Automatic Parallelization Compilers and Tools

Despite decades of researches and development of parallelizing compilers, automatic parallelization of sequential code using a compiler is standing as the holy grail of parallel computing and has had a limited success [SL02]. Great advances has been made in automatic parallelism extraction at the instruction level, however, in order to exploit efficiently modern multicore platforms, compilers need to capture parallelism also at thread level which is a very challenging task. There are a number of Automatic Parallelization tools :

Par4All : Par4All [ACE+12] [VSG+12] is an automatic parallelizing source-to-source compiler for C and Fortran sequential codes. It generates parallel source code based on the input sequential code. It targets various parallel hardware such as multicore and manycore systems (CPUs and GPUs [VCJC+13]) and HPC systems.

Cetus : Cetus [DBM+09] is another source-to-source compiler developed at the Purdue University. The compiler infrastructure is written in Java and can be used to design automatic parallelization compilers and tools. Currently, Cetus exploits several basic parallelization techniques such as reduction variables recognition, privatization and induction variable substitution. A new graphic interface has

been introduced recently to display some information such as graph representation and speedup calculations. A Cetus client-server system has been developed to allow users to parallelize C code through the server. Since May 2013, users can run Cetus through a web browser.

Parallware : Parallware [App20] is a source-to-source compiler developped by Appentra. Parallware analyzes the input sequential code to capture parallelism and generates a parallel code which use compiler directives to express parallelism. Parallware targets multicore and manycore HPC systems.

PLUTO : PLUTO [BRS07b] [BDF+06] is a parallelization tool. It is based on the polyhedral model which is an intermediate program representation used to help performing high-level transformations such as loop parallelization and optimizations [BRS07a] [BBK+08]. PLUTO can transform C source code to capture coarse-grain parallelism and data locality. Affine transformations are captured by the core transformation framework which use them to perform efficient tiling and fusion. Finally, PLUTO is able to generate automatically parallel code that use OpenMP directives from sequential portions of C code. More recently, GPU support has been made available through CUDA code generation.

Polaris Compiler: Polaris [BEF+95] is a Fortran source-to-source compiler that transforms sequential Fortran77 programs into parallel programs. The output source code can use different parallel FORTRAN dialects. Polaris parallelize the input source code through several compilation passes using various parallelization techniques such as data dependency testing, array privatization, induction variable recognition, inter-procedural analysis and symbolic program analysis.

Intel C++ Compiler : The Intel C++ Compiler provides an automatic parallelization feature [Inta] which allow thread-level parallelization of sequential portion of the input code. Targeted sections include loops that exhibit good work sharing. In order to make the threaded parallel program semantically equivalent to the sequential input program, the Intel Compiler performs a data-flow analysis to ensure correct parallel execution and data partition as needed with OpenMP directive. The resulting program is intended to provide performance gains on shared memory multicore systems.

iPat/OMP : iPat/OMP [IHS06] is a tools that aims to assist user in OpenMP parallelization of serial programs. This tool consist in a set functions ion the Emacs editor. User can select a target portion of his program and invoke the assistance command of the tool using Emacs command to get assistance information on parallelization of the selected portion of the code.

Vienna Fortran compiler(VFC) : VFC [Ben99] is a new source-to-source compiler based on HPF+ (optimized version of HPF parallelization system). It addresses the special requirements of irregular applications.

2.1.2 Limitations and Workaround

Automatic parallelization using compilers or tools suffers from inherent difficulties [BE92] [DRVV00] such as:

- Complexity of dependence analysis especially when the code uses indirect addressing through pointers, recursion or indirect function calls.
- Difficulty of global resources access management which requires coordination of concurrent accesses to resources (memory allocation, In/Out, shared data...)
- Unknown iterations count in loops.
- Irregular algorithms using input-dependent control can make compile-time analysis and optimization very difficult.

The inherent complexity in the automatic parallelization process using compilers outlined the need for higher level programming models which allow programmers to "help" compilers by introducing "hints" in their program in order to assist and guide the compilation process. These hints can be more-or-less explicit and can take several different forms:

- Compiler Directives such as those specified by OpenMP [DM98] or OpenHMPP [Hir12]
- Programming Language Extensions such as Cilk Plus [Rob13]
- New Parallel Programming Languages such as Chapel [CCZ07a] or X10 [CGS+05]
- Libraries Based On Traditional Programming Languages such as Intel Threading Building Blocks [Phe08]

2.2 Language-Based Parallelization

2.2.1 Low-Level Parallel Programming Models Limitations

Parallel programming using low-level thread/lock programming models such as the POSIX Threads is a hard task for most of programmers since it is time-consuming, error-prone and requires deep knowledge and skills. Parallel programming is challenging at many levels:

- **Design of the parallel algorithm:** In addition to the challenges exposed by traditional serial algorithm design, parallelization may require deep restructuring of the sequential algorithm which may involve computation decomposition, complex dependencies analysis and data partitioning and distribution...

- **Implementation of the parallel algorithm:** Implementation of the designed parallel algorithm may introduces additional burdens to the programmer such as shared memory management, communication and synchronization of concurrent computations. Programmer should manage shared memory to avoid conflictual concurrent accesses to shared memories often referred as "race condition". While using carefully mutual exclusion mechanisms to prevent "race condition", programmer should avoid deadlock which may be caused by these tools.

- **Testing and Debugging:** Due to the non-deterministic execution of parallel computation, debugging and testing of the parallel programs can be a very hard task which exacerbates the difficulty of localizing potential bugs within large number of parallel computations and the difficulty of reproducing concurrent execution behaviors.

- **Performance tuning:** Performance and execution efficiency is the primary design goal of parallel algorithms. When an application is parallelized successfully, programmer tunes the performance of his program through efficient computation scheduling, good load balancing and effective cache use while avoiding potential memory contention and bottlenecks... This process is often challenging since it requires not only good programming knowledge and skills but also deep understanding of the underlying architecture. The fact that parallel hardware is becoming increasing heterogeneous is making this task even harder.

- **Portability and forward scalability:** A parallel application should exhibit enough hardware abstraction to hide low-level hardware architecture details, accommodate hardware heterogeneity and provide good software portability across different parallel hardware. While displaying good portability, a parallel application should also be able to adapt to new underlying architectures to maintain execution efficiency and provide forward scalability: migration of parallel programs to new parallel hardware requires often substantial rewriting of significant parts of the application which implies the reiteration of previously described process of design, implementation, testing and performance tuning.

As we can see in these different stages of the parallelization process, most of the parallel programming overhead is generated by the parallel-paradigm related routines such as shared memory management, synchronization, communication and workload scheduling. These routines introduce a significant amount of extra-code related the parallel programming paradigms and not the programmer application itself. Thus, a significant part of the parallel programming complexity came from the merging between the application computation which constitutes the payload of the application and the coordination, interaction and communication between these computations.

This complexity outlines the need for high-level programming models which abstract the low-level parallel paradigms-related details. High level programming approaches decouples the programmer algorithm from both the parallel programming-related routines

and the underlying hardware. Structured parallel programming with deterministic patterns often called algorithmic skeletons is a promising high level approach which aims to offer this abstraction and reduce programmer burdens.

2.2.2 Structured Parallel Programming with Patterns

Skeleton-based programming, often referred as structured parallel programming [Col04] [Col91] offers a clear separation between these two complementary entities: **computation**, which performs the calculations in a procedural fashion, and **coordination**, which abstracts the communication and interactions between these computations.[GVL10]

Structured parallel programming is a promising high-level approach which attempts to replace the traditional low-level thread lock model with better abstraction and an easier way to express parallelism through a collection of recurrent parallel patterns [McC10] [AD07]. By abstracting commonly used patterns of both parallel computation, interaction, and communication, algorithmic skeletons allow programmers to code algorithms by composing and parametrizing these skeletons. This allows the programmer to provide an abstract structured description of his program without specifying any platform-dependent primitives.

By decoupling the structure of a parallel program from its behaviour, a structured parallel program benefits from any improvements in the underlying parallel system infrastructure. Therefore, thanks to its high abstraction, the structured parallel programming model appear to be a viable and promising approach which offer not only programmability and reusability but also portability and forward scalability across a wide variety of hardware. Therefore, this approach can improve significantly the programmer productivity by letting him focus on algorithms instead of hardware architectures.

In the following section we give an overview of the background of skeleton-based programming then we expose a set of common sequential and parallel patterns. The two last sections describe a series of representative algorithmic skeletons framework and discusses a set of related approaches.

2.2.2.1 Background

The concept of "**algorithmic skeleton**" was introduced by Cole [Col91] [Col04] and elaborated by Skillicorn [ST98] [McC10]. While specific to computation configuration, this concept is similar to the concept of design pattern [MSM04]. Therefore, we use the term "skeleton" and "parallel pattern" interchangeably to indicate a recurrent execution configuration which specifies a specific pattern of computation, coordination and data access.

A set of characteristics workloads was identified in the View from Berkeley [ABD+09] as "motifs" or "dwarves". These "motifs" consist of a set of different types of execution patterns [McC10] which exposed a set of common sequential and parallel patterns. While the different types of patterns were presented separately, in most applications, a variety of patterns are composed linearly or hierarchically to describe a structured algorithm [BMA+02] [TSS+03] [SG02].

In the early 1970s,it was noted that composing control flow of serial program using a small set of control flow patterns (sequence, selection, iteration and recursion) could make them easier to understand. This structured programming approach resulted into the elimination of "goto" from most programs despite a lot of controversy [Dij79]. In the last decades, structured control flow has been so widely accepted that "goto" is either deprecated or disappeared from most modern languages [McC10] [MRR12].

Analogously, structured parallel programming with parallel pattern is a promising approach which can hide the low level threading and synchronization details while displaying an understandable structured representation of parallel programs. Threads are similar to "goto" in their lack of flexibility and structure [Lee06]. Similarly to "goto", use of concurrent threads with random access to data make program difficult to decompose into different parts which form a structured program where different parts are clearly isolated and do not interfere with each others.

Functional programming can be seen as an alternative. However most modern mainstream programming languages are not functional. Moreover, many algorithms such as graph problems can be difficult to express using pure functional languages. However, functional programming can be considered as a specific case of structured parallel programming where only a subset of the parallel patterns is used.

Collective languages is another class of languages which allows explicit expression of parallel operations over large collections of data. Collective languages can be implemented using either functional or imperative languages. Mainstream languages are often used to implement collective operations as libraries which offer a set of built-in operations. For example, NESL [BHC+93] [Ble96] is a collective functional language while FORTRAN 2003 is an imperative language which offer a set of built-in collective operations. Ct and RapidMind [MWHL06] are imperative languages which are based on collective operations.

Structured parallel programming with patterns can be implemented by designing new languages, by extending existing mainstream languages, by designing code generators [Her03] or by using the traditional mainstream languages to provide libraries of parallel patterns.

2.2.2.2 Sequential Patterns

While addressing parallel patterns, it is important to discuss sequential patterns since sequential computations constitute the major part of most programs and also because parallel programs are derived from sequential ones and some parallel patterns are analogously derived from sequential patterns: for example, parallel loops are directly derived from sequential ones. In the following, we describe a set of common sequential patterns which can be seen as the *building block patterns* of serial programming models such as imperative programming and functional programming.

A Sequential Control Flow Patterns

In structured programming approach, the control flow of a program can be specified through a small set of composable and parametrizable control flow patterns. We note that the four following patterns (sequence, selection, iteration and function and/or recursion) are not all needed to specify generic computations. Only a subset of them leads the classes of functional and imperative languages [MRR12].

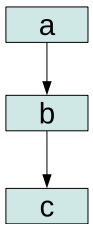


Figure 2.1: Sequence

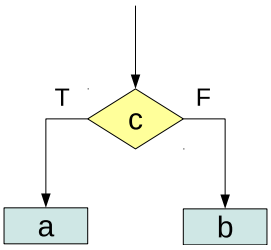


Figure 2.2: Selection

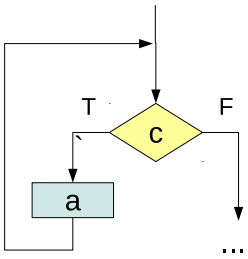


Figure 2.3: Iteration

Sequence A sequence is a list of tasks that are executed in a specific order. The sequence pattern is used to specify dependencies between consecutive tasks so that one task cannot start before another is finished (serial execution).

Selection In the selection pattern, a boolean condition is evaluated to select which task from two tasks should be executed. The evaluation of the later condition constitutes a third task which is executed before the selection.

Iteration In the iteration pattern a task, which constitutes a loop body, is executed repeatably until a variable condition become false. The different iteration of the task are executed in sequence. The condition is evaluated at each iteration before task execution.

Function and Recursion Recursion is a form of dynamic nesting which a allows functions to call themselves recursively. Recursion is a common pattern which is used in many popular algorithms such as *Fibonacci* or *Quick Sort*.

B Sequential Data Management Patterns

Data management patterns abstract data accesses at different level: the random read and write accesses are the lowest level abstraction which maps directly to the low-level hardware mechanisms. Heap and Stack allocation are other patterns which offer a higher abstraction.

Random Read and Write *Random Read or Write* consists in accessing the memory at a given index to respectively retrieve or modify the associated value to that index. Arrays and vectors are some of the simplest abstraction of randomly accessible memory. We note that more sophisticated data structures can be constructed on top of them.

Stack Allocation *Stack Allocation* is last-in-first-out (LIFO) allocation model which is often used to handle nested function calls and local states storage. *Stack Allocation* displays an excellent temporal and spatial data locality properties. This pattern can be parallelized by associating a stack to each thread of control to preserve locality. For instance, the Cilk Plus [Rob13] is using a calling convention which generalize stack allocation in the context of function calls in order to preserve the locality properties of stack allocation.

Heap Allocation When *Stack Allocation* model is not suitable, data can be dynamically allocated from memory pool commonly called the heap. *Heap or Dynamic Memory Allocation* is a more general memory allocation model that consist in allocating fresh memory dynamically when requested. Heap allocation is significantly slower and more complex than the stack allocation. It can result in fragmented allocation over the memory. These scattered allocations can result into poor data locality and reduce considerably memory access efficiency [WJNB95].

Parallel programs using heap allocation can suffer from scalability problems since data structure used to manage the memory is shared among all threads. Often, parallelized heap allocators are used to reduce the concurrent memory allocation overhead. These custom allocators use often a pre-allocated memory pool to avoid calling constantly the native memory allocator which use a global lock to preserve memory coherency. The jemalloc [Eva], tcmalloc [GM], ptmalloc, concur, hoard [BMBW00] and the TBB allocator [Phe08] are the most popular parallel heap allocators.

Data Abstraction and Collection Data management may require more sophisticated data structure than a simple one dimension array. Example of such data abstraction often referred as *Collections* are nested arrays, graphs and tree. *Collections*

can use auxiliary data structures to operate more efficiently on data. *Partitioning*, *searching*, *ordering*... are examples of such operations.

For example, HashMap is and abstract collection which may use auxiliary data structures to store *hashes* and search, and retrieve data efficiently. List is another type of dynamic collection which may be implemented using an auxiliary "linked nodes" structure to store data and manipulate it...

We note that Collections can be nested inside each others to compose multi-dimensional data. Many high-level abstract collections are recurrent patterns and are implemented as library in mainstream languages. The C++ Standard Template Library (STL) [SL94] offer a set of abstract Collections or "Containers" which aims to provide a tool for efficiently storing and operating on data. Vector, Set, List, Map ... are example of the implemented Collections in the STL library.

C Sequential Programming Model

The previously enumerated patterns can be used to describe the two most common sequential programming models which are imperative and functional programming.

Functional Programming The functional programming model is based on hierarchical nesting of functions which produce hierarchical graphs of function calls. Pure functional languages requires only the selection and recursion patterns to specify the control flow of a program. In term of data management, the random read pattern is used to describe data accesses. Pure functional language are universal despite their simplicity. However, algorithm depending on incremental in-place modification of data exacerbate the difficulty of implementation in pure functional languages. One of the major advantage of functional languages is the inherent parallelism which is "implicitly" specified by the program structure since the data dependencies which constrain task ordering is naturally expressed. This make programs written in pure functional languages good candidates for parallelization.

Imperative Programming In the imperative programming model, computations ordering is explicitly specified by the programmer. In order to be universal, at least the sequence, selection and the iteration control-flow patterns need to be supported by the sequential imperative programming model. In term of data accesses patterns, the random-write and random-read need to be implemented. Imperative programming is the most prevalent sequential programming model today. From a parallelization point of view, the over-specification of computation ordering is one of the major inconvenient of the imperative programming model: all dependent or non-dependent computations are explicitly ordered making difficult to determine which ordering specification are essential for the correct execution of the program and which specification ordering are not.

2.2.2.3 Parallel Patterns

Parallel patterns can be classified as either *Computational Patterns* or *Data Management Patterns*. Computational patterns operate on data values while data access patterns organize access to the data without operating on its values. *Computational Patterns* can in turn be classified into *Data Parallel Patterns* and *Task Parallel Patterns*. The first class, e.g. data parallel pattern, abstracts collection-oriented parallel computations. In data parallel patterns, the same task is usually "replicated" to operate concurrently on the elements of a given collection. Task parallel patterns do not operate necessary on data collections. Task patterns specify the execution configuration of a set of tasks which may operate on collections, stream of data or other data structures.

A Data Parallel Patterns

Map The *Map* pattern performs the same computation on all the elements of a collection. This can be done by replicating a task over all element within a partition of the target collection. Typically, *Map* pattern can replace loops in serial programs when loop iterations are independent and their number known. In this case, each iteration can be used as the index of element being processed. The *elemental function* must execute without any side-effect, i.e. should not modify the data on which other iterations are dependent to guarantee the determinism of this pattern. Since elements of the target collection are processed independently, the *Map* pattern can specify a large amount of parallelism while achieving deterministic results.

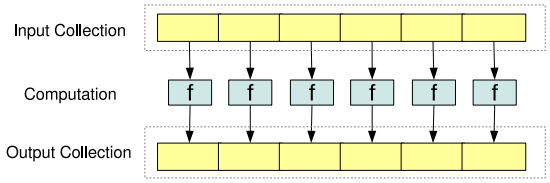


Figure 2.4: The Map Pattern

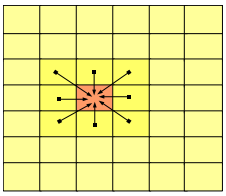


Figure 2.5: The 2D Stensil Pattern

The *Map* pattern can be used in many applications such as in image processing for example : thresholding, color space conversion, gamma correction... In these algorithms, each pixel is processed without interfering with the other pixels so the image can be seen as "collection" or array of pixels which can be processed concurrently using the map pattern.

Stencil The Stencil pattern can be seen as a generalization of the *Map* pattern: the elemental function in the Map pattern consumes one elements to produce another, while in the Stencil pattern, it can consume several elements to produce one element. Figure 2.5 shows an example of a Stencil pattern producing one element using an input elements and its neighboring elements.

Stencil pattern is used in many applications such as convolution in image filtering, motion estimation in video encoding. Additionnally, Stencil is a recurrent pattern in partial differential equation (PDE) solvers such as in fluid flow solvers.

Zip The *Zip* pattern is a generalization of the Map pattern. Instead of using a single collection as data input like in the Map pattern, the *Zip* pattern use two collections. These two input collection are used to produce one single output collection.

Reduction The *Reduction* pattern reduces the elements of a collection to a single element by combining the element of the collection using a pairwise associative operation. Typical operations are *Minimum*, *Maximum*, *Sum*... etc. These operations are often called the combiner functions. The *Reduction* pattern can be parallelized using a tree structure which applies the combiner function to the different partitions of the input collection to produce an intermediate smaller collection, then reiterates this operation until reducing the last and smallest intermediate collection to a single element.

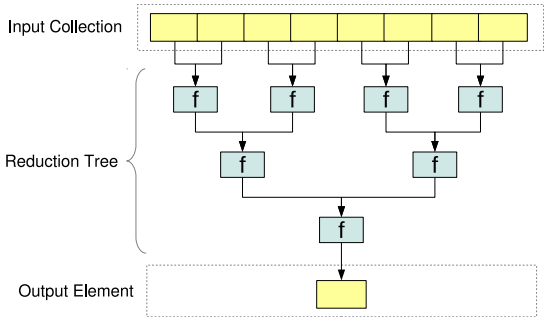


Figure 2.6: Reduction Pattern

The computation of the sum of array elements is an example of cases where reduction can be used. The array can be split into a set of "*sub-array*" or array partition. The sum of elements of these sub-arrays is computed to form new and smaller arrays. The same computation is reiterated to finally sum the elements of the smaller sub-array.

Permute The *Permute* pattern is used to permute elements of a collection. Given the index of the input element, a function produce the destination index of that element. A parallel implementation of the *Permute* pattern may involve communication since operation on each element is likely to interfere with the other element. Consequently communication may be needed to ensure ordering of operation and determinism of the results.

Shift The *Shift* pattern is a specialization of the *Permute* pattern which limits data displacement to left or right. In this case, only the boundaries elements, e.g.

at the far right and left of the collection, requires communication between concurrent permutation functions.

Scan The *Scan* pattern performs partial reduction depending of the output element index or the position in the output collection. For every output index, a partial reduction is performed on input starting from the first element up to that output index. The *Scan* pattern can be considered as a specialization of a serial pattern called fold. In the fold pattern, a function called the successor function is used to generate the current state based on the previous state plus an additional input. When the successor function is not associative, it is generally not possible to parallelize the fold due to direct dependencies between successive computation steps. however, when the successor function is associative, operations can be reordered to allow for parallelization. The associativity of the successor function is the particularity which make the distinction between the scan as a particular case from the fold as the general one.

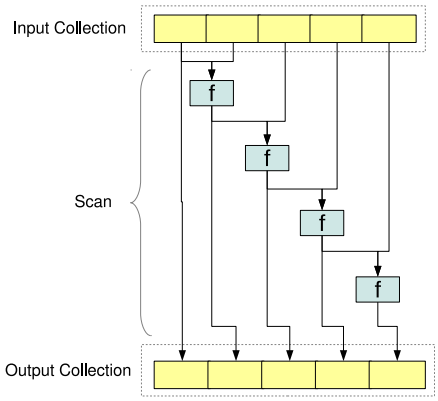


Figure 2.7: Serial Scan Pattern

While it can be possible to parallelize a scan at the cost of more work, it is not possible to achieve a linear speedup. This may lead to poor scalability. Hence alternative algorithm implementation should be considered. Integration, sequential decision simulation and random number generation are popular examples of algorithms which can use the scan pattern [MRR12].

Recurrence The Map pattern can be seen as a parallel specialization of the Iteration pattern since it can be seen as the result of parallelization of a loop where all iterations are independent. Analogously, the *Reccurence* pattern can be seen as a specialization of the Iteration pattern, however, contrary to the Map pattern, the *Reccurence* pattern can be seen as the result of a loop parallelization where loop iterations can be dependent on each others. In other words, in the Recurrence pattern, one output is the result of a function in terms of prior outputs. In a serial code, Recurrences occur

due to loop-carried dependencies, however, in many cases these dependencies can be accommodated and recurrences can be parallelized. In the case of one dimension configuration, when the dependency between iterations is associative, recurrences can be parallelized into logarithmic time. This implementation is often called a scan [Ble90]. In multi-dimensional configuration with a nesting depth of N, recurrences can be parallelized over N-1 dimensions, even if the operator is not associative, using the Lamport's hyperplane theorem [Lam74].

B Parallel Data Management Patterns

Parallel Data Management Patterns (PDMP) are used to organize access to data while preserving coherency, e.g. avoiding conflictual concurrent access such as race conditions. Managing concurrent access to shared data is particularly critical in the case of some access patterns such as the Scatter pattern. In many other patterns, coherency can be preserved by structuring access to data. PDMP are also responsible of improving data access efficiency through improving spatial and temporal data locality. In the following lines, we give an overview of most used PDMP.

Pack Given a data Collection, the Pack pattern can be used to reduce its size by eliminating unused space. This can be done by associating a Boolean marker to each element of the collection. The Pack pattern is then responsible for discarding elements which are marked "False". The other elements, e.g. marked "True", are placed in the output data Collection as a contiguous sequence. Figure 2.8 gives an overview of the pack pattern.

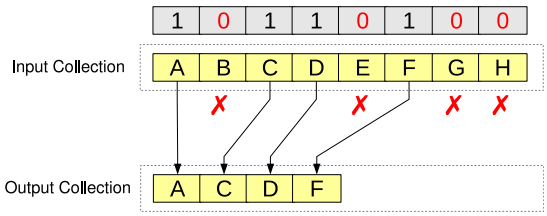


Figure 2.8: Pack Pattern

The Pack pattern is particularly useful when coupled with other computing patterns such as the Map pattern to reduce the size of its output by eliminating unnecessary elements from the computation results. The Pack pattern can be used to emulate SIMD machines with good performances [MRR12] [LLM08] [HLJH09]. Basic data compression algorithms and collision detection are examples of application of the Pack pattern.

Gather The Gather pattern generates an output data Collection using an indexable data Collection and a Collection of indices as inputs: a Gather reads the elements from the input data Collection at the positions specified by the Collection of indices

then write these elements in output Collection. Since it does not display dependencies between its elementary operations, the Gather pattern can be easily parallelized.

Scatter Contrary to Gather, the Scatter pattern use the Collection of indices to write the elements of the indexable collection at the positions specified by these indices. The Scatter pattern may involve multiple concurrent writing at the same position usually called Collisions. This make the parallel implementation of Scatter more complex. Several specializations of this pattern aims to resolve this problem using different policies: The Priority Scatter is an implementation of the Scatter pattern that resolve the multiple concurrent write at the same position using deterministic write priority rules. The Atomic Scatter is another implementation which is however non-deterministic: it does not guarantee operation ordering, however, it ensures that correct results are safely written to a given position when concurrent write happen. There is other types of the Scatter pattern sucha as the Permutation Scatter or the Merge Scatter which are not discussed here and which are detailed in [MRR12].

C Task Parallel Patterns

Fork-Join Pattern The Fork-Join pattern enable the control flow to fork into several parallel execution flows that can rejoin on a common synchronization point Figure 2.9. The parallel flows can result from the replication of a same flow or from a set of different flows.

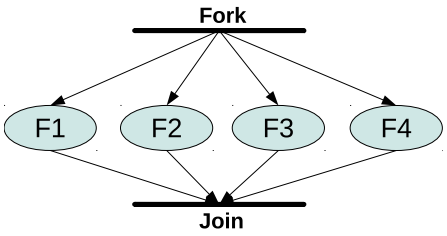


Figure 2.9: Fork-Join Pattern

For example, the OpenMP programming model [DM98] supports both of these execution modes: the "*parallel*" directive can execute the same statement in different threads or execute several different statements (specified as "*sections*") concurrently. The Cilk Plus programming model [Rob13] implements this pattern using the two keywords "*spawn*" and "*sync*". The "*spawn*" keyword is used to execute a function in a concurrent thread instead of calling it int the current one. Several functions can be spawned the same way. The "*sync*" keyword allows the synchronization of the spawned functions at a given point. Nesting multiple fork-join patterns in a structured fashion generates an hierarchical task graph. Task parallelism in a program can be specified at all levels of granularity through using such structured task graph.

Pipeline Pattern Similarly to a production assembly line, the *Pipeline* pattern specifies a consumer-producer relationship between multiple tasks. The *Pipeline* consists in a set of simultaneously active tasks called "*stages*" that communicate following a producer-consumer relationship: each stage is responsible of both consuming and producing item of data. Therefore, each pair of adjacent stages forms a producer-consumer pair. At the opposite of the sequence pattern where completely dependent tasks are executed serially, pipeline stages are activated at the same time. However, in order to recover data coherency, sequentially dependent activities or "*folds*" are serialized, parallelism is exploited only on independent activities.

Super Scalar Task Graph Pattern The *Super Scalar Task Graph (SSTG)* can be seen as a generalization of the Pipeline pattern and also as a composition of both the pipeline pattern and the fork-join pattern. While the pipeline pattern specifies a linear consumer-producer relationship between each pair of stages where each single consumer depends on a single producer, the STTG can specify more generalized consumer-producer relationship where multiple consumers can depend on a single producer or the opposite configuration where a single consumer can depend on multiple producers.

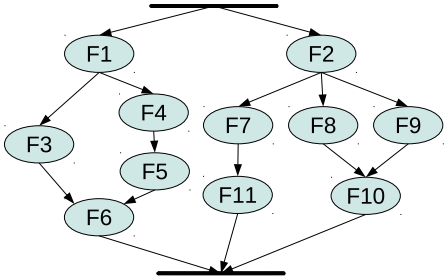


Figure 2.10: Super Scalar Task Graph Pattern

As depicted in Figure 2.10, the task dependency specification in the SSTG pattern can be represented through a directed acyclic graph (DAG). The DAG is composed of nodes and directed edges. The nodes of the graph represent the tasks while the directed edges specify the dependency between the consumers and the producers. The edge is directed from the producer to the consumer.

2.2.2.4 Skeleton Libraries and Languages

Algorithmic skeleton concept are widely accepted as an efficient mean for describing the common control structures of parallel programs in computational science. The skeletal approach has guided the design and development of many languages, frameworks and tools. In the following paragraphs we present a set of the most prominent skeleton frameworks. Despite being structured parallel programming models that are based on deterministic patterns or parallel constructs corresponding to common skeletons, many

popular libraries such as Threading Building Blocks (TBB) from Intel [Phe08], Task Parallel Library (TPL) from Microsoft [Micb] or MapReduce from Google are often not presented in the literature as pure skeleton frameworks but as "related approaches" such as in [GVL10]. Yet, being widely used, these frameworks have bring skeletal programming to mainstream practice and applications. Since we consider the line separating parallel skeletons and these parallel patterns too blur to make clear distinction between them, these libraries will be presented as skeleton or pattern libraries along with the others.

A Skeleton Frameworks Classification

Before presenting the skeleton frameworks, a functional classification of them can be useful to distinguish the common approaches used for implementing algorithmic skeletons. Skeleton frameworks can be classified according to the used programming paradigms such as in [GVL10] where four approaches are clearly identified: Coordination, Functional, Object-Oriented, Imperative. The later classification is mainly based on the language of implementation of skeleton frameworks which is a judicious criterion of classification. In order to make this language aspect of the classification even clearer, we propose the following classifications:

- **New Languages** : This class of frameworks advocates the introduction of new high-level coordination languages that allows programmer to describe the program structure through algorithmic skeletons. The coordination language is then translated into a host language or the execution language responsible of interacting with execution infrastructure. The Structured Coordination Language (SCL) [DT95], the Pisa Parallel Programming Language (P3L) [BK96], the llc language [DGRDS03] and Single Assignment C language (SCL) [Gre05] are examples of programming models that introduces new coordination languages to describe algorithmic skeletons at high level. Source-to-source compilers are often used to translate the skeletal description into host language that can be compiled into executable code using traditional compilers. This approach suffers from a major disadvantage which is the need to learn new language and the use of dedicated compiler and tools that may requires long development time to reach the maturity and performance of traditional compilers.

- **Language Extension** : Instead of introducing new skeletal languages, this approach relies on syntax extension of existing languages to widen their capabilities with a parallel programming extension. Eden [LOmPnm05] and the Higher-Order Divide-and-Conquer language (HDC) [HL00] follow this approach by extending the Haskell language with parallel extension to allow skeletal programming. HDC translates the target program into C program with the MPI environment while Eden uses the Glasgow Haskell Compiler. Similarly to the previous approach, language extension-based approach exposes the need to learn the syntax extensions and relies on intermediate compilers or translators to generate the host language code that can be compiled into executable machine code.

- **Traditional Language** : In order to avoid introducing new languages or extending existing one, this approach tries to exploit traditional language to expose a friendly programming interface that aims to easing skeleton description and composition. Such skeleton framework are often designed as libraries for popular language. Depending on the target language, the programming interface can be designed using object-oriented paradigms or imperative paradigms.

  - ***Object-Oriented*** : In Object-oriented languages, skeletons are often encapsulated into classes and exploit the abstraction of the object paradigms. C++ and Java are the most used host language since they are very popular. We note that auxiliary libraries and environment such as POSIX Threads and MPI may be used to support the execution infrastructure. The Munster Skeleton library (Museli) [CPK09], the Skeleton in Tokyo SkeTO [MIEH06] and the Malaga-La Lagune-Barcelona (Mallba) library [AAB+02] are examples of C++/MPI skeletons that implement respectively task-parallel, data-parallel and resolution skeletons. The Java Skeleton (JaSkel) [FSP06], Calcium [CL07], muskel [ADD07], Lithium [ADT03] and Skandium [LP10] provide a set of skeletons as Java classes.
  - ***Imperative*** : Procedural programming language can also be used to design skeletal programming interfaces. The Skeleton-based Integrated Environment SKIE [BDPV99], ASSIST [Van02], the Pisa Skeleton Library (SKELib) [DS00] and the Edinburgh Skeleton library (eSkel) [BCGH05] are examples of skeletal libraries that exploit traditional procedural languages to describe skeleton structures. While implemented on top of C++, ASSIST and SkIE are classified as imperative because they exploit mainly the C capabilities of C++ language. While delivering higher performances than the other higher level approaches, Skeleton frameworks that are based on low-level procedural languages such as C suffer from weak programmability due to their verbose programming interface. content...

B Skeleton Frameworks Description

In this paragraph, skeleton frameworks are briefly presented then their individual features are described:

ASSIST: ASSIST is a programming environment for parallel and distributed programs. It is based on a structured coordination language that use the parallel skeleton model. ASSIST represents parallel programs as a graph of software modules [Van02]. The graph itself is a software module that specifies the interactions between a set of modules composing the application. The interacting modules can be parallel or sequential. Sequential modules can be written in C, C++ or Fortran while parallel modules are programmed using a dedicated ASSIST parallel module named "" [GVL10]. In order to decouple the compiler and run time layer from the actual grid infrastructure, ASSIST exposes a Grid Abstract Machine (GAM) layer. Programming in ASSIST can

be peculiar since skeletons are not pre-defined, and the programmer is responsible of specifying them by specializing the generic parmod construct.

Calcium and Skandium : Calcium [CL07] provides a set of task and data parallel skeletons in a Java library. Skeletons are nestable and are instantiated using parametric objects. Calcium supports skeleton execution on different shared and distributed infrastructures such as symmetric multiprocessing platforms and cluster platforms. Calcium provides a performance tuning model which helps programmer to locate bugs and performance bottlenecks in his code. Additionally, Calcium provides a transparent algorithmic skeleton file access model for data intensive applications [CL08]. More recent researches of this group has centered on a complete re-implementation of Calcium named Skandium. The new implementation is focused on easing parallel programming on multicore architectures [LP10] [GVL10].

OSL : The Orléans Skeleton Library (OSL) is a C++ skeleton library which is based on the Bulk Synchronous Parallel (BSP) model of parallel computation. OSL provides a collection of data parallel skeletons. OSL is implemented on top of MPI and uses meta-programming techniques to offer a good efficiency. OSL aims to provide an easy-to-use library which allows simple reasoning about parallel performances based on a simple and portable cost model [JL09].

Eden : Eden [LOmPnm05] extends the Haskell functional language by providing support of parallel shared and distributed memory environment. In Eden, parallel programs are organized as set of processes explicitly defined by the programmer. These processes communicates with each other implicitly [BLMP97] through unidirectional single-writer-single-reader channels. The programmer is responsible of defining data dependencies of each process. Eden defines a process model that allows process granularity control, data distribution control and communication topology definition. The concept of the implementation skeleton that describe the parallel implementation of a skeleton independently from the underlying architecture has been introduced by Eden [KLPR01]. Eden supports task and data parallelism through a set of skeletons that are defined on top the process abstraction layer.

eSkel: The Edinburgh Skeleton Library (eSkel) [BCGH05] is deployed in C and uses the MPI environment. Skeletons in eSkel can be used in two distinct modes: nesting and interaction. The first mode can be persistent or transient, persistent nesting specifies that nested skeletons are instantiated once and remain reusable through the application while the transient mode specifies that the skeletons are constantly created and destroyed each time they are invoked in the application. In term of performance prediction, eSkel employed different performance model including empirical methods and process algebra through Amoget [GVL10]. More recent researches from the Edinburgh group have addressed the adaptability in structured parallel programming [GVC07] in general and more particularly the adaptation of the pipe skeleton [GVC08] and the farm skeleton [GV06].

HDC: The Higher-order Divide-and-Conquer language (HDC) [HL00] extends the Haskell functional language and widen its scope with parallel extensions that specify skeletal behavior. HDC presents functional program as a set of polymorphic higher-order functions. These function are compiled into C code with MPI environment then linked to skeleton implementations. HDC focus mainly on the divide and conquer paradigm and provide efficient implementation of specific cases of this pattern such as constant recursion degree, fixed recursion depth or multiple block recursion... The HDC programming model uses a performance model that guide the division of the problem into sub-problems according to the available processing unit count.

Jaskel : The Java Skeleton (JaSkel) [FSP06] provides a set of skeletons such as pipe and farm. Skeletons can be specialized using inheritance. When using each of these skeletons, programmer is responsible of implementing the used skeleton interface to encapsulate his application-specific code. Skeleton nesting can be done through basic Java class system. JaSkel skeletons are provided in three versions: sequential, parallel in shared and distributed memory versions such as OCamlP31 [CMV+06]. We note that distributed computations are supported in JaSkel through aspect-oriented programming, more precisely using AspectJ.

Alt and HOC : *Alt* have proposed a series of skeleton frameworks for distributed memory systems and grids [AG03]. Alt provides a set of skeletons in form of services that are accessible through Java Remote Method Invocation [AG05]. A client program can find these services and call them remotely through an invoke API. Data between program and services is exchanged through special container objects. Alt skeletons are not nestable, hence, control flow between skeletons should be explicitly specified in the client application. The Higher Order Component (HOC) attempted to conceptualize the *Alt* approach by combining skeletons, components and services to enable the remote client to access to distant services that implement parallel constructs [DG04]. The execution flow between different skeleton services is regulated by a dedicated description language.

Lithium and Muskel : Lithium [ADT03] and its successor Muskel [ADD07] are two skeltons frameworks developed at University di Pisa. They provides a set of nestable task-parallel and data-parallel skeletons as a Java library and a set formal semantics [GVL10]. A performance model based on skeleton rewriting techniques, task look-ahead, and server-to-server lazy binding [ADT03] is used to optimized performances of the target program. In term of implementation, Lithium exploits macro-data flow to capture parallelism. Muskel provides a set of features oriented toward performance and fault tolerance such as quality of service [Dan05], security between task pool and interpreter [AD07] and resource discovery, load balancing and fault tolerance when interfaced with Java JNI Technology. More recent researches from the Pisa group has addressed autonomic components [ADK09], skeletal extendability, and behavioral skeletons [GVL10].

FastFlow : The FastFlow framework [ADKT12] is another algorithmic skeleton programming framework developed at the Departments of Computer Science of the Universities of Pisa and Torino. FastFlow aims to promote high level parallel programming and provides a collection of skeletons and patterns to express stream-parallelism, data-parallelism and data-flow parallelism by providing a collection of skeletons. FastFlow offers a set of high-level parallel patterns such as "*parallel_for*" and "*parallel_forReduce*". In recent researches, GPGPU support has been introduced in later versions of FastFlow.

llc : The llc language [DGRDS03] uses OpenMP-like syntax to implement skeletal algorithms. C language and MPI are used as a host language. For a given set of skeletons, the MPI code is generated automatically [RDAS09]. llc has been successfully used to implements many irregular linear algebra problems [DBQODS07].

P3L : P3L is a skeleton based coordination language [BDO+95]. P3L furnishes a set of skeletons to coordinate the sequential parallel execution of C code. A P3L dedicated compiler uses a set of implementation templates to compile P3L code for a specific target architecture. Hence, each skeleton has a set of different implementations that are optimized for different architectures. The templates that implement a skeletons provide a performance model that can optimize program performance through guided program transformations [BCD+97]. P3L provides a set of modules that correspond to a set of skeleton constructs with input and output streams. Data parallel skeleton can be nested inside task parallel skeletons. When the programmer specifies the type of the input and output data streams, a type verification is performed at the data flow level [GVL10].

SkIE : Similarly to P3L, the Skeleton-based Integrated Environment SKIE [BDPV99] is a coordination language. SKIE provides a graphical user interface and a set of debugging and performance analysis tools. Hence, the programmer interacts with a graphical tool to compose parallel modules which are based on skeletons.

SAC : The Single Assignment C language (SAC) is an imperative language dedicated to array computing. It supports data parallelism through multi-threaded operations on vectors and parallel loops. SAC uses a HPF-like syntax and is implemented on a top of a host language, it relies on Pthreads for multi-threading primitives [Gre99] and supports many multi-core architectures [GJP10].

SCL : The Structured Coordination Language (SCL) [BK96] is one of the first skeletal programming language. SCL is designed to act as a base language which can be integrated with a host language such as Fortran. The SCL is based on three types of skeletons: computation, elementary and configuration. The computation skeletons are task parallel skeletons that specifies the computation control flow. Elementary skeletons are a set of data parallel skeletons such as *scan*, *map* and *fold*. Finally, the configuration skeletons offer an abstraction of commonly used data structures such as parallel or distributed array namely ParArray. While SCL skeletons are instantiated in Fortran, SCL primitives cannot be invoked directly from Fortran [DGTY95].

SKElib : SKELib [DS00] inherits the contributions of both P3L and SkIE among others. However, SKELib does not use any coordination language. Instead, a collection of skeletons is provided through a C library. The achieved performances are comparable to those of P3L.

Intel Thread Building Blocks Threading Building Blocks (TBB) [Phe08] is a parallel programming C++ template library developed by Intel. The library aims to offer an expressive way of exploiting recent multicore systems. TBB can be considered as a skeleton library since it offers a set of concurrent data structures and algorithms which abstract low-level threading details. For instance, data structures include "*concurrent queue*", "*concurrent map*", and "*concurrent vector*". Parallel algorithms include parallel loop implementations such as "*parallel for*", "*parallel while*" or "*parallel do*". Algorithms collection includes also several other parallel skeletons such as "*parallel pipeline*", "parallel sort" or "parallel scan". TBB provides also a set of useful classes such as scalable memory allocators, mutual exclusion mechanisms or thread-safe container. In TBB, computations are encapsulated in "*tasks*" to abstracts the access to the processors. The TBB run-time system is responsible of scheduling dynamically these tasks on the available processors while trying to use CPU caches efficiently.

Parallel Patterns Library (PPL) The Parallel Patterns Library (PPL) [CM11] is a C++ library developed by Microsoft. It was first integrated in Visual Studio 2010. PPL is based on an imperative programming model that aims to offer scalability and ease of use. In order to abstract low-level threading details, the library uses the *Concurrency Runtime* which is responsible of task scheduling and resource management. Similarly to TBB, the PPL offers a collection of concurrent containers and parallel algorithms. PPL support task parallelism and provide a set of data parallel algorithms that act concurrently on collections of data.

2.2.2.5 Comparison of Structured Parallel Programming Frameworks

Table 2.1 is an updated version of the comparative skeleton framework table from [GVL10]. It gives a comparison between the different skeleton libraries and languages presented in the previous section in term of :

- Programming Language
- Execution Language
- Distribution Library
- Type Safety
- Skeleton Nesting (Composition)
- Supported Skeleton Set

	Programming Language	Execution Language	Distribution Library	Type Safe	Skeleton Nesting	Skeleton Set
Alt	Java	Java	Java RMI	Yes	No	Map, Zip, Apply, Scan, Sort, Reduce, Replicate
ASSIST	Custom Control Language	C++	TCP/IP SSH/SCP	Yes	No	Seq, Parmod
Calcium	Java	Java	ProActive	Yes	Yes	Seq, Pipe, Farm, For While, Map, D/C, Fork
Eden	Haskell Extension	C	PVM MPI	Yes	Yes	Map, D/C, Pipe, IterUntil, Torus, Ring
eSkel	C	C	MPI	No	Yes	Pipe, Farm, Deal, Butterfly, HallowSwap
HDC	Haskell (Subset)	C	MPI	Yes	Yes	Map, Red, Scan, Filter, dcA, dcB, dcD, dcE, dcF
HOC	Java	Java	Globus	No	No	Farm, Pipe, Wavefront
JaSkel	Java	Java	RMI	No	Yes	Farm, Pipe, Heartbeat
Lithium	Java	Java	RMI	No	Yes	Farm, Pipe, Map, Reduce
Mallba	C++	C++	NetStream MPI	Yes	No	Exact, Heuristic, Hybrid
Muesli	C++	C++	MPI OpenMP	Yes	Limited	Array, Matrix, Farm Pipe, Parallel Comp.
Muskel	Java	Java	RMI	No	Yes	Farm, Pipe, Seq, Custom
P3L	Custom Control Language	C	MPI	Yes	Limited	Map, Reduce, Seq, Comp Pipe, Farm, Scan, Loop
QUAFF	C++	C	MPI	Yes	Yes	Seq, Pipe, Farm Scm, ParDo
SAC	Custom	C-like	Threads	No	No	GenArray, ModArray, Fold
SCL	Custom Control Language	Fortran	Ad-hoc Tools	Yes	Limited	Map, Scan, Farm, Fold, SPMD, IterateUntil
Skandium	Java	Java	Threads	Yes	Yes	Seq, Pipe, Farm, For, While, Map, D/C, Fork
SKELib	C	C	MPI	No	No	Pipe, Farm
SkeTo	C++	C++	MPI	Yes	No	List, Matrix, Tree
SkeIE	GUI / Custom Control Language	C++	MPI	Yes	Limited	Pipe, Farm, Map, Reduce, Loop
Skil	C Subset	C	-	Yes	No	ParData, Map, Fold
SkiPPER	CALM	C	SynDex	Yes	Limited	Scm, Df, Tf, InterMem
TBB	C++	C++	-	Yes	No	For, While, Reduce, Scan, Sort, Pipeline
XPU	C++	C++	-	Yes	Limited	sequential, parallel, pipeline , parallel_for, parallel_vector

Table 2.1: Comparative table of the algorithmic skeleton frameworks.

2.3 Discussion

High level parallel programming with skeleton and patterns address many limitations of the traditional low-level thread-lock parallel programming model. In the later programming model, parallel paradigm-related code is merged with the code of the application. Skeleton programming decouples the parallelism specification from the application-specific code: common parallel computation patterns are identified and provided as a set of abstract and reusable parallel programming patterns. Naturally, this approach can offer many advantages in terms of programmability, flexibility and hardware adaptivity. Yet, the skeleton frameworks presented in this chapter expose different trade-off between them and focus mainly on one of these aspects.

2.3.1 Programmability

One of the major advantage of the skeleton approach is its high programmability that improves the programmer productivity by relieving him from managing low-level multi-threading details and routines such as thread creation, communication and synchronization.

One of the major obstacle for achieving high programmability is the poor parallelism expressiveness of traditional sequential programming languages: parallelism expression using traditional languages is often too verbose, error-prone and requires deep restructuring of the sequential code. For this reason, many of the enumerated parallel skeletons frameworks either introduced new parallel programming languages such as *P3L* or extended the existing programming language with custom syntax extensions and compilation directives to offer better parallelism expressiveness... Consequently, these frameworks introduced custom compilers and tools designed to handle these parallel extensions at the cost of limited portability. Despite the exposed parallel programming interfaces of these frameworks are often less verbose than the traditional low level parallel programming model, the programmers are often reluctant to learn new languages, extensions and paradigms. This is one of the reason that prevented many skeleton frameworks from finding their path to the industry and mainstream programming. In our professional experience within the **Thales Airborne Systems** corporation, we observed that the programmers who developed applications using popular and mature programming languages are reluctant to learn new languages or use "*young*" specialized compilers and tools because of productivity, reliability and portability considerations.

2.3.2 Flexibility

Many programming models focus on a specific type of parallelism and offer a set of skeletons that target that type of parallels, for instance *SkePU* [EK10] furnishes mainly data parallel skeletons and their implementations on various multicore and manycore architectures, and do not offer skeletons for pipeline or task parallelism. In such case, the applications that do dot expose massive data parallelism cannot take advantage of

such library. Consequently the target application should exhibit enough data parallelism to justify the use of this skeleton library. Moreover, other applications that may contain data parallelism in addition to other types of parallelism cannot be parallelized using exclusively *SkePU* and requires other ones to handle task or pipeline parallelism. This lack of flexibility limit the spectrum of the targeted applications and dissuade mainstream programmers from using such skeleton libraries.

2.3.3 Performance and Hardware Adaptivity

The "manual" performance tuning for specific architecture is one of the major drawbacks of low level parallel programming models such as POSIX Threads [NBF96]. Programmer tunes the application performance for the target platform by adjusting the threads count to the available processor count and mapping threads to specific processors to improve cache efficiency and data locality. When switching another platform with different processor count and cache topology, the same performance tuning process is reproduced.

Higher level programming models including skeleton frameworks offers enough abstraction to hide the underlying shared memory architecture details and thus allow dynamic performance tuning and relieve the programmer from adapting his application to different parallel hardware. Nevertheless, many of these programming models use performance tuning mechanisms that are limited to extracting the processor count and adjusting the thread count to improve dynamically the forward scalability. Other aspects such as adapting the task-processor mapping to the underlying processor cache topology to improve spatial and temporal data locality are critical for performance as we will demonstrate in the task parallelism chapter. Yet, the later cache-aware scheduling aspect is often either neglected or addressed with simplistic approaches by most of skeleton frameworks. These approaches are discussed in data and task parallelism chapters.

2.4 Motivation

Parallel programming using low-level programming model is still a hard task for the average sequential programmer. High-level structured parallel programming are in high-demand since they address many limitations of the traditional parallel programming models. The industrial context (*Thales Airborne Systems*) in which our researches took place confirmed this observation and outlined the need for a high-level parallel programming model which improves programmer productivity while delivering reasonable performances. This formulated the main motivation of our research work which is "*easing parallel programming without sacrificing performances*". The goal was to design a structured parallel programming model which offer a high programmability while delivering comparable performances to lower level approach. Design goals included:

1. Using a traditional programming language (C++) without any extension to provide good portability: many parallel programming models introduce new pro-

gramming languages, extends existing languages or define compiler directives and thus requires specialized compiler and tools, this can limit their portability. Moreover, the low technological maturity of such new compilers can make questionable their use in critical industrial applications.

2. Easing the parallelization of existing sequential application by reusing the legacy code with the minimum possible alteration: when parallelizing sequential application, many parallel programming models requires deep restructuring of legacy sequential code.

3. Easing parallelism expression by offering an intuitive and compact parallel programming interface: parallel programming models which use traditional programming languages suffer often from poor parallelism expressiveness and expose a verbose programming interface.

4. Designing a flexible parallelization methodology which allows parallelization of general-purpose application which may expose different types of parallelism at different levels of granularity. Many parallel programming models introduces parallel paradigms that target specific application domains and focus on specific type of parallelism or target only popular parallel problems. Consequently such paradigms are hardly applicable to general-purpose applications, therefore the programmer faces a complex productivity-performance trade-off where they should extract enough parallelism to justify the use of a dedicated parallel library.

In this next chapters we present two C++ parallel programming frameworks namely XPU and FATMA which have been designed based on the previously enumerated design goals to address the limitations of many state of the art parallel programming models in term of programmability and performance. Both XPU and FATMA exploit exclusively the potential of standard C++ programming language and its metaprogramming capabilities to ease parallelism expression while delivering performances which are comparable to low-level programming models.

Part II

Explicit Parallelism Expression : XPU

3

XPU Methodology and Architecture

3.1 Parallelization Methodology

XPU proposes a task-based parallelization methodology that consists in decomposing the target sequential program into a set of tasks then specifying parallelism of these tasks. XPU allows the programmer to specify different types of parallelisms at different granularity levels to form a structured parallel program. The next two paragraphs describe the task decomposition and the parallelization process.

3.1.1 Task Decomposition

Task-based programming is based on the decomposition of a program into a set of tasks that cooperate with each others to perform the main work of the application program. Task granularity can be controlled and specified by the programmer: a program is basically the main task which is split into several coarse-grain tasks which may be split, in turn, into finer-grain tasks, and so on... until we reach the finest-grain allowed by the host programming language (Fig. 3.1).

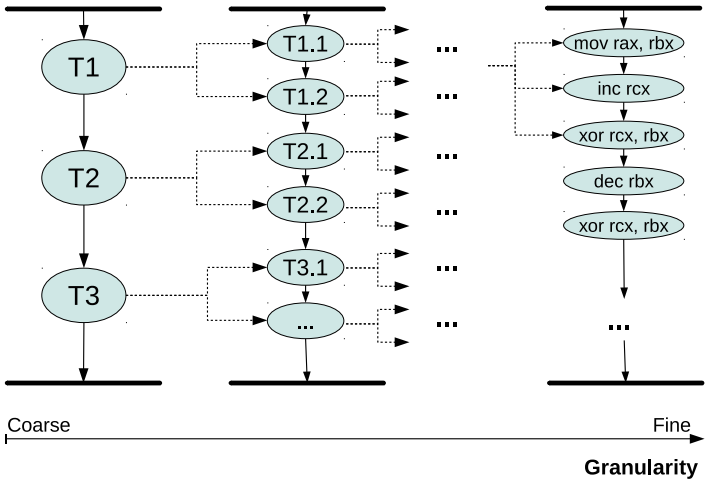


Figure 3.1: Program can be decomposed into a set of tasks at different granularity levels

Depending on the used programming language, these tasks can correspond to different pieces of code of various granularities such as C/C++ function, C++ object method, C++0x lambda expression or simply a set of machine instructions.

3.1.2 Parallelization

Each task of the application program performs a piece of work in which it may consume or produce data, i.e., read or write private or shared data. In order to speedup program execution on parallel computing architectures, we have to extract the maximum amount of parallelism. The ideal case is the one in which all tasks, at the finest possible granularity level, do not display any data or control dependencies, so they can be executed simultaneously (Fig. 3.2).

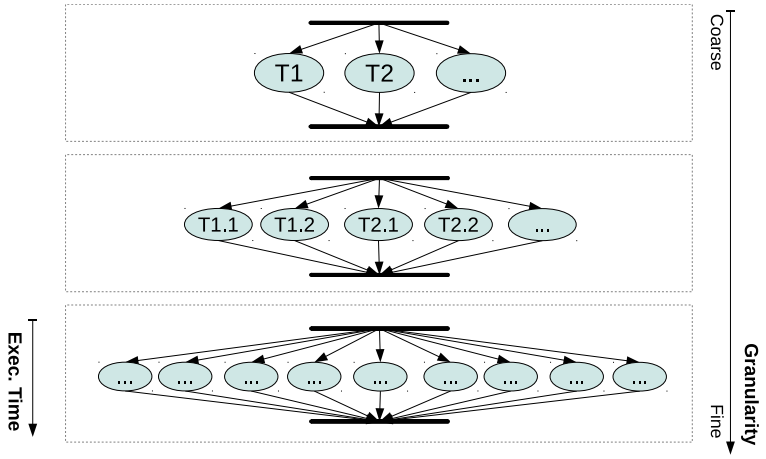


Figure 3.2: In ideal parallel program, finest grain tasks can be executed simultaneously.

Unfortunately, real world programs are "more-or-less" parallelizable depending on their natures: while many scientific simulations exhibit massive data parallelism and thus are highly parallelizable, many other general-purpose applications, which represent the vast majority of the softwares, are much less parallelizable due to data and control dependencies and explicit task ordering. Indeed, these algorithmic constraints introduce the need for synchronization and ordering to preserve memory coherency and algorithmic consistency. Consequently, each subset of the tasks composing the program can be executed either in parallel or sequentially depending on these constraints which define thus the parallel-sequential code ratio or the available parallelism in the target program.

At the end of the parallelization process, we obtain a hybrid execution graph containing both sequential and parallel sections (cf. Figure 3.3). The available parallelism varies depending on the nature of application, but the model remains usable for either highly or weakly parallelizable programs and even for those which are fully sequential.

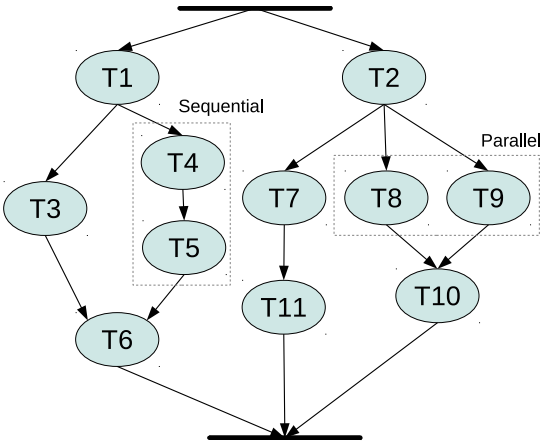


Figure 3.3: Realistic parallel program contains both parallel and sequential tasks at different granularity level due to the potential dependencies between some tasks.

Parallel programs can exhibit different types of parallelism that specifies specific parallel execution configurations such as:

- **Task Parallelism** where different tasks of the program can be executed simultaneously while other are executed sequentially, task parallelism is often specified through task graphs.
- **Data Parallelism** where a same task can be executed on different chunks or partitions of large data. Data parallelism can be implemented at thread level using a coarse grained task or at instruction level using vectorization (SIMD).
- **Pipeline Parallelism** where different tasks are executed asynchronously as pipeline stages, the pipeline specifies a consumer-producer relation between its different stages.

3.1.3 XPU Program Representation

Based on the previous observations, the XPU programming model represents the parallel program as a set of tasks, these tasks are organized within "*task groups*". XPU propose different implementations of "*task groups*", each one implement a specific parallel or sequential execution configuration. In addition to sequential execution, many of these task groups can specify different types of parallelism including task, data and pipeline parallelism as shown in the Figure below. We note that these task group implementations can be easily extended to support more parallel execution configuration.

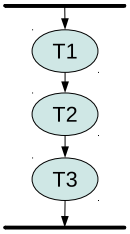


Figure 3.4: Sequential Execution

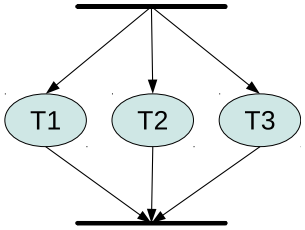


Figure 3.5: Parallel Execution

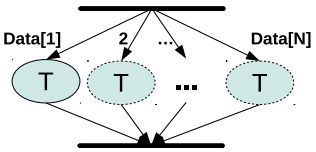


Figure 3.6: Data Parallel Execution

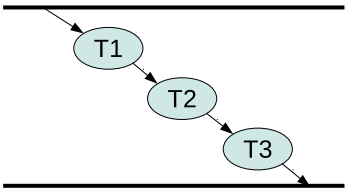


Figure 3.7: Pipeline Execution

In order to be able to express different kinds of parallelism at different levels of granularity, task groups can be composed hierarchically. Figure 3.8 depicts and example of hierarchical task group composition that allows programmer to specify task, data and pipeline parallelism simultaneously. Based on this relatively simple intermediate program representation, XPU aims to provide a programming interface which can express the different types of parallelism and to compose them hierarchically at the cost of the littlest possible amount of extra-code while reusing legacy sequential code without alteration.

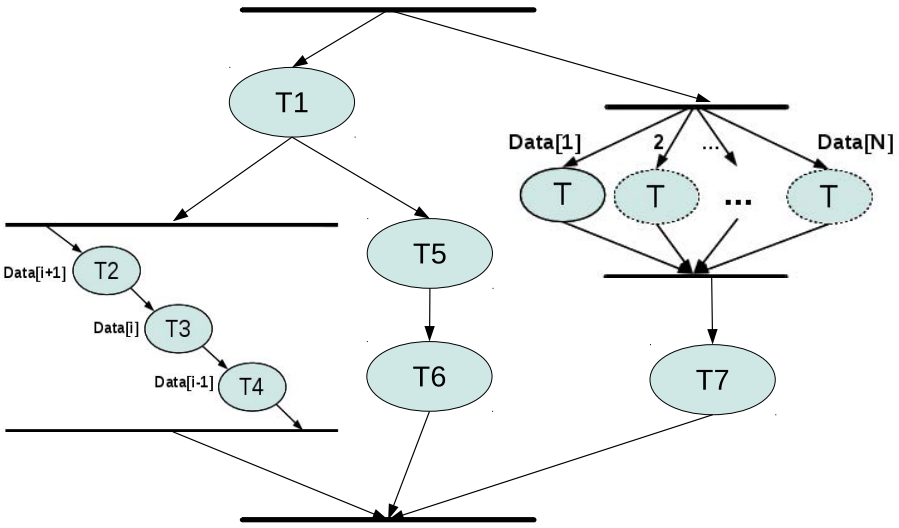


Figure 3.8: Parallel program combining several types of parallelism at different granularity levels.

3.2 XPU Architecture

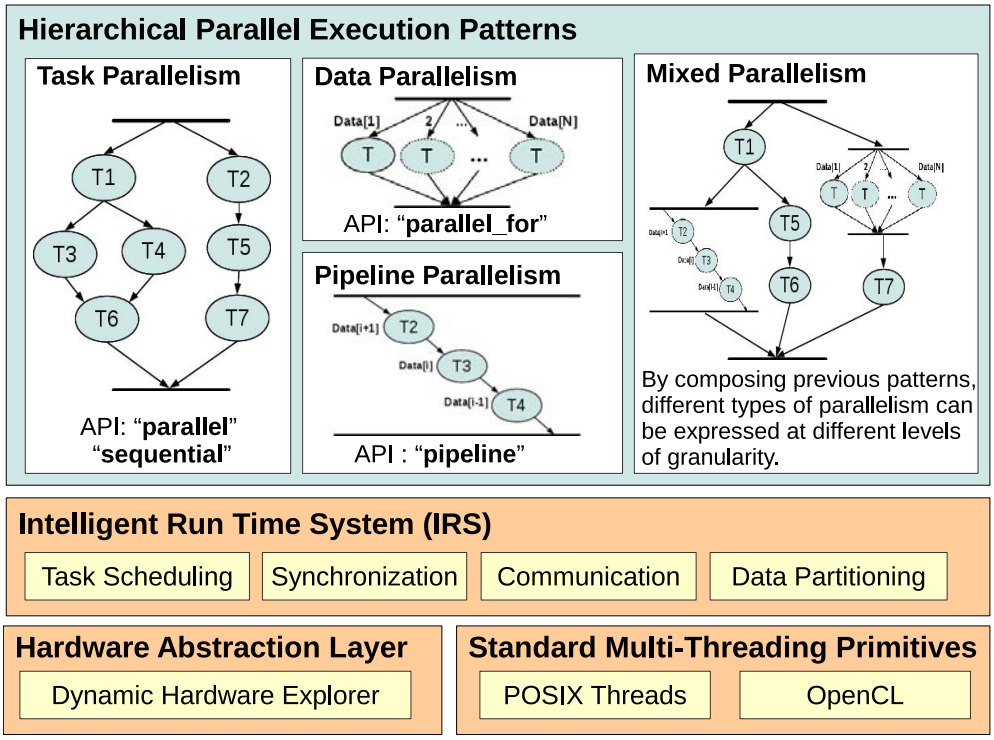


Figure 3.9: Overview of the XPU Architecture

Figure 3.9 gives an overview of the architecture of XPU. It is mainly composed of three layers. Each layer has specific design goals which are presented briefly in the following four paragraphs.

3.2.1 Execution Patterns API

The top layer of the XPU is composed of a set of light-weight API which allows the programmer to express different types of parallelism and compose them hierarchically in their program. The primary design goal of this API is providing high flexibility and expressiveness to improve programmer productivity.

The API is compact and can be composed of five keywords: "**task**", "**sequential**", "**parallel**", "**parallel_for**" and "**pipeline**". The first keyword allows the programmer to define a tasks by reusing his legacy sequential code. The remaining keywords are different implementations of task groups: "sequential" specifies sequential execution of a set tasks or task groups, "parallel" specifies parallel execution of a set of tasks or task groups, "pipeline" allows the use of tasks as stages of a pipeline execution configuration.

The "parallel_for" is another task group that specify the parallel execution of the same tasks on different partition of a large data.

3.2.2 Intelligent Run-Time System (IRS)

The IRS is the intermediate layer which is responsible of extracting information about parallelism specification and task-data dependencies from the top layer (API) and the information about parallel hardware from the bottom layer (HAL) to exploit them to achieve efficient execution. For instance, the IRS exploits information on processor cache topology to perform efficient cache-aware scheduling such as in our XPU CHATS algorithm that will be detailed in the chapter 5 (Data Parallelism). Also, the IRS uses information on available processing unit count to perform efficient dynamic data partitioning in order to achieve good scalability over a wide variety of architectures.

3.2.3 Hardware Abstraction Layer (HAL)

The HAL is the bottom layer which is responsible of exploring dynamically the underlying parallel hardware in order to extract hardware information such as processor count, cache topology and vectorization capabilities (cf. Chapter 4 and 5 : "Task Parallelism" and "Data Parallelism"). Hardware description is then made available to the IRS to optimize the execution according to that description.

3.2.4 Standard Multi-Threading Primitives

Standard Multi-Threading Primitives is a part of the bottom layer and is responsible of the abstraction of the low level multi-threading routines (thread creation, synchronization mechanisms, mutual exclusion...etc) provided by modern multiprocessing-capable operating systems. This layer is designed mainly to ensure portability of XPU to other systems.

We note that the current implementation has been developed on UNIX-like systems, however, due to limited libraries dependencies (POSIX Threads (PThreads) and optionally OpenCL for GPU computing support), XPU is easily portable to most of modern systems since multi-threading libraries are available on most platforms.

3.3 The Hierarchical Task Group Graph

In order to accommodate heterogeneity of the different execution patterns, so that they can fit into a single homogeneous structure representing the program, we define a common abstract construct named "*task_ group*". All our execution patterns implement this common interface: for example "*sequential_ tasks*" is a group of tasks scheduled to run sequentially while "*parallel_ tasks*" is a group of tasks scheduled to be executed simultaneously (a basic fork and join pattern) and "*pipeline*" is a group of communicating tasks running as a chain of overlapping processing stages... etc. We note that

"*Task*" is also, by design, a "*task_ group*" containing a single task.

Figure 3.10 gives an overview of the different implementations of the abstract task group interface. The flexible HTGG design allows the extension of the task group implementations to express more execution patterns and meet specific programmer needs in various application domains.

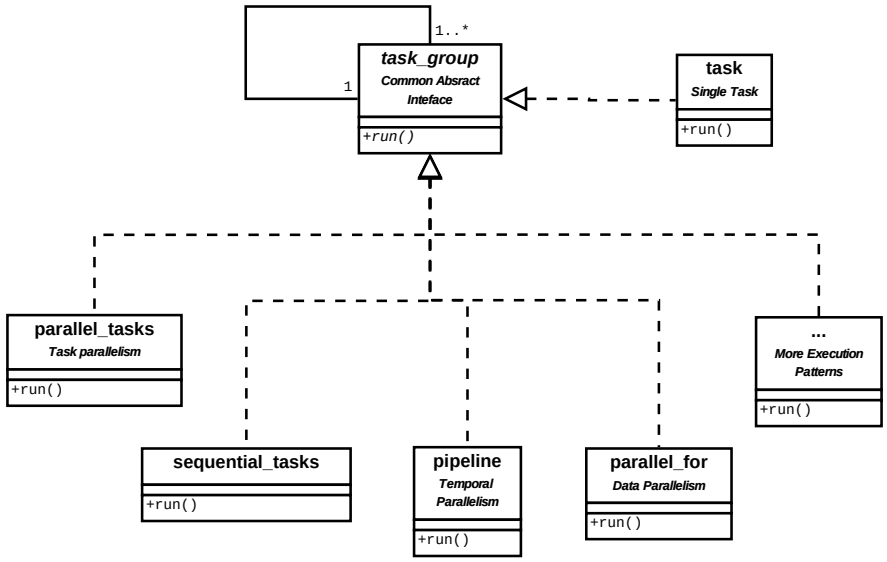


Figure 3.10: Different implementations of the Hierarchical Task Group Graph

Thanks to the flexible design of the HTGG (cf. Figure 3.10), most of the provided constructs are nestable and can be composed hierarchically inside each other. By expressing parallelism at several levels of granularity using these patterns, we obtain a hierarchical structure composed from task groups of "task_group" which we named HTGG. Task ordering is specified inside each construct, so when a task group is called, it executes its sub-task groups following the specified execution pattern and each of these sub-task groups will, in turn, do the same with their sub-task group ...etc.

In the next chapters, we detail the different components of our programming model starting from its elementary and atomic component which is the task up to the different parallel patterns which specify the execution of a group of tasks and implement the abstract task group interface. Use of these patterns is illustrated through examples of programs in each chapter. The use of these patterns and their composition is demonstrated through practical applications such as radar signal processing, polyphase filter bank implementation, blackscholes algorithm and fluid hydrodynamic simulation...

3.4 Overview of the XPU Programming Model

In this section, we present a set of figures which summarize the XPU parallelization stages from parallelism expression to actual tasks execution on the parallel hardware. Each of these illustrations is dedicated to a specific type of parallelism. Supported parallelism types are task parallelism, data parallelism and pipeline parallelism. Each of these kind of parallelism will discussed in detail in a dedicated chapter.

3.4.1 Task Parallelism

3.4.1.1 Sequential Execution

Figure 5.11 shows how sequential execution can be specified using the XPU API "*sequential*" and how the run-time executes the tasks on the target platform.

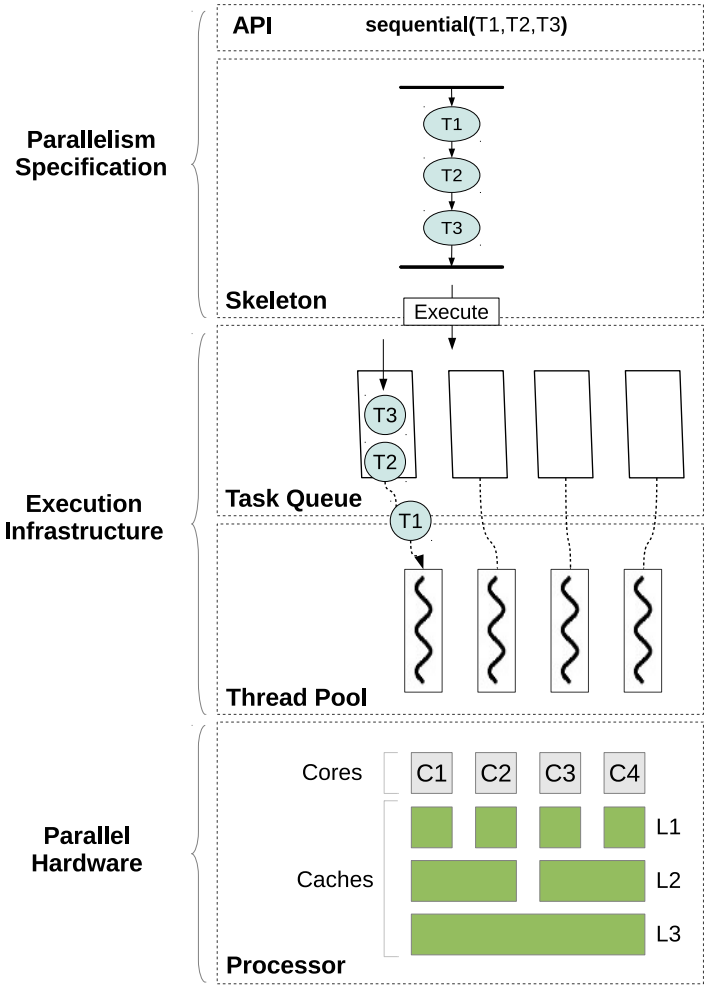


Figure 3.11: Sequential task execution details.

3.4.1.2 Parallel Execution

Analogously to the sequential execution, Figure 5.12 shows how parallel execution can be specified using the XPU API "*parallel*" and how the run-time executes the tasks concurrently on the underlying multicore architecture.

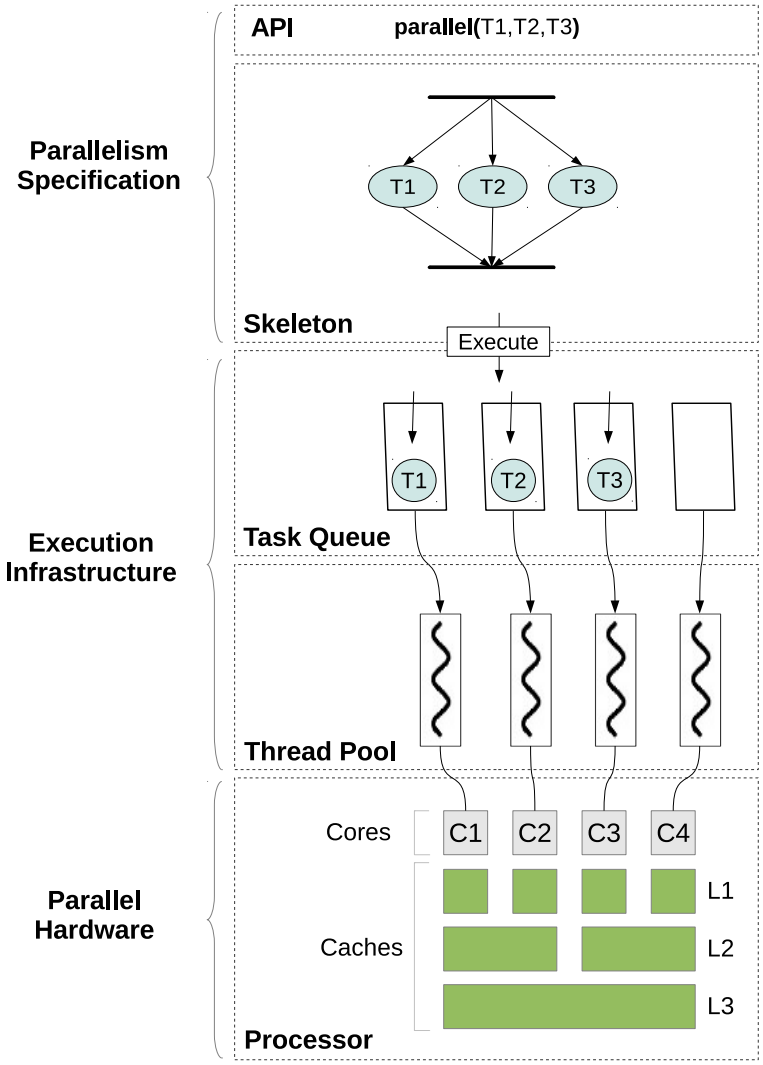


Figure 3.12: Parallel task execution details.

3.4.2 Data Parallelism

Figure 3.13 describes data parallelism expression using the XPU parallel loop implementation, namely "*parallel_for*" and details how the main workload is distributed across the available processors. We note that a smart task scheduling is performed by the IRS to exploit efficiently the processor caches. More details about this cache-aware

scheduling technique are provided in the description of the XPU CHATS algorithm in the "Data Parallelism" Chapter.

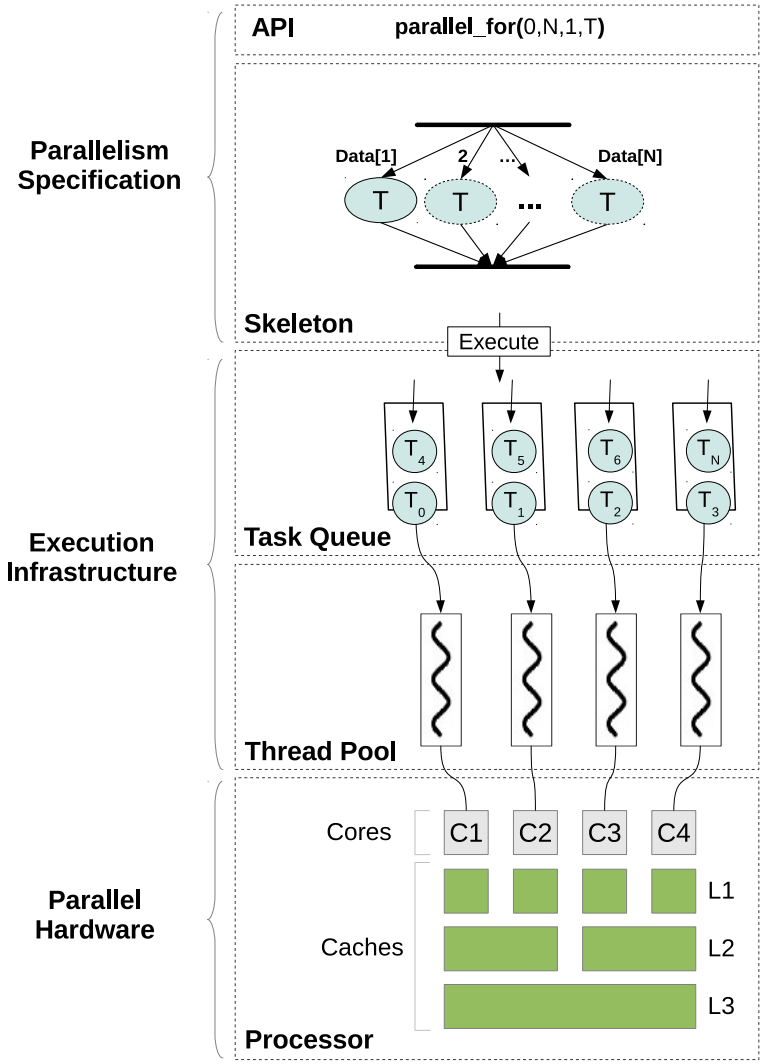


Figure 3.13: Parallel loop execution details.

3.4.3 Pipeline Parallelism

Finally, Figure 3.14 shows the pipeline parallelism specification in the XPU programming model and shows how the different pipeline stages are executed on the underlying multicore platform by the XPU runtime system.

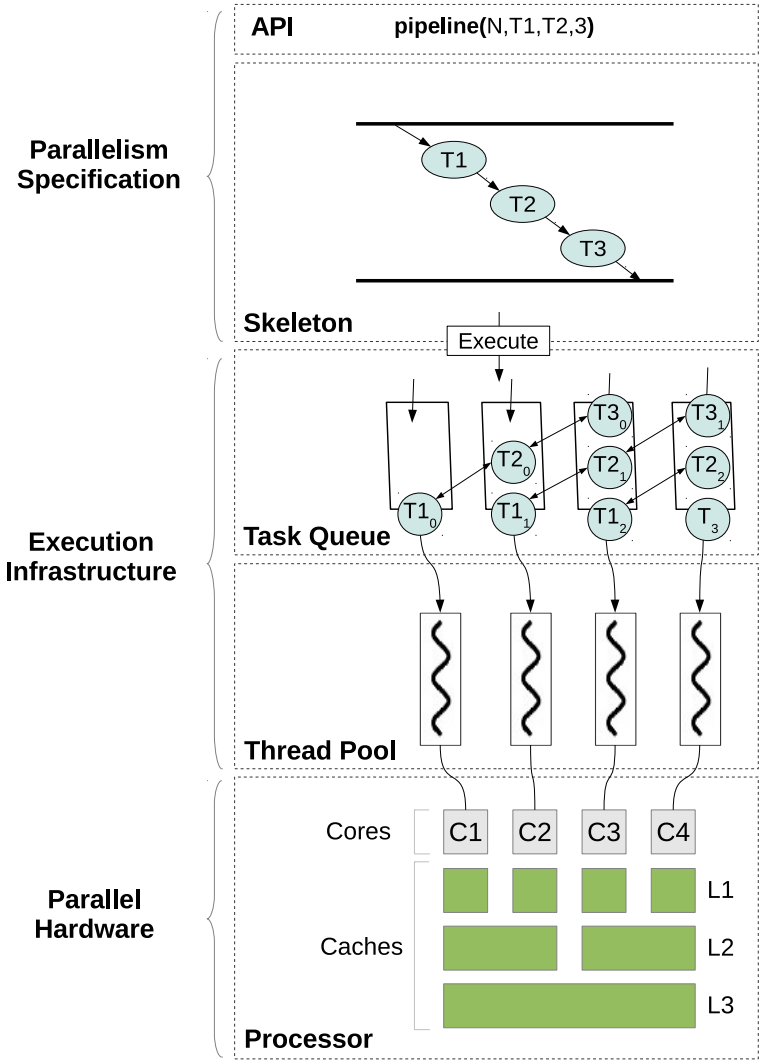


Figure 3.14: Pipeline execution details.

In the next chapters, we see how we reuse legacy code to define XPU tasks then how we can specify the parallelism of these tasks using the different XPU skeletons. Several applications are used to illustrate the use of XPU and to evaluate its performances and programmability in comparison to concurrent approaches.

4
Task Definition

Decomposing a program into a set of pieces of code is the first step in the parallelization process in most parallel programming models. In low-level thread-lock programming model, these pieces of code are called "*callbacks*" while in higher level task-based programming models these pieces of code are encapsulated in "*tasks*".

4.1 Task Definition and Programmabitlity

Task definition mechanism is critical for both programmability and performance. In order to define a *task* or a *callback* using an existing piece of code many low and high level parallel programming models require considerable amount of extra-code and a significant alteration of the legacy code. These two key considerations have a direct impact on programmability and developers productivity. In order to improve XPU programmability, two aspects have been considered as design key in both task definition and parallelism expression:

- Reducing the amount of the required extra-code to define a task.
- Promoting reuse of the sequential code with the lowest possible alteration.

In order to outline the programmability of our programming model, we compared it to the lower level parallel programming model PThreads [NBF96] then to a high-level task-based programming model named Intel Threading Building Blocks (TBB) [Phe08]).

4.2 PThreads Callbacks

In the traditional low-level PThreads programming model, the "*callbacks*" play the role of "*tasks*" in the task-based programming models. "*Callbacks*" corresponds to functions that are executed by the concurrent threads of multithreaded applications. When trying to reuse a legacy function as a PThreads callback, the original sequential code is often greatly altered since the targeted piece of code has to meet the native callback prototype "***void * function(void *)***" which imposes many restrictions on the programmer when parallelizing applications or reusing sequential code : only "*static*" functions can

be used as callbacks, the C++ object methods cannot be used directly. In addition, the set of data which is consumed or produced by the callback must be stored in a common intermediate opaque structure (*void **) then extracted and restored to their original type through type casting.

Listening 4.1 shows a simplified example of how a callback can be defined and how a thread is created in the *POSIX Threads* programming model. The intermediate steps of packing required arguments into a custom structure "args" in the main function and unpacking these arguments inside the callback function are not detailed in the given code.

The lack of flexibility of the *PThreads* programming model leads to many modifications of the legacy code when parallelizing a sequential code. Moreover, usually a lot of programming paradigm-related extra-code is introduced and make the code verbose, less readable and error-prone. This poor programmability amplifies the burdens on the programmer dramatically and makes the reuse of sequential code difficult.

```
1
2  void * callback(void * args)
3  {
4    // unpack arguments then
5    // call the original code
6  }
7
8  void main()
9  {
10   int   arg1, arg2;
11   float arg3;
12   pthread_t     id;
13   pthread_attr_t attr;
14   struct custom_struct args; // custom structure to pack all arguments
15   // pack arguments
16   args.first = arg1;
17   args.second = arg2;
18   args.third = arg3;
19
20
21   pthread_create(&thread, &attr, callback, (void *)&args);
22
23   // ...
24 }
```

Listing 4.1: Callback definition and thread creation in POSIX Threads programming model

4.3 Threading Building Blocks Task

Intel Threading Building Blocks (TBB) [Phe08] is a high level programming model which provides more abstraction than PThreads and allows the reuse of sequential code in a less restrictive way. However, significant modification to sequential code is still required: task code and its consumed or produced data should be encapsulated in a specific task class respectively as object attributes and object methods. Consequently, sequential code cannot be reused directly and has to be significantly transformed. This leads to verbose code which is hard to read and which requires significant programming effort.

```
1  class task_1 : public tbb::task
2  {
3   public:
4     task_1(double * data_1, int data_2) : m_data_1(data_1), m_data_2(data_2)
5     {
6       // initialization
7     }
8
9     tbb::task * execute()
10    {
11       // ... function code ...
12       return 0;
13    }
14
15  private:
16
17    double * m_data_1;
18    int      m_data_2;
19 };
```

Listing 4.2: Task definition mechanism in Intel Threading Building Block.

4.4 XPU Task

Since promoting the reuse of sequential code is one of the primary design goals of XPU, we tried to overcome the previously enumerated limitations of PThreads or TBB through a more flexible task design. In XPU, a task is basically an abstract callable piece of code which can be executed. This piece of code may consume or produce data. Data is passed in the form of arguments to each task.

4.4.1 Using Legacy Functions

The programmer can encapsulate easily a legacy C/C++ function in an XPU task. Listing 4.3 shows how a C/C++ function can be reused as a task. Contrary to the POSIX Threads programming model which does not allow reusing functions which do not meet the callback prototype "*void * function(void*)*", XPU task can be defined by reusing any legacy function without introducing any constraints on its signature, e.g. its argument count or types and its return type.

Since XPU is implemented using an object-oriented language C++, XPU task is implemented internally as a C++ object which encapsulates the target piece of code, however, it does not requires that programmer write a class which implement a specific task interface such as in the TBB programming model and most of the parallel programming models which are implemented using object oriented languages such as C++ and Java.

As shown in Listing 4.3, the first parameter of the task is a pointer to the target function. The remaining parameters are the arguments which should be passed to the function when executing the task. Based on our personal experience, the easy reuse of functions as XPU tasks makes C programs good candidates for relatively fast parallelization since they are often well structured as modular functions.

```
1  // simple functions with different arguments count and types
2  int read(const char * file, int * stream)
3  {
4    // code...
5  }
6  int sort(int * data, int size)
7  {
8    // code...
9  }
10
11 int main()
12 {
13   int  size
14   int * data;
15
16   // tasks definition
17   task read_t(read, "file.dat", data);        // unnamed task
18   task sort_t(sort, data, size);              // task with different argument types
19   task named_sort_t("sort", sort, data, size); // named task
20
21   // running the task
22   read_t.run();
23 }
```

Listing 4.3: Task definition using legacy functions.

In order to ease application debugging, execution monitoring and performance tuning, the first parameter of task can be optionally a string which specifies the task name. Line 19 in Listening 4.3 shows an example of the definition of a task named "sort".

Task names can be used in the output traces of an XPU application when the monitoring mode is activated. The output traces tell the programmer when each task started and ended and gives the programmer an overview of the execution time of each defined task. By giving names to tasks, the programmer can identify more easily running tasks, can monitor their execution and can detect potential bugs or bottlenecks.

4.4.2 Using Object Methods

C++ Programmers organize often their programs in C++ classes and encapsulate their functions code in their classes as object methods. These object methods cannot be used easily as callbacks or tasks without restructuring the code. Particularly, the PThreads programming model and its derivatives do not support using object methods as callbacks else they are defined as public static methods. This implies that the target callback do not have direct access to the private object attributes.

Analogously to C/C++ functions, XPU allows the use of legacy C++ object methods as tasks. LISTING 4.4 shows how a task can be defined by reusing an object method. The first parameter of the task is a pointer to the class instance, the second parameter is a pointer to the class method and finally the remaining parameters are the arguments which should be passed to the object method when executed.

```
1  class image
2  {
3      public:
4
5        int sharpen(int val)
6        {
7        }
8
9        int blur(int x, int y)
10       {
11       }
12 };
13
14 int main( )
15 {
16    image img("img.jpg"); // object instantiation
17    task sharpen_t(&img, &image::sharpen, 10);
18    task blur_t("blur", &img, &image::blur, 4, 16);
19 }
```

Listing 4.4: Task definition using an object method.

4.4.3 Using Lambda Expression

Lambda expressions has been introduced in the later versions of C++ (C++0x and later). XPU allows the programmer to use lambda expressions as tasks. Listing 4.5 shows how a lambda expression can be used as XPU task.

```
1 int main( )
2 {
3   float * data;
4
5   task filter([](float * samples, float freq) { /* code */ }, data, 16000.0f);
6
7   filter.run( );
8
9 }
```

Listing 4.5: Task definition from a class method.

4.5 Design and Implementation Of The XPU Task

Using traditional and standard programming languages such as C or C++ offers excellent code portability over different compilers and systems and takes advantages of language maturity and sophisticated compilation and optimization techniques which have been developed during decades and are still constantly improved.

Many high level parallel programming models such as Intel TBB [Phe08] which uses the C++ programming language tried to exploit these traditional languages to provide a friendly parallel programming interface which can ease parallelism expression and hide low-level parallel programming-related details. Unfortunately, while displaying good portability and achieving high performances and good scalability, most of these high-level programming models suffer from verbose programming interface which introduces often a significant parallel paradigm-related extra-code and require significant alteration of the legacy sequential code.

Many other parallel programming models tried to accommodate the "lack of parallelism expressiveness" of traditional programming languages such C and C++ and proposed either new parallel programming languages or extended these traditional languages with compiler annotations or extra keywords to ease parallelism expression and guide compiler when parallelizing code, for instance Intel Cilk Plus [Rob13] which extended C++ syntax with custom keywords is an example of such programming models. This approach suffers from limited portability over different compilers and platforms in addition to learning curve steepness: programmers prefer using well-known and mature traditional programming language rather than learning new languages or new paradigms introduced by various language extensions without any guarantee on achievable performances.

In our works, we tried to use exclusively standard C++ without any extension to provide a parallel programming interface with high programmability. Traditional C++ language is a rich programming language witch can provides enough expressiveness to allow easy parallelism specification at the cost of few extra-lines of code. We exploited the potential of C++ language and more particularly its meta-programming capabilities (see Annex A) to provide a friendly programming interface. The XPU API allows simple and fast definition of tasks at the cost of a single line of code. Moreover, it allows reusing legacy code without alteration and therefore improves programmers productivity. By using C++ template metaprogramming we take advantage of compile-time compiler optimization to produce efficient code.

4.5.1 Internal Task Design

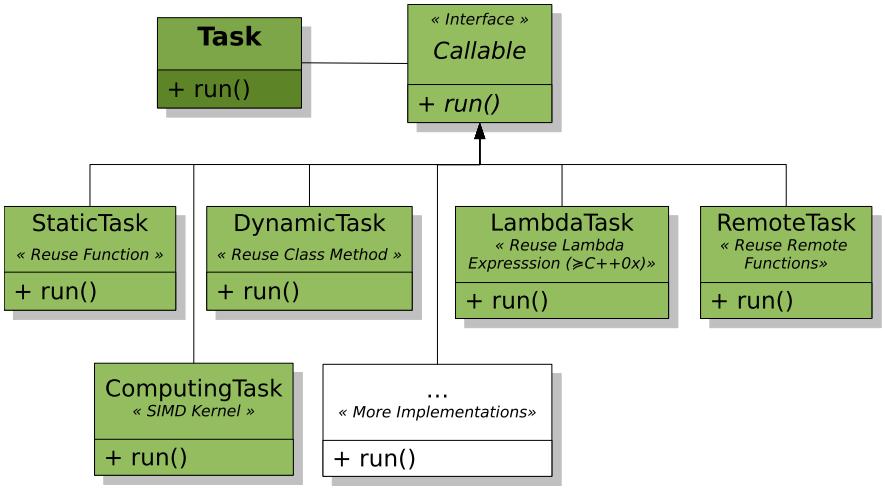


Figure 4.1: Task design allows reusing different pieces of code through an extendable set of implementations

Figure 4.1 gives an overview of the internal design of the XPU Task. In order to support multiple implementations, the *Task* object uses an abstract interface named *Callable*. Several implementations of this interface are provided to support different pieces of code. For instance, the *StaticCallback* implementation allows the reuse of classic C or C++ function as a task while the *DynamicCallback* implementation allows the reuse of C++ class methods etc... These implementations are responsible of allowing the main *Task* object to call or execute the encapsulated piece of code with the appropriate arguments. Each *Task* stores mainly a pointer to the target code and all the required data to call that code such as the arguments or parameters which are passed to the encapsulated function or object method when called. We note that auxiliary data may be stored to handle more complex implementations such as computing task or remote

task call.

In addition to calling the encapsulated code, the various task implementations are responsible of extracting the data dependencies of that code and should be able to run the code within a critical section defined by one or more "*locks*". The later functionality is required by several parallel patterns to provide the ability to protect data against conflictual concurrent accesses (*race conditions*) in certain cases. This feature is detailed in the chapter.

Finally, we note that arguments which are used when executing the encapsulated code can be modified before execution to meet the requirement of some parallel patterns such as the "*parallel_for*" and the "*pipeline*" patterns. The later patterns operate on indexable collection of data and may require a task to update the index of the data being processed before executing the processing function: when iterating through that indexable collection of data, the index passed to the function as an argument is constantly modified.

4.5.2 Task Implementation

4.5.2.1 Promoting Reuse Of The Legacy Code

Sequential C and C++ programs may includes several functions with various signature, i.e. functions with various return types and arguments count and types. When parallelizing these programs and trying to reuse these functions as tasks, the programmer is confronted to the lack of flexibility of many low and high level programming model as we have seen in the case of POSIX Threads or TBB. In both cases, programmers are constrained to rewrite a significant portion of their legacy functions in addition to writing paradigm-related extra-code. This amplifies the burdens of the programmer and reduce his productivity.

In order to address this problem, the task definition mechanism must adapt to different pieces of code including functions with different prototypes to relieve programmer from that burdens and allowing him to reuse his legacy sequential code ideally without any alteration and at the cost the less possible extra-code. More specifically, in the case of C++ functions and object methods, three issues should be addressed:

1. Argument type variability.
2. Argument count variability.
3. Return type variability.

In XPU, these issues are addressed using C++ template metaprogramming techniques (See Annex A) in the different task implementations. C++ template programming is used to address the argument type variability issue. C++ template specialization is exploited to address the second issue while the last issue is accommodated

through the use of C++ template classes. More information about C++ template programming and the various C++ metaprogramming techniques can be found in the Annex A at the end of this manuscript.

4.5.3 Transparent Task-Data Dependencies Extraction

4.5.3.1 Task Data Dependency Information and Its Use

Task data dependency is a valuable information that can be used for multiple purposes including improving execution efficiency through cache-aware scheduling. For instance, as we will see in the next chapter, task data dependency can be used to perform smart task-processor mapping to improve temporal and spacial data locality in processor caches, tasks using the same data can be scheduled to run on the same core to promote data reuse and reduce communication overhead.

Moreover, task data dependency can be used to automate shared data management routines to reduce programmer burdens and improve his productivity. Ultimately, as we will see in the second part of this thesis which is dedicated to automatic task parallelization with the FATMA framework, the information on task data dependency can be used for transparent parallelization of a large sequence of tasks: knowing the task accesses to data, an intelligent runtime system performs advanced task dependency analysis to build a "*superscalar task graph*" that specifies parallelism.

4.5.3.2 Task Data Dependency Extraction

A The algorithm

In XPU, task data dependency extraction is performed by analyzing the arguments that are passed to the task. When defining an XPU task, data access is specified implicitly or explicitly through the passed arguments: arguments which are passed by value are considered as local read access data while arguments which are passed by pointer are considered as true dependencies. Arguments passed by value are local copies of the arguments provided by the caller of the task, thus they do not interfere with other tasks and do not introduce any potential dependency implying other tasks and can be simply ignored. However, pointer arguments are the addresses of shared memory that can introduce potentially dependencies between tasks. Consequently, differentiating values from pointers is a primordial step for data dependency extraction.

The task definition mechanism of XPU exploits C++ meta-programming techniques [Kos] to detect pointers in the task arguments list by performing Compile-Time Type Identification (CTTI) [Sin04]. The later technique is described in the next paragraph.

The detected pointer arguments indicate that task accesses the data but do not precise how the data is accessed, e.g in *read* or *write* mode. In order to extract the later information, XPU use the same CTTI technique to determine whether the pointer

are constant. XPU considers constant pointers as "*read only*" data while non-constant pointers are considered as "*read-write*" data. We note that the task definition API of XPU provides a macro, namely "*__read_only(x)*", that allows the programmer to specify explicitly that the data is accessed in "*read only*" mode when he define a task. In addition to value and pointer, in the C++ language, an argument can be a reference. The XPU do not handle reference arguments yet. Our first experimentation exposed a value/reference ambiguity issue with some compilers. Thus, the support of reference arguments has been omitted from the current XPU framework.

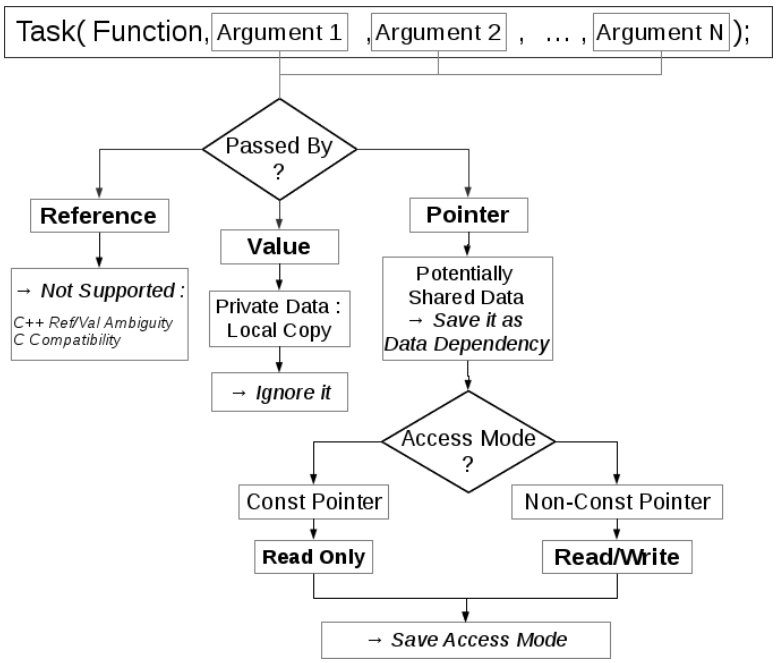


Figure 4.2: Task data dependencies detection

Figure 4.2 gives an overview of the XPU task data dependency detection algorithm. Thanks to the use of CTTI in the implementation of the algorithm, a significant part of the algorithm is executed transparently at compile-time, thus the algorithm do not introduce any significant execution overhead at runtime. Moreover, the task data dependency analysis is performed at the task definition stage, so before any task execution ("*offline*"), therefore in all cases, any execution overhead would not impact the actual task execution.

B Data Type Identification

In C++ programs, data type can be identified at run-time or at compile-time. Data type identification at run-time is known as Run-Time Type Identification (RTTI) and

is based on the *typeinfo* class of the standard C++ library. The *typeinfo* class can be used to retrieve information identifying a type. Its *typeid* method can be applied to any expression that has a type. It returns a null-terminated character string that describe the type of that variable or expression.

Our very early version of XPU used the RTTI technique to distinguish various data types and more particularly to separate pointers from values. Unfortunately, the RTTI approach suffers from two major drawbacks. The first is that RTTI introduce an execution overhead since it identifies types at execution time, the second disadvantage is the weak portability of programs using RTTI over different C/C++ compilers. For instance, we observed that different compilers return different character strings to describe the same type. For this reason we used the Compile-Time Type Identification (CTTI) which overcomes the RTTI limitations and offers many advantages despite its implementation complexity.

The CTTI technique relies on C++ metaprogramming techniques including template programming and functions polymorphism [Kos] to distinguish between *pointer*, *constant pointers* and *values*. Polymorphism is a programming feature that allows a code (function or class...) to behave differently in different contexts. The C++ programming language offers different means to exploits the polymorphism capability with functions or object methods. Function polymorphism in C++ is called function *overloading* and allows the compiler to select one function among a set of overloaded functions depending on the type and/or the count of the given arguments. The function is selected at compile-time and several optimizations can be performed by the compiler to produce an efficient code.

We combined function overloading with template programming to implement efficiently a smart CTTI mechanism able to recognize whether an argument is a pointer of a value independently of its type. Listing 4.6 shows a basic example of CTTI implementation. The function "*is_pointer*" has a first implementation which take a pointer as argument (line 4). A second implementation overalod the first one and take a value as argument (line 12). Consequently, the compiler will select automatically the first implementation when the function is called with a pointer parameter such as in line 23, otherwise it will select the overloaded implementation such as in line 24.

One of the major advantage of the CTTI approach is its nearly null execution overhead: the type identification is performed by the compiler at compile time so no execution is introduced at run-time. Moreover, the compiler can perform many optimization at compile-time to produce an efficient code. Consequently, the CTTI-based implementation of our Task Data Dependency extraction algorithm displays a near null execution overhead and offers a good portability over various C++ compilers including the GNU C++ Compiler (g++) and the Intel Compiler ICC.

```
1
2  // first implementation
3  template<typename T>
4  bool is_pointer(T * x)
5  {
6     printf("the parameter is a pointer.");
7     return true;
8  }
9
10 // overloaded implementation
11 template<typename T>
12 bool is_pointer(T x)
13 {
14    printf("the parameter is a value.");
15    return false;
16 }
17
18 int main( )
19 {
20   float * a;
21   float b;
22
23   is_pointer(a); /* displays "the parameter is a pointer." */
24   is_pointer(b); /* displays "the parameter is not a pointer." */
25
26   return 0;
27 }
```

Listing 4.6: A simple example of how to use C++ Compile-Time Type Identification to separate "*values*" expression from "*pointers*" expressions.

4.5.4 Extending Task Implementations

The task design provides a high abstraction and allows us to extend its implementations to support different types of codes. "*Remote Task*" and "*Computing Task*" are two implementations which has been proposed as a "*proof-of-concept*" to demonstrate the potential and the flexibility of XPU task design. These two experimental implementations has been developed for demonstrative purposes and their development has not been pushed farther since their are designed respectively for distributed systems support and code vectorization using just-in-time compilation which are out of the scope of our research works. Until now, our works focused mainly on parallel programming on shared memory multicore systems using standard C++ language without the use of any compilation technique or special execution environment. However, these imple-

mentations appeared to be promising and may be developed and discussed in depth in future works.

4.5.4.1 Remote Tasks

In order to support distributed memory systems such as clusters or grids, the XPU Remote Task implementation allows the execution of an XPU task on a peer machine or a remote cluster "*node*". The main XPU program is executed on the master node while a set of services are executed on the slave nodes. The main program can execute a remote task on a slave node using the remote service. Each service is capable of executing a limited set of tasks. We note that tasks are identified by their names.

When executing a remote task, the main program sends a request to the peer service and blocks waiting for its response. The request contains the task name and the data or the parameters of the task. The task is executed then the result is sent back to the master node.

```
1  int main( )
2  {
3    double * data;
4    int     size;
5
6    xpu::data d(data,size); // encapsulate data to allow data exchange between nodes.
7
8    xpu::init();          // init xpu and wait for services to register
9
10   task t("sort", d);
11
12   t.run(); // run on remote computing node: find a node capable of executing "add",
13           // call it (send input) then get the result.
14
15 }
```

Listing 4.7: This example shows how XPU Remote Task can be used on a master node.

Listing 4.7 shows how an XPU Remote Task can be defined on a master node and how the task can be executed by a remote service. Listing 4.8 shows how the task is defined and implemented in th remote service side. We note that the XPU runtime system determines on which node the task is executed depending on the available nodes and the execution capabilities of their services.

```
1  int sort(xpu::data d)
2  {
3    // ... code ...
4  }
5
6  int main( )
```

```
{
  xpu::init(); // init xpu and wait for services to register

  xpu::data d;     // define a new data object
  d.allocate(size); // allocate memory to receive the data from the master node.

  xpu::task t("sort", d); // task definition
  xpu::service s;       // create a new service
  s.register(&t);       // register the task
  s.start();            // start the service
}
```

Listing 4.8: XPU Remote Task implementation on a slave node.

Figure 4.3 gives an overview of the distributed service architecture. This service-based architecture can support static and dynamic distributed memory systems. Static distributed systems are constituted by a fixed number of nodes while dynamic distributed systems are composed of number of nodes that can vary dynamically at run time: new nodes can join or leave the distributed computing system dynamically.

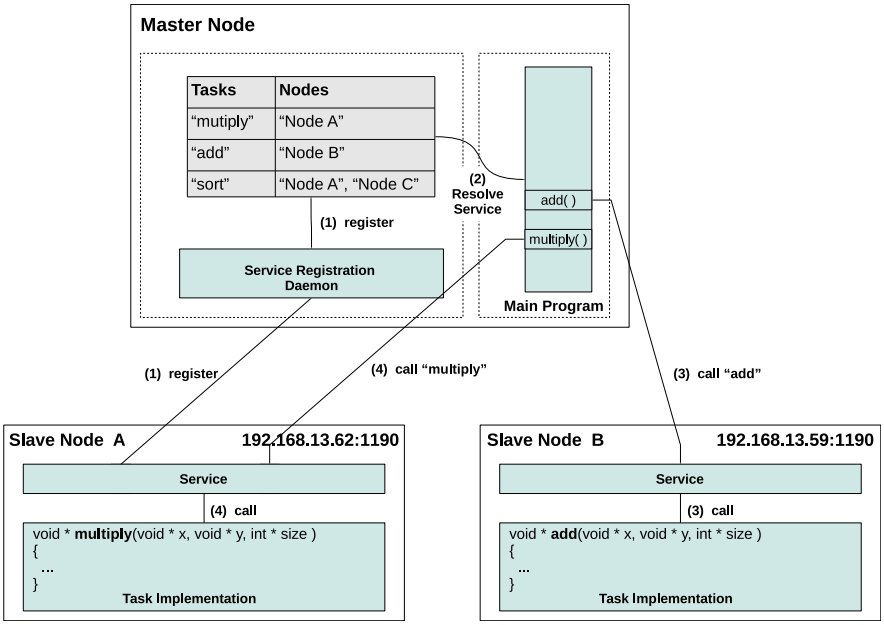


Figure 4.3: Service-Based Infrastructure For Executing Remote Tasks on Dynamic Distributed Systems

When a new node joins the distributed system, its associated service connect to the master node and register itself as an available service. Registration includes all required information such as address, execution capabilities, and names of the tasks that are

available on the slave node. These information can be extended with other useful details to improve execution efficiency.

When the main program need to execute a task with a given name, it checks the list of the registered services and find an available peer service which is capable of executing the task. Once the service selected, the task is submitted to the peer node to be executed. Our experimental implementation does not defines particular criterion for service selection. However criterion such as available processing units or communication latency can be a good selection criterions that can improve execution efficiency.

In the experimental implementation of the remote task, we introduced a registration daemon in the XPU run-time. The communication between the different nodes is based on the standard BSD socket environment and is thus portable to many systems. The POSIX Threads API is used for multithreading primitives.

4.5.4.2 Computing Tasks

Computing tasks is another experimental implementation of the XPU task interface. This implementation enables the programmer to create a task simply by specifying a computing kernel as a character string. The computation kernel operates on a large vector of data (simple or double precision *float*). Vectors are named and their names are used inside the kernel to resolve data addresses. This task implementation is designed mainly to implement data parallelism in a simpler way than verbose programming models such as OpenCL. While exposing a compact programming interface, the computing task do not allow yet the implementation of complex computing kernels such as in OpenCL or CUDA: the current kernel syntax is limited to simple math expression with basic mathematical operations.

The computing task implementation uses Just-In-Time (JIT) Compilation technique to generate the executable code that correspond to the specified computing kernel (the math expression). The generated code exploits the vectorization capabilities of x86 processor and use the SIMD instruction extension (x86 SSE). The JIT compiler generates an executable function that can be called when running the task. Being at the early stage of development, the current computing task implementation supports only instruction level parallelism (ILP) through the x86 SSE (version 2 or later) streaming instruction extension. Multithreading support will be introduced in later implementations. The current implementation uses a custom parser to handle basic math expressions and relies on the AsmJIT library [Kob] for just-in time compilation.

```
1 int main( )
2 {
3   int size = 1024000;          // vector size
4   float * x = new float[size]; // vectors
5   float * y = new float[size]; // ...
```

```
6   float * z = new float[size];
7
8   xpu::computing_task t(size);
9
10  t.add_var("x",x,size);      // adding variable names and addresses
11  t.add_var("y",y,size);      // to enable the JIT compiler to resolve
12  t.add_var("z",z,size);      // variable names in the kernel
13
14  t.set_kernel("x=y+x;");     // kernel setting and JIT compilation
15  t.run();                    // execute the kernel
16 }
```

Listing 4.9: Computing task allows the definition a computing kernel as a simple math expression. An internal JIT compiler is responsible of generating a vectorized executable code.

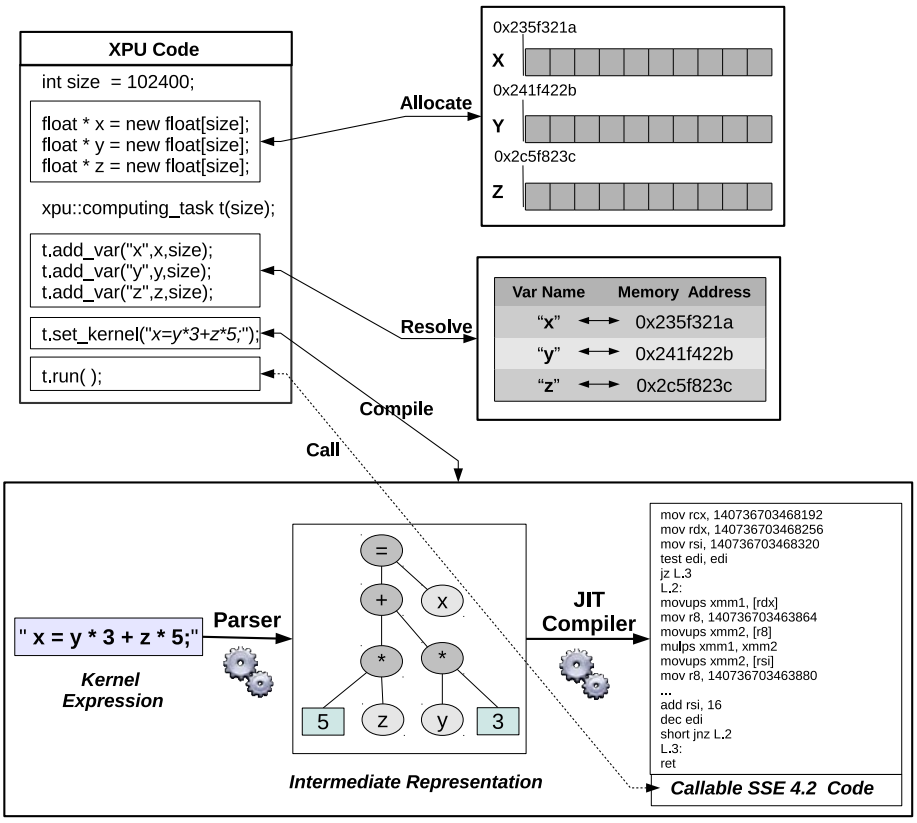


Figure 4.4: Implementation details of the XPU Computing Task

Listing 4.9 shows how a computing task is defined and executed. Figure 4.4 illustrates the implementation details of the computing task API. After allocating memory for the vectors (lines 6,7 and 8) on which the kernel will operates, the task is defined and

the vector length is specified (line 10). An association of the vector names and their memory addresses is given (lines 12,13 and 14) to allow the JIT compiler to resolve the variable names when parsing the kernel code. Finally, when setting the kernel or the math expression (line 16), the JIT Compiler compiles the kernel and generates the executable code. The generated code can be executed simply by invoking the "*run*" method of the task (line 18).

In the next three chapters, we see how we can specifies the execution configuration of a set of defined XPU tasks using the available XPU constructs. These parallel constructs allow the programmer to express different types of parallelism, namely "Task Parallelism", "Data Parallelism" and "Pipeline Parallelism". Each chapter is dedicated to a specific type of parallelism and presents the corresponding XPU skeletons. Several applications are used to evaluate the different skeletons in term of programmability and performances.

5 Task Parallelism

Task parallelism is a common execution pattern in parallel applications where tasks are organized as a task graph. The task graph specifies both parallel and sequential execution. As we have seen in Chapter 3, the HTGG allows flexible specification of task parallelism at all levels of granularity. As the number of tasks and the task graph complexity grow, parallelism expression become a hard and error-prone task: programmer is responsible of determining the task graph and handling tasks synchronization and communications at the different levels of the graph hierarchy. In this chapter, we discuss task parallelism expression and we describe the XPU API that allows explicit task parallelism specification. We show how we can improve programmer productivity through a light weight and intuitive API formed with only two keywords ("*parallel*" and "*sequential*") and designed for high programmability. We describe the execution infrastructure and we show how we can improve the execution efficiency of task graphs through cache-aware and load-balanced scheduling. We compare our programming model with the Intel TBB in term or programmability and performance.

5.1 Task Graph Representation

The parallelization process of a program starts with its decomposition into a set tasks. Dependencies between these tasks are then analyzed to extract task parallelism. A program may contain parallel and sequential regions: dependent tasks are executed serially while independent tasks can be executed simultaneously. At the end of this tasks dependencies analysis, we obtain a task graph that specifies parallelism at different granularity levels. Such task graph can be represented either as a superscalar task graph or as an hierarchical task graph. Both of these representations are introduced in the Chapter 2 of this dissertation. Figure 5.1 gives an overview of the difference between the two representations.

As illustrated in Figure 5.1, the hierarchical task graph is composed of hierarchical fork-join building blocks that use barriers to synchronize tasks at join points. Barriers can introduce idle times at the execution time, thus the later representation exposes poor execution efficiency in comparison with the superscalar representation that specifies peer-to-peer synchronization between tasks. However, "manual" parallelism spec-

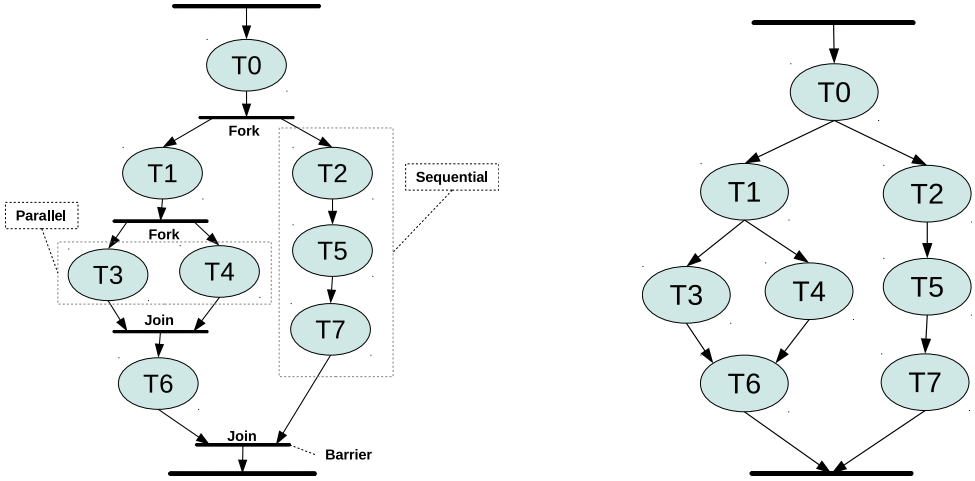


Figure 5.1: Unlike the superscalar task graph representation (at the right), the hierarchical task graph representation (at the left) introduces join points (barriers) where parallel tasks are synchronized. This can introduce unnecessary idle times at the execution time.

ification using superscalar task graphs can be a very hard task since it may require complex task dependencies analysis. For this reason, superscalar task graph construction is often automated using specialized parallelization runtimes, tools or compilers. For instance, we use this representation internally in our FATMA framework to perform automatic parallelization. Implementation details of FATMA are discussed in the third part of the thesis wish is dedicated to automatic parallelization.

The hierarchical task graph representation is more suited to "manual" or explicit parallelism expression: the task graph is hierarchical and is composed mainly of basic fork-join constructs at the different levels of its hierarchy, thus it is more adapted to highly structured parallel programming models. Since our primary goal is to ease explicit task parallelism expression, we use the hierarchical representation and we show how it can map to a very simple and light-weight API.

5.2 Parallelism Expression

As we have seen in our task graph example, parallel regions are composed of two or more parallel tasks that are executed simultaneously before synchronizing on a common join point. Specifying such parallelism between two tasks using low-level thread-based parallel programming models such as PThreads or Java Threading API translates into the creation of a thread for each task then the synchronization of these parallel threads on a common join point using barriers or thread join API. These operations can corresponds to dozens of lines of parallelism-related extra-code and the amount of extra-code

can grow quickly as the tasks number grows. Moreover, when addressing more complex cases where numerous parallel and sequential tasks are specified at various granularity levels, parallelism expression became even harder.

Directive-based structured parallel programming models such as OpenMP [DM98] can offer a simple mean for expressing parallelism in the simple case where parallelism between few tasks is specified: parallelism of few tasks can be specified using parallel OpenMP *sections* through a couple of lines of compilation directives or "*pragma*". However, in more complex cases implying hierarchical task graphs or nested parallelism, the corresponding OpenMP code can become much verbose and hard to read and maintain. Skeleton-based programming models can offer a good alternative through providing a set of nestable skeletons that hide parallelism specification and synchronization details behind a compact API. Yet, many popular skeleton-based programming models such as Intel TBB do not take advantage of this opportunity and expose a relatively verbose API as we will see in our experimental section.

We tried to address this programmability issue when designing the XPU programming interface through providing a particularly "compact" and intuitive API that allows the programmer to specify parallelism at the cost of negligible amount of extra-code while reusing his legacy code without any significant alteration.

If we observe the hierarchical task graph representation, we can notice that tasks are executed either sequentially or in parallel. By furnishing two skeletons that implements these two execution patterns we can be able to express both the execution modes. If in addition we can make these two skeletons nestable or composable we can be able to express parallelism of any hierarchical task graph.

We designed two skeletons named simply "**parallel**" and "**sequential**" that specifies respectively parallel and sequential execution. These two keywords can be combined to specify an hierarchical task graph where both parallel and sequential regions can be nested inside each other at different granularity levels. The three following paragraphs details the two skeletons and how they can be composed.

5.2.1 The Sequential Skeleton

The *sequential* skeleton specifies ordered execution of a set of tasks. It can be instantiated using the keyword "**sequential**", the resulting construct is an XPU *task_group* that can be nested in other parallel and sequential skeletons. Execution of the *sequential* skeleton corresponds to executing tasks in the specified order. Figure 5.2 shows the *sequential* skeleton and its corresponding XPU code. After the creation of the XPU tasks (line 8) using the three functions "f1", "f2" and "f3", the sequential skeleton is created (line 11). Once created, the skeleton is executed by calling the "run" method (line 13). When executing the skeleton, the tasks are executed in the specified order

```
// functions
int f1(char * data);
int f2(void);
int f3(void);

void main() {
 /* task definition */
 task t1(f1, data_1), t2(f2), t3(f3);
 /* task graph construction */
 task_group * program;
 program = sequential(&t1, &t2, &t3);
 /* task graph execution */
 program->run();
}
```

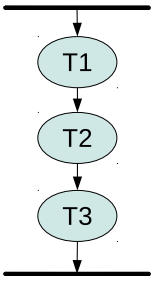


Figure 5.2: "sequential" specifies the sequential execution of a set of tasks.

and the construct wait until all tasks terminate before getting back to main execution flow.

When used alone, the sequential skeleton does not offer any advantage over serial function calls. This skeleton is designed to be used in combination with the other parallel skeletons. In particular, composing "parallel" and "sequential" skeleton allows the construction of task graph containing both parallel and sequential sections.

5.2.2 The Parallel Skeleton

The parallel construct specifies the simultaneous execution of a set of tasks. While parallelism is specified, tasks can be executed simultaneously or sequentially depending on the available processing units. In particular, when the number of parallel tasks is greater than the processor count, a subset of the tasks are executed sequentially while the other are executed simultaneously.

Figure 5.3 shows how the *parallel* skeleton can be constructed and executed using XPU. Firstly, XPU tasks are created from functions, then the "parallel" keyword is used to construct the skeleton. Once constructed, the parallel tasks can be executed by invoking the "run" method (line 13). The construct spawns the parallel tasks then block until all of them are terminated before the caller code can proceed.

In addition to specifying parallelism, under certain circumstances, the parallel skeleton is able to detect and protect transparently the data that is shared between parallel tasks to avoid race condition and to relieve the programmer from managing shared data "by hand". This feature is detailed in the next section.

```
1  // functions
2  int f1(char * data);
3  int f2(void);
4  int f3(void);
5
6  void main() {
7   /* task definition */
8   task t1(f1, data_1), t2(f2), t3(f3);
9   /* task graph construction */
10  task_group * program;
11  program = parallel(&t1, &t2, &t3);
12  /* task graph execution */
13  program->run();
14 }
```

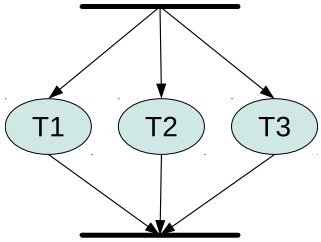


Figure 5.3: The "parallel" construct specifies the simultaneous execution of a set of tasks.

5.2.3 Task Graph : Skeleton Composition

By design, the *parallel* and *sequential* skeletons are composable and can be nested within each others. This allows the programmer to specify parallel and sequential execution of a program as an hierarchical task graph. The nesting depth is not limited and enables parallelism specification at all levels of granularity. Figure 5.4 shows and example of an hierarchical task graph and the corresponding XPU code. After defining tasks (line 9), the task graph is created by alternating the use of the "parallel" and the "sequential" skeleton as shown in line 12. Each skeleton corresponds to parallel or sequential sections. The graph can be executed simply by invoking the "run" method of the root skeleton (line 19).

The resulting execution grantee the respect of the specified parallelism in the task graph but does not guarantee the execution order of parallel tasks. The platform processors are used to exploit the available parallelism.

5.3 Automatic Shared Data Detection and Protection

In the Chapter 5, we presented the Transparent Task Data Dependency detection (TTDDD) feature of the XPU task. This functionality allows the XPU runtime to extract the dependencies of each task and more precisely the accesses ("*read-only*" or "*read-write*") of the task to the data. TTDDD can be useful to improve execution efficiency through cache-aware scheduling or to improve programmer productivity by automating shared memory management.

In the particular context of the implementation of the "*parallel*" skeleton, we ex-

```
// functions
int f1(char * data);
int f2(void);
int f3(void);
// ...

void main() {
 /* task definition */
 task t1(f1, data_1), t2(f2), t3(f3); // ...
 /* task graph construction */
 task_group * program;
 program = parallel(sequential(&t1,
                               parallel(&t3,&t4),
                               &t6),
                    sequential(&t2,
                               &t5,
                               &t7));
 /* task graph execution */
 program->run();
}
```

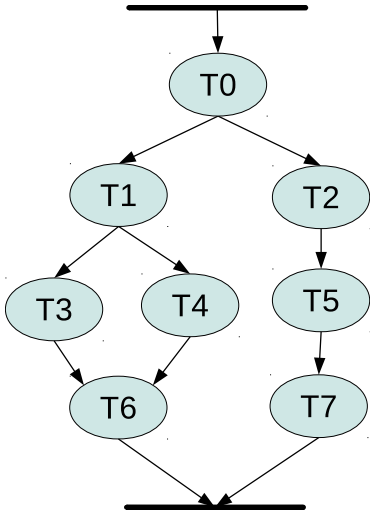


Figure 5.4: Task graph with both parallel and sequential regions can be specified by combining "parallel" and "sequential".

ploited the TTDDD feature of the XPU task to automate the detection and the protection os shared data against potential "*race condition*". The *race condition* phenomenon occurs when concurrent tasks accesses simultaneously to shared data and that one or more of these tasks modify the value of the shared data. This phenomenon is know as "*race condition*" and can compromise the execution integrity of a program by corrupting shared data. Moreover, *race condition* can be difficult to detect when debugging an application due to the non deterministic execution of parallel tasks. Traditionally, programmer is responsible of handling shared data management and "*race condition*" avoidance: the programmer detects potential "race condition" then using mutual exclusion mechanisms that guarantee atomic accesses to data at a time and eliminate the risk of "*race condition*". The process of detecting race condition and eliminating them constitute one of the burdens of parallel programmers that limits their productivity. As we will see in this section, this burden can be avoided by automating shared memory management under certain circumstances. We implemented a mechanism named Automatic Shared Data Detection and Protection (ASDDP) in XPU. It consists in the detection of shared data and potential conflictual concurrent accesses to the data then the protection of that data using mutual exclusion mechanism when required.

5.4 Shared Data Detection

Assuming we have a "parallel" construct that specifies parallelism between N tasks, shared data can be detected by following the next steps:

1. Interrogate each of the N tasks about its data accesses. The result is a set of pointers to data with "*read-only*" or "*read-write*" access flag.

2. Compare the returned pointers of all tasks with each others to extract the shared data by applying the following rule: if two or more tasks accesses a data, the data is considered as shared.

3. Distinguish the two types of shared data:

   The first type of shared data is accessed by concurrent tasks exclusively in "*read only*" mode. Consequently no mutual exclusion or protection is required.

   The second type of shared data is the one accessed in "*write*" mode by at least one task of the concurrent tasks. Such shared data introduce potential "*race condition*" and requires protection with mutual exclusion mechanism.

5.4.1 Simple Configuration

Figure 5.5 shows two simple configurations where data is accessed concurrently by parallel tasks. In the first configuration, the shared data "*Data 3*" is accessed in "*read only*" mode and thus it does not introduce any risk of "*race condition*", however, in the second configuration, the shared data is accessed in "write" mode simultaneously by two concurrent tasks introducing a potential "race condition". In the later case, shared data is protected using mutual exclusion or critical section mechanism as we will see in the next paragraphs.

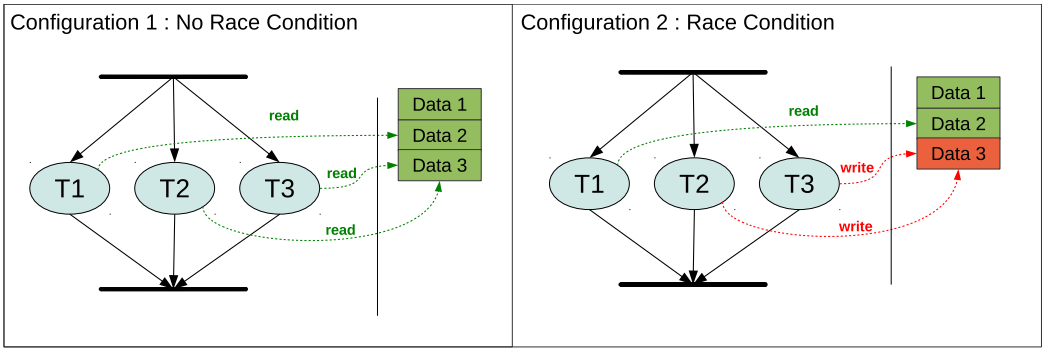


Figure 5.5: Race condition does not occur in the first configuration since the shared data (data 3) is accessed concurrently in read only mode. However, race condition can happen in the second configuration where concurrent writes are operated on the shared data.

5.4.2 Generalization to Hierarchical Task Graph

Similarly to the XPU "*task*", "*parallel*" and "*sequential*" constructs are different implementations of the common "*task_group*" interface. Therefore, as XPU "*task_groups*", they are able to return their dependencies similarly to any "task" simply by returning all the dependencies of their encapsulated tasks. Consequently, if we replace the task T1, T2 or T3 in the previous example by any other *task_group* such as "*parallel*" or "*sequential*", the shared data detection mechanism can still work correctly. Moreover, if we replace the later constructs by a composite one in which many different constructs are nested hierarchically, the mechanism can still work recursively at the different levels of the hierarchy by applying the same algorithm each time a "*parallel*" construct is encountered. This will guarantee the detection of shared data at any granularity level of an hierarchical task graph.

Figure 5.6 illustrate an example of an hierarchical task graph where "sequential" and "parallel" construct are composed. The example contains two parallel region at two different granularity levels. The detection algorithm will be applied in the first parallel region (red), then in the second parallel region (blue). When processing the first parallel region, the data accessed by the tasks T2, T5 and T7 will be compared with the data used by T1,T3,T4 and T6 to extract their intersection, e.g. shared data. In the next stage, the same algorithm is applied to the second parallel region: the data accessed by T3 will be compared with the one accessed by T4.

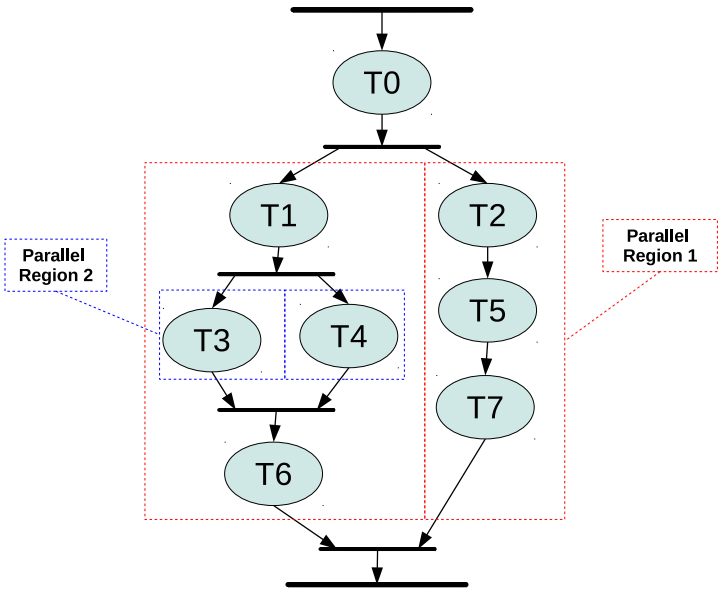


Figure 5.6: The shared data detection algorithm can be applied to all hierarchy levels of and hierarchical task graph.

5.5 Shared Data Protection with Critical Sections

5.5.1 Mutual Exclusion with XPU Lockables

Once shared data detected, in order to protect it against unsafe concurrent accesses, the run-time guarantees exclusive or atomic access of the tasks to that shared data. Each data is identified by its memory address (pointer), when the data is flagged as shared and accessed in *write* mode, the run-time associates a unique "*lockable*" to that data if no "*lockable*" is already associated to it.

XPU uses a central "*lockable*" factory that grantee that a unique lock is associated to each data. The "*lockable*" object is an abstraction of mutual exclusion mechanisms such as POSIX "*mutex*" or "*spinlocks*" that displays a common interface. Figure 5.7 shows the design of the XPU *Lockable* interface and its various implementations. It shows also the central *LockableFactory* that delivers unique "*lockable*" for each data.

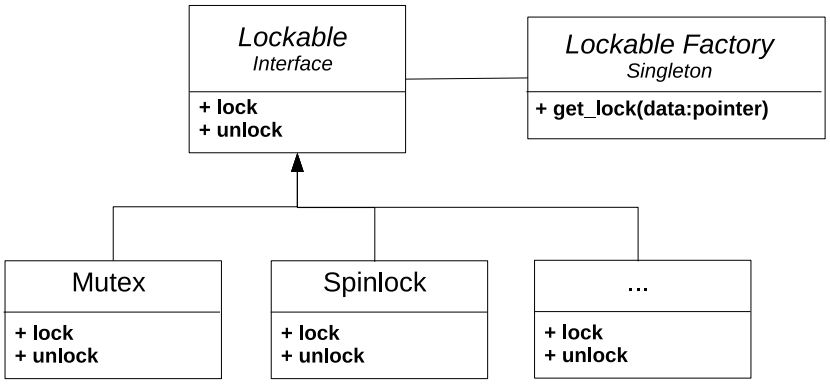


Figure 5.7: Design of the Lockable interface and its implementations. The Lockable Factory guarantee that a unique Lockable is associated to each data identified by its memory address.

Similarly to a POSIX "*mutex*" or a "*spinlock*", when a task A attempts to lock a "lockable" L which is already locked by another task B, the task A is blocked until the task B unlock L.

5.5.2 Critical Section

The XPU runtime protects data by executing the tasks accessing concurrently to the shared data inside critical sections relatively to that shared data. Running a task inside a critical section guarantees the atomic or exclusive access to the shared data and eliminates the risk of "*race condition*".

The conventional task execution corresponds to calling the encapsulated code(function or object method...). In order to transform the task into a critical section relatively to

a shared data, the task locks the shared data before executing its code, once the code execution terminated, it unlocks the data. Thus, once entred the critical section, the task has an exclusive access to the shared data. Figure 5.8 illustrates the use of critical sections to protect shared data against concurrent accesses within an hierarchical task graph. We note that a task can lock multiple shared data when needed. However, potential deadlocks can occur if the "locking" of the different shared data is not ordered properly. The XPU runtime system can resolve potential deadlocks simply by ordering the locking of the various shared data.

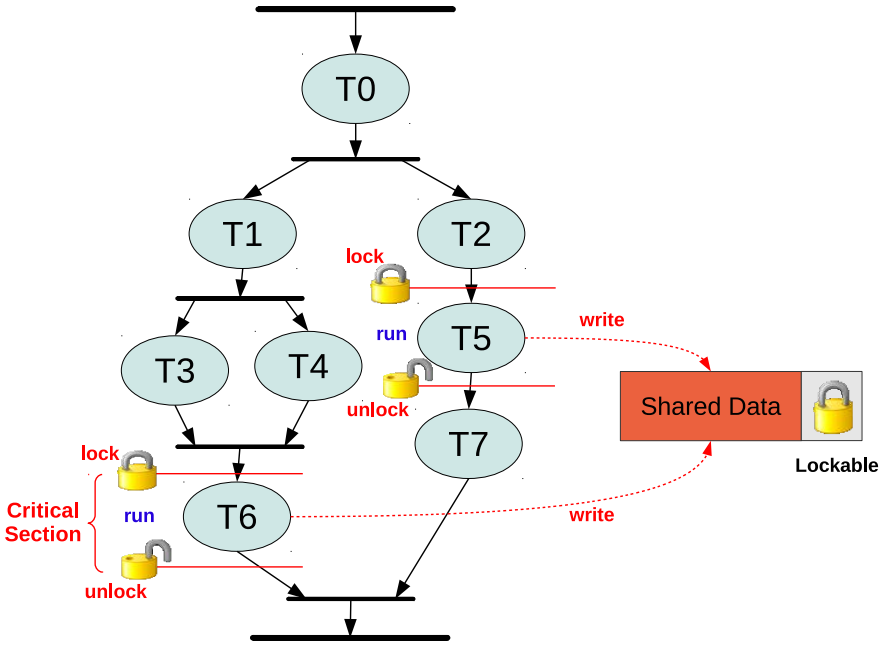


Figure 5.8: Task accessing concurrently to shared data are executed inside critical sections: the shared resource is locked before the task execution then unlocked when execution terminate.

5.6 Programmability Comparison with Intel TBB

In [KLLDS12], we tried to evaluate XPU programmability in comparison with TBB. A traditional approach for quantifying programmablity or required programming effort is to compare the parallel program to its sequential version in terms of the number of lines of code [TTTn+09]. We consider a simple sequential application in which we call successively 7 functions, this application can be parallelized as shown in our first example in 5.9. We associate a task to each function and we note that "Data_4" is shared between the parallel tasks T4 and T5.

We tried to express task parallelism as specified in the task graph using XPU and

TBB then we counted the required parallelism-related extra-code lines and the reused sequential code lines. We used "CLOC" [Sol] to count lines of code in each version, we removed all blank lines and comments, then we used the classic "*comm*" string comparison tool, available in most UNIX systems, to compare the parallel and sequential code. The comparison allows us to determine the number of lines of reused sequential legacy code and required extra-code in the parallel version.

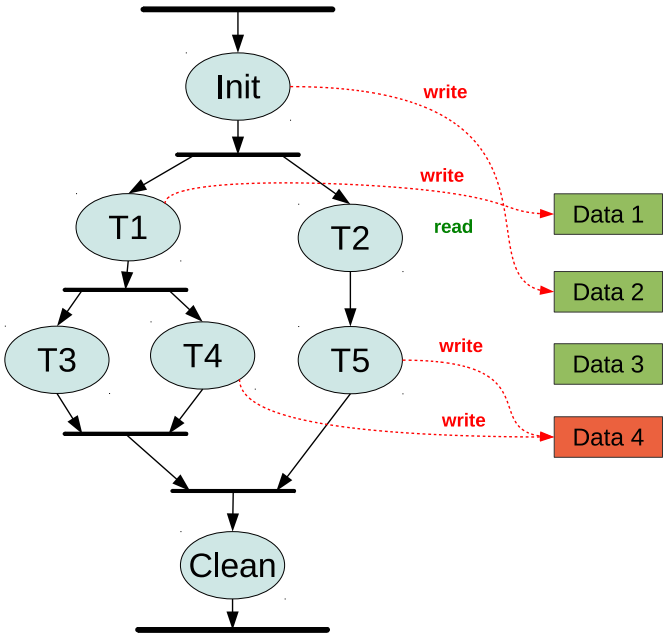


Figure 5.9: Example of a task graph that specifies parallelism and data dependencies.

In the TBB version we define 6 task classes: 3 tasks to encapsulate Function 1,2 and 3, one intermediate task to hold these 3 tasks, another intermediate task to call successively Function 2 and 5 and finally we define the root task that spawns these intermediate tasks. We use TBB task parallelism primitives: allocating, spawning and waiting for root and child tasks. A TBB mutex is used to protect shared data "Data_4" from race condition [Phe08].

XPU version requires a single line to define a task for each function in the parallel section and another 2 lines to build the task graph and execute it. The sequential code of functions does not require any modification. The shared data are detected and protected automatically by the run-time system. As shown in Figure 5.10, while XPU version reuses 80 lines out of the 86 lines of sequential legacy code and requires only 10 extra-lines for parallelization, the TBB version reuses only 42 lines of legacy code and requires more than 140 lines of parallelism-related extra-code.

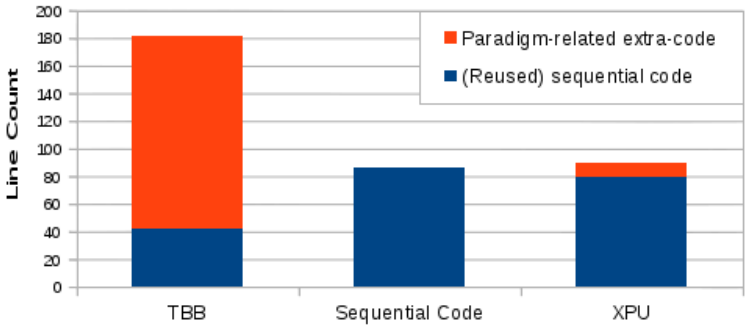


Figure 5.10: Comparison between XPU and TBB in term of required programming effort: required extra-code and reuse of sequential code.

5.7 Execution Infrastructure

When parallelizing an application, the programmer selects the appropriate skeleton to specify parallelism and uses the XPU API to express it. Once parallelism specified, the XPU runtime is responsible of the efficient execution of the tasks according to the specified parallelism. The XPU execution infrastructure is composed mainly of a persistent thread pool or a "*generic worker*" pool. The "*generic worker*" is called generic because it can execute different types of tasks coming from different skeletons. In order to execute tasks, the tasks are submitted to the workers through the worker's "*task queue*". The "*task queue*" is a First-In-First-Out (FIFO) queue that support one-to-one communication pattern and thus can be efficiently implemented as a lock-free queue with very low communication overhead. The current implementation of XPU does not use lock-free queue but still provide decent performances in comparison with state of the art programming models such as GNU implementation of OpenMP or TBB. The use of lock-free queue in future versions can improve the performances of the task queue.

The execution infrastructure is based on a *master-slave* architecture where the master thread which is responsible of executing the skeleton submits the tasks to slaves that are responsible of their execution. When executing skeletons, the tasks are submitted in their specified order sequentially to the same worker or simultaneously (parallel) to concurrent workers depending on the used skeleton. The task is submitted as an abstract "*work*" which provides a common interface that allows the "*generic worker*" (slave) to execute it and the master thread to block waiting for the "*work*" to terminate.

The "*generic worker*" are mapped or "pined" to the available processors or cores. The XPU runtime can assign the execution of a task to a specific core by submitting that task to the worker running on the target processor or core. This design offers to us a great flexibility to prototype different schedulers with various scheduling strategies. For instance, if we want to perform efficient processor cache use through improving

data locality, we can schedule tasks that are using the same data on the same core to promote data reuse and reduce communication overhead and "cache pollution".

5.7.1 Sequential Execution

In order to illustrate the execution of skeleton, we consider the example of the "*sequential*" skeleton that execute serially a set of tasks. As illustrated in Figure 5.11, the programmer instantiates the skeleton using the "*sequential*" API of XPU. When the "*run*" method of the skeleton is invoked, the runtime execute it: the tasks are submitted in their sequential order to one "*worker*" through its "*task queue*", the "*worker*" pops the tasks from its FIFO and executes them one after the other. After submitting the tasks, the master thread blocks and waits until all the tasks are executed. When tasks terminate, the master thread is unblocked and continue its execution.

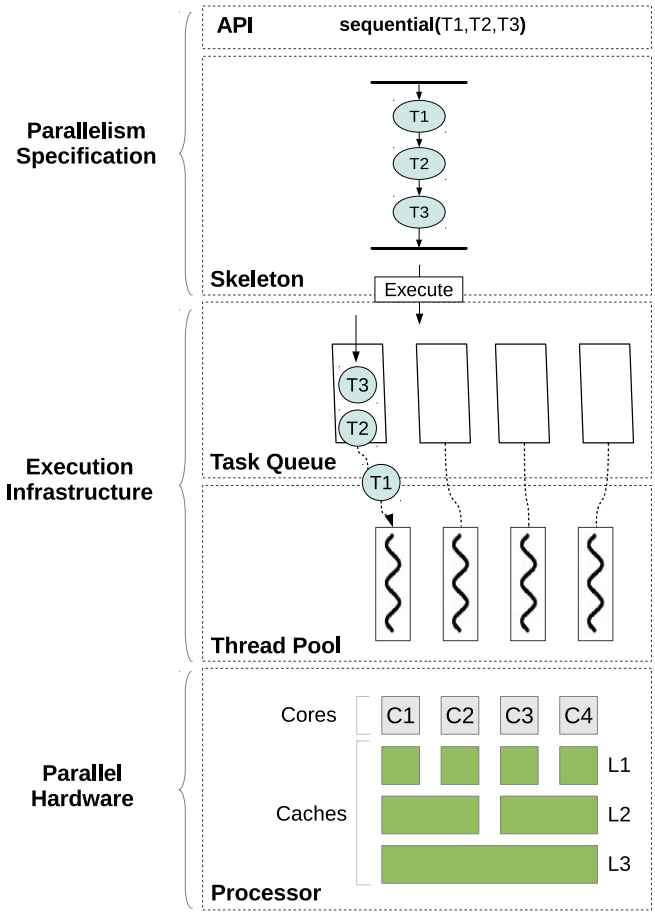


Figure 5.11: Execution details of the "*sequential*" skeleton.

5.7.2 Parallel Execution

The execution of the "*parallel*" skeleton is similar to that of the "*sequential*" construct, however, contrary to the "*sequential*" skeleton execution where tasks are submitted in their sequential order to the same worker, the tasks of the "*parallel*" skeleton are submitted to different workers running concurrently on different processors. After scheduling tasks for execution, the skeleton wait for them to terminate. Once all tasks finish their execution, the master thread is unblocked and continue its execution.

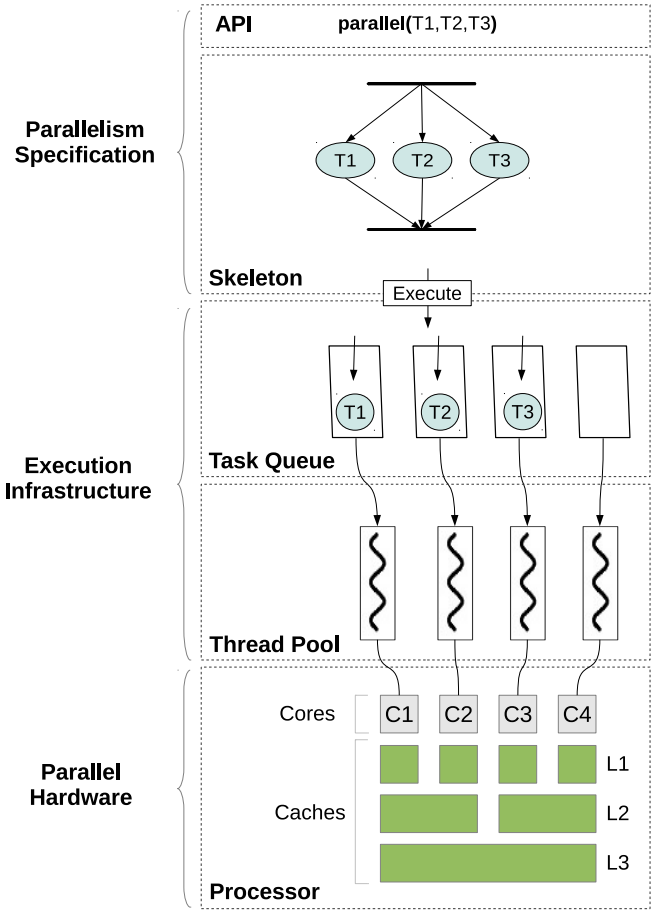


Figure 5.12: Execution details of the "*parallel*" skeleton.

5.8 Conclusion

Task parallelism using the hierarchical task graph offers a flexible parallelization methodology which allows extracting parallelism from sequential programs even when they expose a little amount parallelism. Task parallelism can be a great parallelism multiplier

provided that the program's tasks have reasonable granularities. However task parallelism using static task graphs exposes some limitations such as its weak scalability. Task parallelism can be coupled with data parallelism to extract more parallelism and scalability. Generally, data parallelism offers much more parallelism than task parallelism and displays often an excellent scalability provided that the application do not exposes dependencies between its data items so they can be processed concurrently. In the next chapter, we show how XPU can ease data parallelism expression and we present the *parallel for* skeleton and the vectorization capabilities of XPU.

6

Data Parallelism

In this chapter we show how XPU can facilitate the design of applications where data parallelism is crucial. We introduce the XPU *parallel_for* pattern which allows loop parallelization and we describe the details of its implementation. In a second part, we present the XPU CHATS algorithm [1] which allows the parallel loop construct to perform efficient cache-aware scheduling. In a third part, we focus on vectorization capabilities of XPU. Finally, we show how we can use most of the aforementioned techniques to parallelize two popular applications from the PARSEC Benchmark Suite [BKSL08] which are the Blackscholes and the Fluid Simulation applications, then we compare them to other parallel implementations which use several state-of-the-art parallel programming models such as Intel TBB, OpenMP or PThreads.

Data parallelism refers to scenarios in which the same operation is performed concurrently on elements of a data container [Micb]. Data parallelism can be specified at different levels of granularity and can be implemented at thread level (Thread Level Parallelism (TLP)) or at the instruction level (Instruction Level Parallelism (ILP)).

In data parallel operations, data are partitioned so multiple threads can operate on different data partitions concurrently. Figure 6.1 shows an example of data parallelism implementation in an image processing application: contrary to sequential image processing where image pixels are processed one after the other, TLP is implemented by partitioning the image into four partitions which are processed simultaneously by four threads running on the four processor cores. Moreover, in order to increase throughput, ILP can be used to process several pixels simultaneously within each partition. Depending on the available vectorization technology and the used data type to represent a pixel. Thanks to vectorized instruction set, up to 64 pixels can be processed simultaneously.

[1] **Publication**: Nader Khammassi and Jean-Christophe Le Lann, "***Design and Implementation Of A Cache Hierarchy-Aware Task Scheduling For Parallel Loops On Multicore Architectures***", Third International Conference on Parallel, Distributed Computing Technologies and Applications, PDCTA 2014, Sydney, Australia

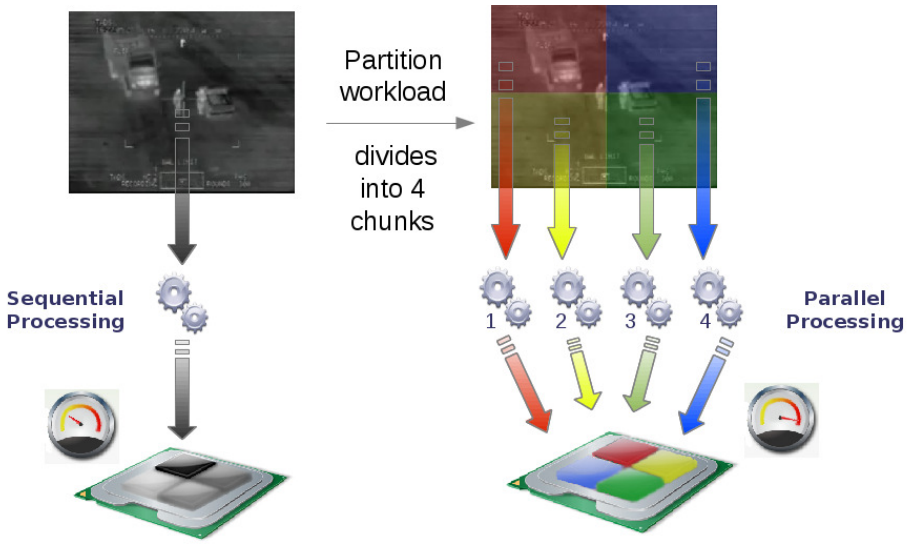


Figure 6.1: Example of data parallelism implementation in an image processing application

XPU enables the programmer to express data parallelism at thread level (TLP) through parallelized *for* loop, at instruction level (ILP) through a set of vectorized data types (SIMD) and at both of them through the XPU "parallel vector" interface.

6.1 Parallel Loop

Parallel loop is one of the most used execution pattern since it can act as a great parallelism multiplier in data parallel applications while achieving an excellent scalability. It targets often "for loops" when processing large number of data items. "Parallel for" execution pattern is implemented in most of the popular parallel programming models such as OpenMP [DM98], Intel TBB [Phe08] or Cilk++ [Lei09]...

Parallelizing "*for loop*" is performed mainly in two steps: partitioning data element set to split the main workload into a set of small workloads then scheduling these workloads on the available processing units. Figure 6.2 shows a basic partitioning scheme where the main data array is divided into four chunks. These chunks can then be processed simultaneously on four different processing units. The main array size may not be divisible by the number of processing units. In this case, quasi-fair partitioning algorithms can be used to provide the fairest possible partitioning to avoid potential load unbalance between the processing units. Data partitioning and partition count do not depend only on the available processing units count: the used scheduling strategy plays a central role in the choice of the workload granularity and partition size and count.

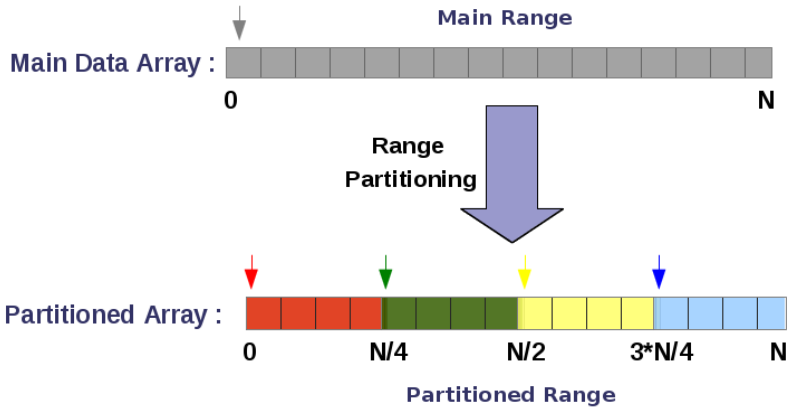


Figure 6.2: Basic partitioning of a data array.

In the next paragraphs, the term "*range*" is used to indicate a data set defined by its start index and its end index. "*range*" is often used to refer to the main data set. "*sub-range*", "*partition*" or "*chunk*" can be used interchangeably to refer to a part of a main data set.

In XPU, the "*parallel_for*" pattern is a task group implementation specifying the parallelization of *for* loop. When defining a parallel loop (see Listing 6.1), its main range will be partitioned transparently into several sub-ranges using a pseudo-fair partitioning algorithm depending on the available processing units. Partitioning is performed in an offline fashion at the pattern construction stage, i.e. before its execution.

Offline partitioning contributes to minimize execution overheads and allowing us to implement various data partitioning schemes without caring about algorithmic complexity: for instance, we experimented several workload distribution techniques and data partitioning algorithms without modifying the "*parallel_for*" front-end due to its high abstraction of implementation details.

6.1.1 Parallel For Loop Programming Interface

```
1  // main task of the parallel loop
2  int process(int from, int to, int step, image* images)
3  {
4      for (int i=from; i<to; i+=step)
5         images[i]->filter();
6  }
7
8  void main()
9  {
10   image * images = new image[N];
```

```
11    task process_t(process, 0,0,0, images);
12    task_group * pf;
13    pf = new parallel_for(0, image_count, 1, &process_t);
14    pf->run();
15  }
```

Listing 6.1: Use of the XPU parallel for loop.

6.1.2 Cache-Hierarchy Aware Scheduling (CHATS)

6.1.2.1 Chip Multicore Processor and Cache Hierarchy

Chip Multiprocessor (CMP) architectures are becoming widely available on many scales: from personal computers to embedded systems to high performance supercomputers...[KB09] [KAG+09]. CMP cores count is growing continuously and their cache topologies are becoming increasingly hierarchical and deeper. Cache-aware scheduling has become a great design challenge in parallel programming for recent multicore architectures.

Chip Multiprocessor (CMP) may exhibit different cache topologies with varying numbers of hierarchical shared and private caches at different levels. An effective task scheduling policy must take into account cache sharing not only at the SMT (Simultaneous Multi-Threading), CMP and SMP (Symetric Multi-Processor) levels but also at the different cache levels of a same chip.

6.1.2.2 Task Scheduling

Task scheduling is critical for execution efficiency especially in the case of parallel loops which are often a great performance multiplier. An efficient cache-aware scheduling policy for recent CMP should take into consideration three major parameters: spatial and temporal data locality in caches, communication and load-balancing. Hierarchical cache topology determines communication latencies between cores at the different levels of cache and thus, has a direct impact or these three critical scheduling factors.

In the following paragraphs, we describe a cache hierarchy-aware task scheduling (CHATS) policy which target to provide efficient hierarchical cache utilization without neglecting load-balancing in parallel loop implementation.

CHATS consider spatial and temporal data locality (data reuse and communication) and load-balancing as the most critical parameters for delivering high performance and execution efficiency. We implemented this scheduling algorithm in the parallel loop construct "*parallel_for*" of the XPU framework and we compared it to parallel programming frameworks using different scheduling techniques. We used both synthetic workloads and real application from the PARSEC and Intel RMS Benchmarks [BKSL08].

6.1.2.3 Related Works

Traditional scheduling techniques such as dynamic scheduling [ACD+09] or task-stealing [CKK+08] [FLR98] [Lei09] make different trade-offs between data locality and load-balancing but does not take into consideration cache hierarchy and communication latencies. Some prior works target to design cache-aware scheduling policies which target to improve cache-utilization by focusing on one or more cache-related considerations:

- **Processor-Cache Affinity Scheduling** [SL93] focused on temporal data locality and data reuse between threads.
- **Thread Clustering Scheduler** [TAS07] detect sharing patterns between threads online using monitoring techniques and attempts to reduce cache-coherence overhead by clustering threads sharing same data onto close cores.
- **CAB** [CGH13] aims to improve task stealing on hybrid SMP-CMP by reducing memory footprint and cache misses, It focus mainly on data sharing at the SMP level and try to reduce inter-socket communication.
- **Constructive Cache Sharing** [CGK+07] aims to reduce the memory footprint through exploiting potential overlap of shared data among co-scheduled threads.
- **CATS** [YLY10] target to improve cache performance by considering data reuse, memory footprint and coherency misses. None of these prior works take into consideration the cache hierarchy of CMPs.

6.1.2.4 Unified CMP Architecture Model

Multicore processor employs a cache structure to simulate a fast common memory. This cache structure may display different cache sharing degrees at different levels. It is mainly composed of hierarchical private and shared caches.

Figure 6.3, 6.4 and 6.5 shows a set of CMP architectures from different vendors with varying cache configurations. For example, while the Intel Nehalem architecture associates a private L1 cache for each core, a private L2 cache and a shared L3 cache between all cores, Intel Dunnington architecture is uniformly hierarchical: a private L1 cache is associated to each core, an L2 cache is shared between each 2 cores and finally an common L3 cache is shared between all cores. The Sun UltraSPARC T2 Processor uses a private L1 cache for each core and a shared L2 caches between all cores.

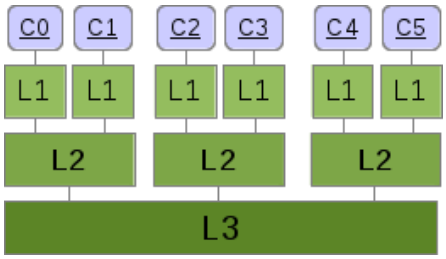


Figure 6.3: Simplified Architecture Overview of the Intel Dunnington Processor

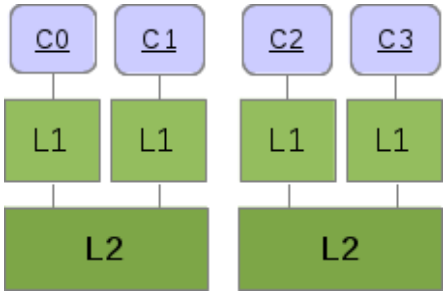


Figure 6.4: Simplified Architecture Overview of the Intel Hapertown Processor

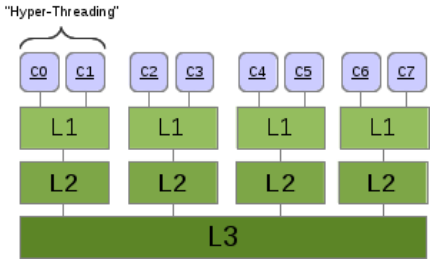


Figure 6.5: Simplified Architecture Overview of the Intel Nehalem Processor

Cache level count and cache sharing degrees at each level are key information for our scheduling policy. The variation of sharing degree at different levels force programmer to make explicit and architecture-specific program optimization in order to get efficient execution.

In order to be provide an efficient execution on various possible underlying architectures with different cache topology, a cache-aware scheduling algorithm should be dynamically adaptable to the target architecture. Consequently, such scheduling algorithm should have a detailed description of the cache topology of the underlying CMP.

This description can be established through dynamic exploration of the target platform at the initialization of the run-time system. Modern operating systems provide means to obtains cache hierarchy details at high level either through system files such as in the Linux OS or through native API such as Windows [Mica].

Variation of the cache level count and cache sharing degrees raise the need to unify them under a single abstract description. The Unified Multicore Architecture Model (UMAM) [SZ08] that can be used to provide a unified description for different CMP architectures.

Memory hierarchy including cache levels and main shared memory can be described using UMAM as shown in TAB. 6.1 which gives an example of three different platforms. The two first columns gives the cache-levels and cores count, the next columns gives the count of cores sharing L_i caches or M_i memory ($i \in [1..n]$; n : memory hierarchy levels count).

Architectures	Parameters					
	Cache Levels	Core	L1	L2	L3	Memory
Intel Nehalem Core i7 Q920M	4	8	2	2	8	8
SMP (2 x Intel Nehalem Xeon E5620)	4	16	2	2	8	16
Intel Dunnington Xeon X7460	4	6	1	2	6	6

Table 6.1: UNAM

6.1.2.5 Cache Hierarchy-Aware Task Scheduling (CHATS)

The design and implementation of CHATS rely a several basic building blocks which allows partitioning of the main work of a parallel loop into a set a little works which can be executed concurrently by several threads. We start by defining the components of our run-time system.

A The Run-Time System

The run-time system is based on a worker (thread) pool able to execute tasks. Each worker have a FIFO (First-In-First-Out) work (task) queue. A scheduler can submit tasks to the workers through this work-queue. A worker remains idle until a task is pushed into its work queue so it wake up and execute its task. Submitting a task follows a one-to-one communication scheme between the main thread holding the scheduler and each worker to reduce communication overhead. Figure 6.6 gives and overview of the run-time system.

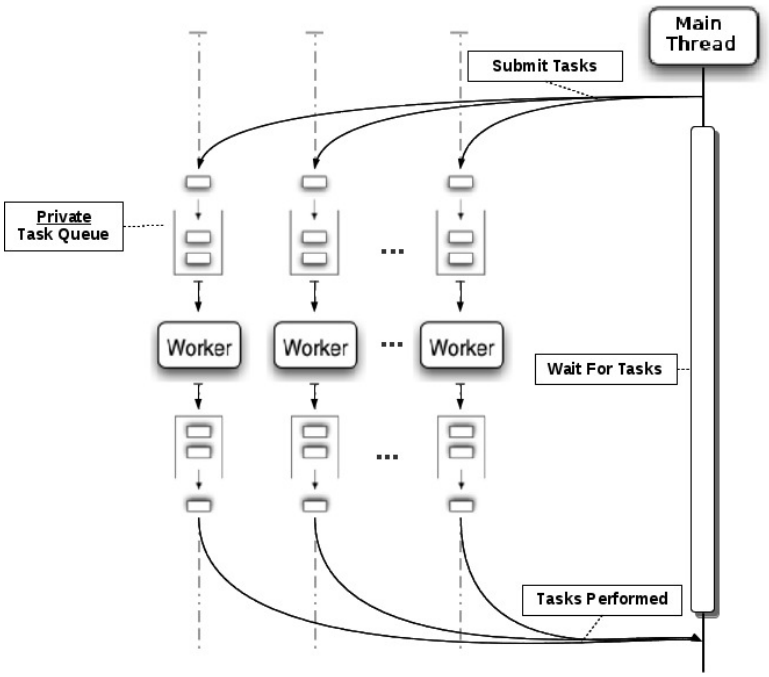


Figure 6.6: Worker Pool-Based Run-Time System With Private Work-Queue

B Work Unit

A work unit is a task which should be executed on a range of iteration then a range of shared iterations. In XPU "*parallel_for*" loop, a work unit is composed of:

- **Range** : Specifies a range of iterations to process (min, max, progression step)
- **Shared Range** : As "*Range*", it specify a range of iteration, however, it allows "stealing" of iterations by concurrent threads.
- **Task** : The code which will process each iteration of a given range and/or shared range(s).

C Data Partitioning

Data partitioning is a primordial step in parallelization of a loop. In our case we use basic a quasi-fair partitioning algorithm which decompose a "*Range*" into N "*Range*" and M "*Shared Range*". The algorithm ensure that the generated partitions are quasi-equals.

D Parallel For Loop : Data Partitioning and Cache Topology

Let's consider a "for loop" **F** defined by : i = [0..n] by 1, where F corresponds to a "*Range*" which can be partitioned into N "*Range*" and M "Shared Range". Determining

M and N depends directly on the underlying architecture:

- N = Workers count Cores count
- M = Number of shared caches at all levels (consecutive cache levels shared by the same cores are considered as one)
- P = N+M : Total partitions count

Let's consider a *Nehalem Intel Core i7 Q720* with 8 Hardware Threads and 4 Physical Cores (Figure 6.7). Data partitioning is described in Figure 5, so P is equal to 13 in this case. Green ranges are private "*Range*" so a worker doesn't share them with other workers. Orange box correspond to "Shared Ranges" which are shared among two co-scheduled SMT workers (threads) sharing L1/L2 caches. Finally, red box is a "Shared Range" from which all workers can steal iterations, it corresponds to the L3 cache.

Figure 6.8 shows the data partitioning scheme for an SMP platform containing two Intel Nehalem Processors having eight hardware threads (4 Physical cores with Hyper-Threading enabled).

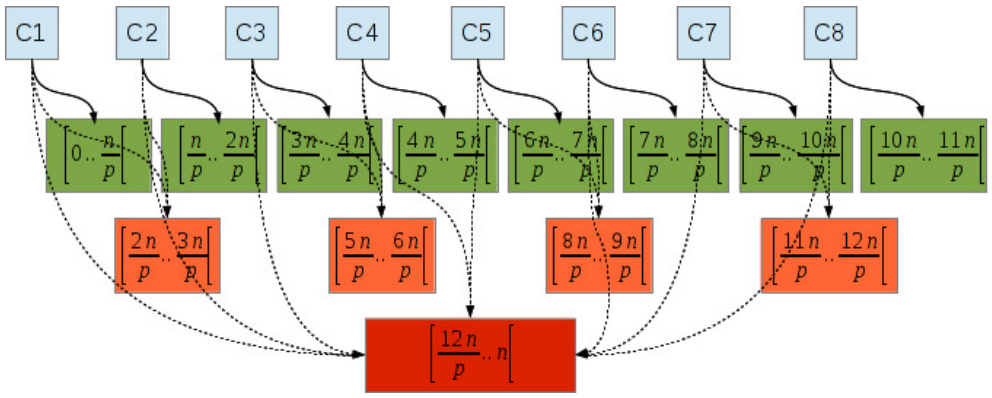


Figure 6.7: CHATS Data Partitioning on an 8 Hardware Threads Intel Nehalem Processor

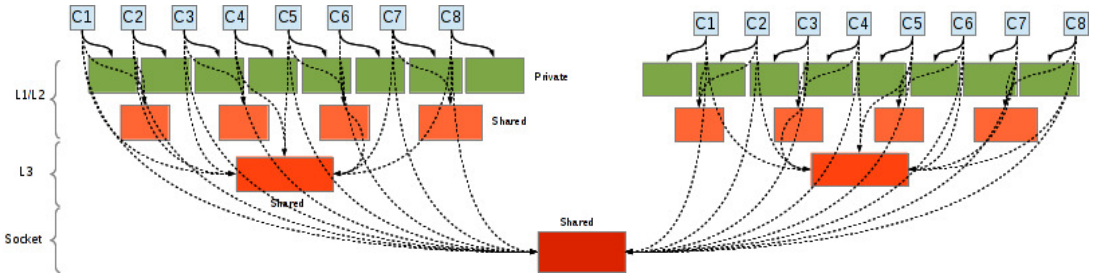


Figure 6.8: CHATS Data Partitioning on an on hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz)

E Workload Scheduling

Once data partitioned into "*Ranges*" and "*Shared Ranges*", we can submit works to our workers running on the different cores. Submitted work will specify a Task, a Range and one ore more Shared Ranges. If we take the partitioning pattern of the Figure 5. "Worker 0" running on "Core 0" will get a work containing:

- One "*Range*" : [0 .. n/p[
- Tow "Shared Ranges" : [2n/p .. 3n/p[and [12n/p .. n[

Analogously, the other workers will gets their three ranges of iterations.

F Execution Semantic

"Worker 0" will execute the task code on each iteration of its private range without any communication with the other threads. Once finished, it will try to steal iterations from the shared ranges if available. Iteration stealing requires communication (locking) between threads working on the same shared range. This communication overhead is reduced by the fact that threads communicates through shared caches. So, the communication introduced buy concurrent accesses to "Shared Range" [12n/p .. n[is more costly than the one introduced by concurrent accesses to [2n/p .. 3n/p[. However, we note that low level caches-associated "Shared Range" are fewer than those associated to high level caches (1 associated to L3 and 4 associated to L1/L2).

We outline that "Shared Ranges" aims to provide good load-balancing at the lower possible cost in term of communication overhead : when worker finish their work on private works, they does not remain idle, instead, they steal works from shared ranges or more precisely the "closed" shared range to their high level caches.

6.1.3 Performance Evaluation

We compare our CHATS implementation to several popular programming models implementing static scheduling, dynamic scheduling and task stealing which we present briefly.

6.1.3.1 Common Scheduling Policies

A Static scheduling

Static scheduling is the most straightforward scheduling technique: data is statically partitioned into N equal or pseudo-equal chunks, these chunks are then processed respectively by N parallel threads. This scheduling scheme avoid communication between threads, offer good data reuse when the parallel loop is executed several time. However, this method may result in load unbalancing, especially in the case of heavy workload, since faster threads remains idle, waiting for other threads until finishing their work.

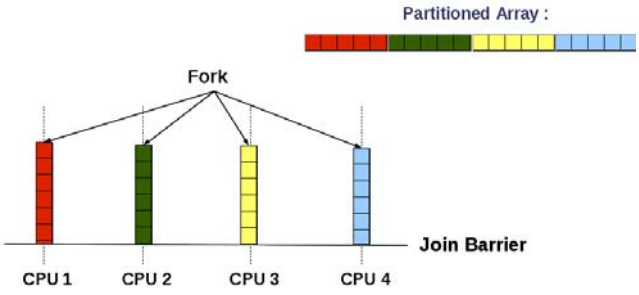


Figure 6.9: Static Scheduling In Theory

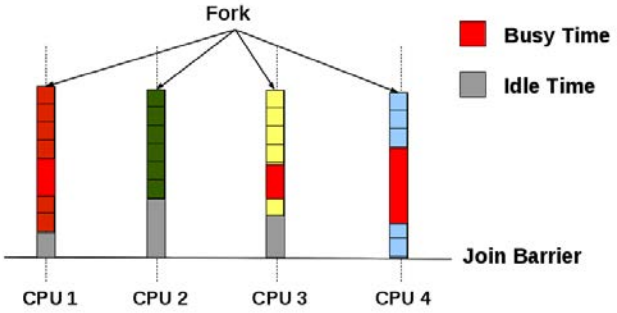


Figure 6.10: Static Scheduling In Practice

B Dynamic scheduling

Dynamic scheduling provide better load balancing since threads does not remains idle as long as chunks are available in the common work queue. Unfortunately, while improving workload distribution, this technique may introduces a costly communication between threads accessing concurrently to the common work queue (Many-To-Many Communication). This may results into ineffective uses of processor caches. Also, this technique provide poor data reuse since a same chunks may be processed by different threads on different cores when the parallel loop is executed multiple time. Bad data reuse may amplify consequently cache-miss rate.

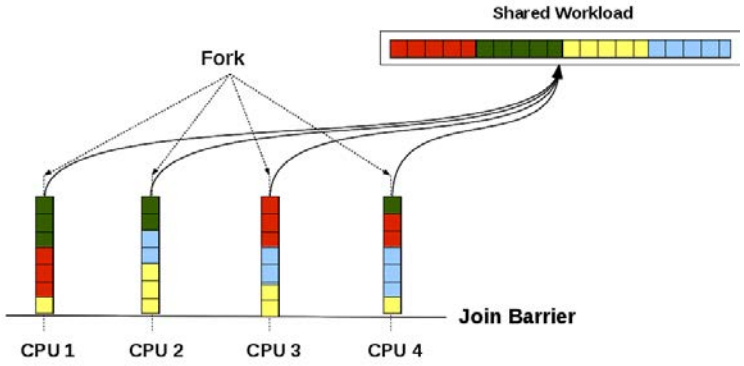


Figure 6.11: Dynamic Scheduling In Theory

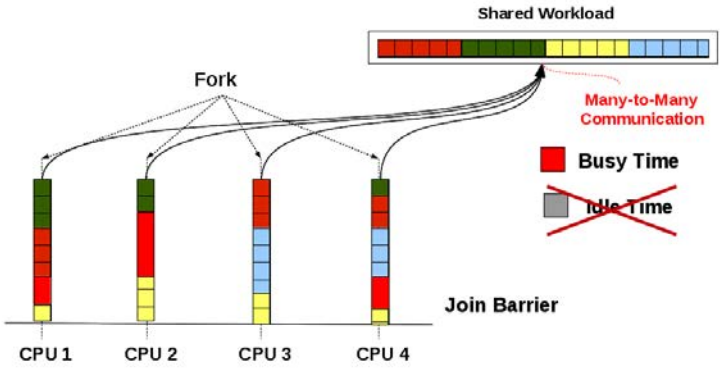


Figure 6.12: Dynamic Scheduling In Practice

C Task Stealing

Task-stealing is a popular scheduling algorithm which is introduced in Cilk [FLR98]. Task-stealing attempts to combines advantages of the two previous scheduling policies

by making another trade-off between efficient cache utilization and load-balancing, In task stealing, each thread (worker) has a task pool in which its tasks are stored. Whenever a worker finishes its current task, the worker try to get another task from its task pool. If there's no more works to perform (its task pool is empty), the worker select randomly a "victim" worker and try to steal a work from its task pool. If succeeded, it execute the stolen task, otherwise, it try to steal a task from another randomly-chosen worker [CGH13].

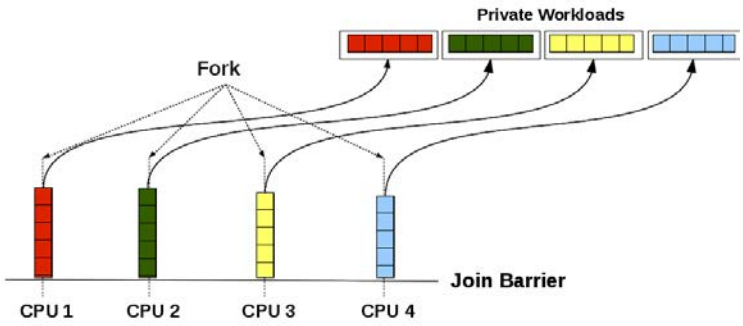


Figure 6.13: Task Stealing Scheduling In Theory

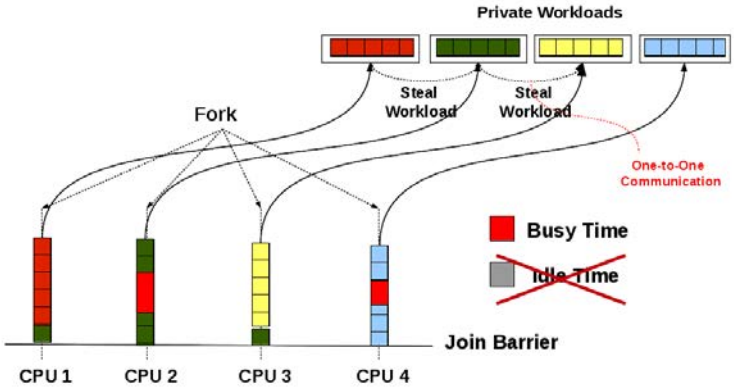


Figure 6.14: Task Stealing Scheduling In Practice

Task-stealing performs good load-balancing since no thread (worker) remains idle as long as there is available "works", i.e. ,available tasks in the task pools of workers. However, task-stealing may introduce significant communication overhead since "victim" threads are chosen randomly without considering cache-hierarchy or communication latencies.

Deep cache hierarchies introduce non-uniform communications between cores, con-

sequently, the choice of the "victim" thread becomes critical for performance: stealing a task from a "close" thread (sharing high-level cache with the stealer) is much cheaper than stealing a task from a "far" thread (running on a core which does not share any cache with the stealer).

6.1.3.2 Experiment : Parallel For On Synthetic Workloads

In order to evaluate the performances of our approach we designed an experiment which aims to evaluate cache utilization efficiency and global performances of a configurable target application. We generate a synthetic work load witch allow us to control unit workload and global workload and simulate data reuse. Thus, in order to achieve efficient execution, a scheduling strategy should provides good spatial and temporal data locality and an efficient load balancing.

The used unit workload is a sequential function performing a "quicksort" on a small vector. We control the unit workload by varying the size of this small vector. So, our input data is a set of small vector, our program perform a "quicksort" in each of these small vector. "quicksort" sort a vector performing multiple compare and swap so intensive read/write accesses to data. This make it good candidate to evaluate efficient cache utilization.

In this experiment we try to evaluate the efficiency of our scheduling policy CHATS to: static scheduling, dynamic scheduling and task stealing. We run our synthetic workloads on an hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz) and we measure average execution time for different workload as well as cache-misses for each of the scheduling policies.

6.1.3.3 Results

As shown in Figure 6.15, results shows that XPU processes the heaviest workload about 20% faster than the second fastest candidate. We notes also that XPU become more efficient as the workload is bigger.

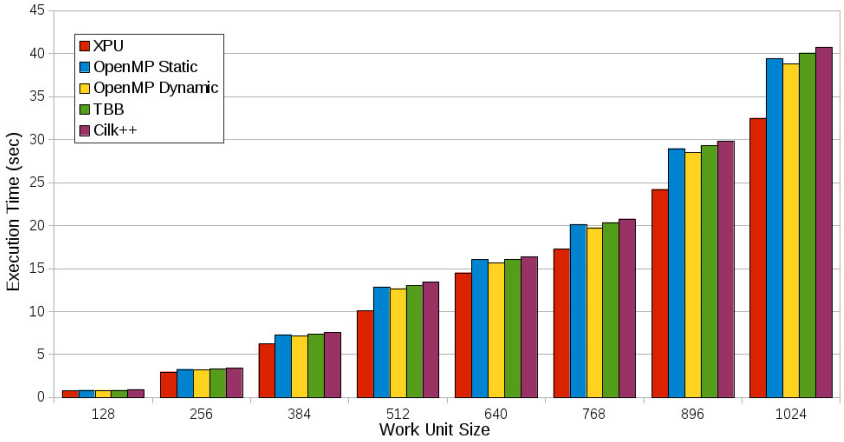


Figure 6.15: Average processing time of different workload sizes.

Figure 6.16 shows that XPU generates a low cache-miss rate in comparison to the other candidates. XPU cache-miss rate remains close to the static scheduling-based candidate. Static scheduling is known to offer very good data locality and does not introduce communication overheads.

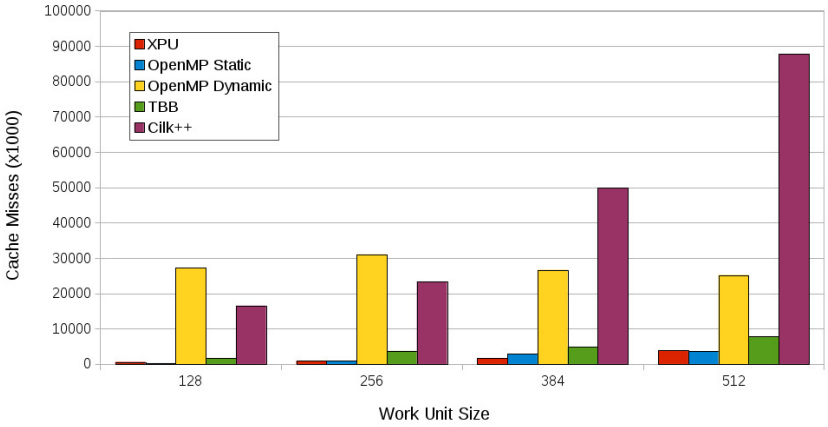


Figure 6.16: Cache-miss rate for different problem sizes.

6.2 Vectorization

6.2.1 SIMD Technologies

Most recent multicore and manycore processors implements thus ILP (Instruction Level Parallelism) by providing a set of SIMD (Single Instruction Multiple Data) instruction set. FIGURE 6.17 illustrates the difference between a traditional scalar operation on a single data pair to produce a single result and an SIMD operation which allows simultaneous additions of four pair of data to produce four results at once. SIMD operations can increase significantly computation throughput an improve processor use efficiency.

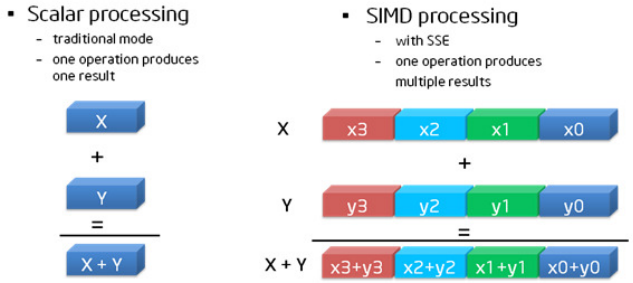


Figure 6.17: Traditional scalar operation on single data versus simultaneous SIMD operation on multiple data

ILP implementations are available in most recent monocore, multicore and manycore processors. Recent Intel and AMD processors implements the Streaming SIMD Extension (SSE) [RPK00] [Cor01] and MMX instruction set [PW96] which are a SIMD instruction set extension to the x86 architecture. SSE operates on 128 bits-wide register and allows simultaneous SIMD operations on eight chars, four integers, four simple precision floats or two double precision floats. Advanced Vector Extensions (AVX) [Cor14] [HTHW14] is an advanced version of SSE which can operate on wider registers 256 bits.

A more recent version of AVX named AVX-512 implements SIMD operations using 512 bits-wide registers [Cor14] [HTHW14]. Many others SIMD intrinsics are available on a variety of parallel hardware: FMA (Fused Multiply Add) [Cor14] [AMD09] instruction set has been introduced as and extension of the x86 architecture instruction set and is supported by recent AMD (starting with Bulldozer architecture) and Intel processors (starting with Haswell processors). XOP (eXtended OPerations) [AMD09] is another extension to the 128 bits SSE intrinsics and has been introduced in AMD Bulldozer processor core.

FIGURE 6.18 shows how SSE SIMD operations are performed on single precision and double precision floats using the 128 bits-wide XMM register. A single XMM register

can hold either two double precision floats (64 bits-wide each), four single precision float (32 bits-wide each), four integer (32 bits-wide each) or 16 char (8 bits-wide each).

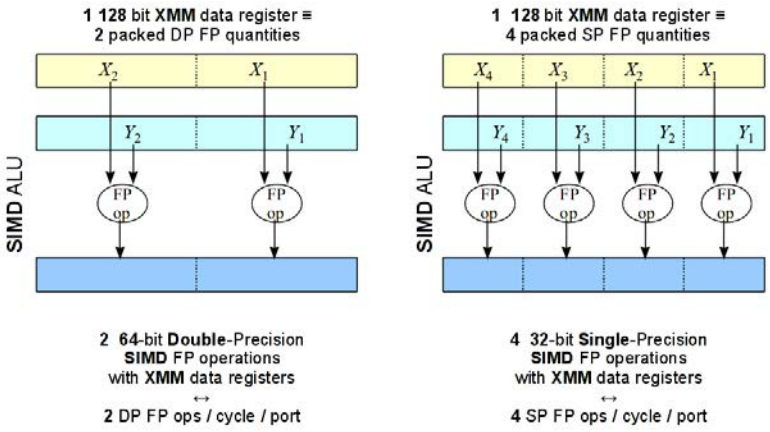


Figure 6.18: SIMD Operations using 128 bits-wide XMM register in single precision and double precision float configurations

Various SIMD operations types are implemented in the different SIMD instruction sets. Supported operations include arithmetic operations, logical operation, memory load/store operations, comparison intrinsics and conversion intrinsics. More recently, application-related instructions have been introduced: instructions performing rounds of the AES (Advanced Encryption Standard) and CRC (Cyclic Redundancy Check) algorithms are examples of such specialized instruction sets.

Despite their ability to provide significant accelerations, SIMD instruction sets are often difficult to use efficiently by non-expert programmers. In addition, they exacerbate potentially fragile cross-compiler portability. Recently, a C++ template library named Boost.SIMD [EGFL12] [EGF+12] attempted to address these problems through offering a high abstraction of the low level SIMD instructions. The later library aims to simplify the "*SIMDization*" while maintaining a good cross-compiler portability. Boost.SIMD inherits several paradigms of the Standard Template Library [SL94] such as the *iterators* and the *algorithms* which encapsulate several recurrent kernels. While easing vectorization of STL-based applications, fine-grain vectorization of existing code which do not use the STL primitives may require verbose restructuring of the target application algorithm.

6.2.2 XPU Vectorization Support

We tried to exploit widely available SIMD intrinsics and more specifically SSE instruction set to make vectorization more accessible to the average programmer. XPU provides

a vectorized built-in types which act like a regular scalar type and support traditional scalar operation while performing transparently SSE-related routines such as load/store operations and memory alignment requirements. These routines are handled internally and thus are transparent to the programmer. Basic operations such as addition, substraction, division or multiplication... are vectorized and implemented using the SSE2 instruction set. C++ operator overloading is used to hide this SIMD implementation behind a simple interface.

Functions	**XPU Vectorized Functions (SSE4.2)** *(Millions Floats / Sec)*	**Sequential (standard C math library)** *(Millions Floats / Sec)*
cos	240.9	44.78
sin	243.2	43.4
sincos *(= sin // cos)*	230.1	-
log	254.1	40.92
exp	253.3	34.7
sqrt	619.0	73.29
rsqrt *(= 1/sqrt)*	1 199.5	51.63
rcp *(= 1/x)*	1 215	198.4
+	1170.1	477.6
-	1180	477.2
/	1149	388.01

Figure 6.19: Vectorization can improve substantially floating operation throughput in comparison with traditional operations

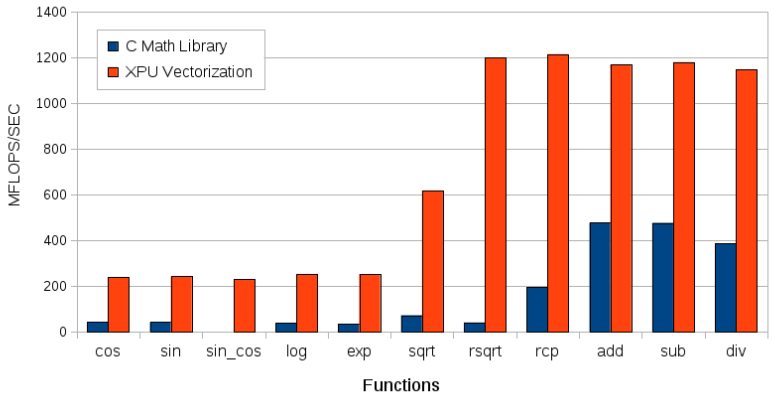


Figure 6.20: Functions vectorization offers significant floating operation throughput improvement in comparison with standard C math library functions implementations

6.3 Abstract Parallel Vector Interface

The vector data structure is a popular homogeneous data container which can contain a significant amount of data. This structure is particularly suited for massively parallel operations on different partitions of the target vector. Providing an abstract parallel vector interface allows the parallel processing of its element using different algorithms and on various multicore and many-core platforms: for instance Thrust [HB] is a parallel algorithms library which offers a C++ standard template library-like vector interface and inter-operates internally with CUDA, TBB or OpenMP to run on multicore and many-core architectures including CPU and GPU. PVL [KB09], PVTOL [KRS+08] and VSIPL++ [LKHR05] uses high-level vector, matrix and tensor data structures in conjunction with task maps and data maps to parallelize data processing and distributes the workloads across available processors [KB09].

OpenCL is established as a standard for heterogeneous computing [AM11] and was explicitly designed with abstractions that are low-level, high performance and portability [KB09]. The OpenCL programming interface requires explicit management of memory, kernels compilation and workload scheduling resulting in verbose code and requiring deep understanding of all software and hardware components.

In our programming model we defined an abstract parallel vector interface in top of OpenCL enabling the programmer to perform massively parallel operations on heterogeneous multicore architecture. Our vector interface uses the operator overriding feature of the C++ language to translate transparently basic operations (such as addition or subtraction...etc) into corresponding OpenCL kernel at run-time then manage transparently memory transfers and workload scheduling. Listing 6.2 shows how simple computation can be performed using the XPU vector container.

```
1  #define size 1000000
2  int main()
3  {
4     xpu::vector<float> A(size), B(size), C(size);
5     // Transparent operations on GPU/CPU/...
6     A = B + C;  // for i in [0..size[ do A[i] = B[i] + C[i]
7     C = A.sin(); // C[i] = sin(A[i])
8     B = C - A;  // B[i] = C[i] - A[i])
9  }
```

Listing 6.2: Parallel vector interface: OpenCL kernel is generated transparently and memory transfers are managed automatically.

6.4 Data Parallel Applications

In order to evaluate the performances of the XPU data parallel constructs, we use its "*parallel_for*" loop and its vectorization capabilities to parallelize two popular applications from the PARSEC benchmark suite [BKSL08], and we compare the XPU-based implementations with those provided in the PARSEC Benchmark. The provided parallel implementations use various parallel programming models such as OpenMP, Intel TBB or Pthreads. The first application is the "Fluid Animation" application which implements incompressible fluid dynamics and the second application is the "Black-Scholes" options pricing application.

6.4.1 Fluid Animation

The "*fluidanimate*" application is a part of the PARSEC Benchmark and the Intel RMS Benchmark [BKSL08]. It uses an extension of the Smoothed Particle Hydrodynamics (SPH) method and the Navier-Stokes equation to simulate an incompressible fluid for real-time animation purposes [MCG03]. The kernels of the "*fluidanimate*" application are specially designed to increase speed and stability. The output of this application can be visualized by rendering the surface of the fluid.

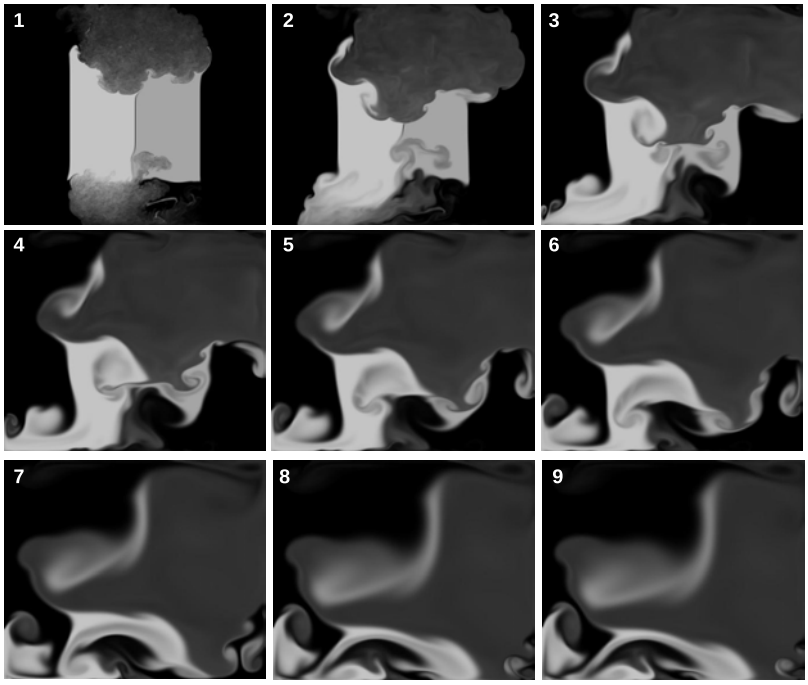


Figure 6.21: Example of incompressible fluid simulation based on the Navier-Stokes equation. We note that this simulation has been implemented using XPU and is based on an algorithm for real-time fluid animation which is described in [Sta03].

Figure 6.21 gives an overview of the output of an incompressible fluid simulation rendering. The simulation frames shown in the figure are taken from an XPU application that implements a real-time fluid dynamics simulator proposed by [Sta03] and based on a solver for the Navier-Stokes equations. The rendering of the output of the PARSEC "*fluidanimate*" application is similar to the Figure 6.21.

The "*fluidanimate*" application uses a generally accepted representation of fluids which is based on a set of PDEs called the Navier-Stokes equations. Figure 6.22 shows the simplified Navier-Stokes equation for a Newtonian incompressible fluid [PG92] which formulates the conservation of momentum.

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \mu\nabla^2\mathbf{v} + \mathbf{f}.$$

Figure 6.22: The Navier-Stokes equation for a Newtonian incompressible fluid which formulates the conservation of momentum.

The Naviers-Stokes equation can be solved using the SPH method. SPH was originally designed for astrophysics simulations, but has been applied to various continuous domains such as elastic solids and fluids. In the SPH, continuous quantities are represented as discrete particles which have a limited area of influence. These particles are not represented as point masses, instead they are fuzzy quantities which fade with distance. Figure 6.23 shows how smoothed particles are represented graphically and mathematically as a smoothing function.



Figure 6.23: Smoothed particle fade with distance.

The smoothing function is a kernel *W(r,h)* that computes a quantity which is attenuated with distance from the particle position r within a core radius h. Smoothing kernels are used to improve the accuracy and the stability of the SPH method. As shown in Figure 6.24, the SPH method defines a scalar quantity A at each location r as a weighted sum of all the particles: the index j iterates over all the particles which are defined by their respective positions r_j, their masses m_j, the densities ρ_j at their locations and their respective field quantities A_j. Each particle holds various properties like density, velocity, acceleration and can hold custom quantity such as temperature.

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j)$$

Figure 6.24: The SPH algorithm derives a scalar quantity A at location r by computing a weighted sum of all particles.

Similarly to solving point mass particle systems, the Navier-Stokes equation can be solved using the smoothed particle system by computing the pressure and the viscosity forces at each particles position (cf. Figure 6.25. The math behind each equation is not the purpose of this chapter and will not be detailed. For more information about Navier-Stokes equation solving, you can refer to [MCG03] and [PG92].

$$f^i_{pressure} = -\sum_j m_j \left(\frac{p_j + p_i}{2\rho_j}\right) \nabla W(r_j - r_i)$$

$$\mathbf{f}^i_{viscous} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(r_i - r_j)$$

Figure 6.25: The SPH algorithm derives a scalar quantity A at location r by a weighted sum of all particles.

6.4.1.1 The Algorithm

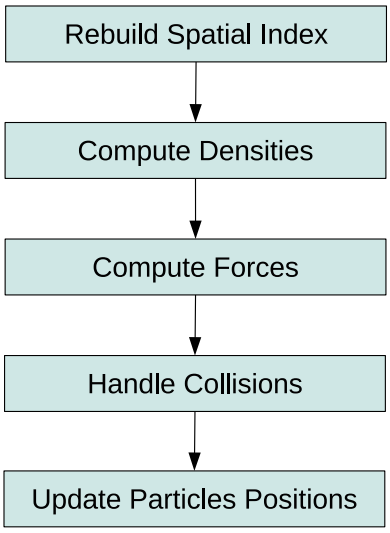


Figure 6.26: Every simulation step, the algorithm executes five kernels.

As depicted in Figure 6.26, the "fluidanimate" application executes these five kernels every time step :

1. **Rebuild spatial index** Because the finite support h of the smoothing kernels $W(r\text{-}r_j, h)$, the maximum interaction distance between particles is h. A spatial indexing structure is used to exploit the proximity information for limiting the number of particles which need to be evaluated [BKSL08]. This structure is built by the functions **ClearParticles** and **RebuildGrid** to be exploited by the next steps.

2. **Compute densities** This kernel computes the fluid density at the position of each particle. The density is higher in the position where particles are packed together more closely. This step is performed in three phases which are implemented in the functions **InitDensitiesAndForces**, **ComputeDensities** and **ComputeDensities2**.

3. **Compute forces** This kernel computes the forces using the densities computed in the previous step. This kernel is implemented in function **ComputeForces**. It computes pressure, viscosity and gravity. We note that in this step, collisions between particles are handled implicitly.

4. **Handle collisions** This kernel is implemented in function **ProcessCollisions** and is responsible of updating the forces to handle collisions of particles with the scene geometry.

5. **Update Position** This kernel uses the computed forces in the previous step to calculate the accelerations of the particles and their positions. The kernel is implemented in function **AdvanceParticles**.

6.4.1.2 Data Structure

The data structure is a three dimensional array that maps to the three dimensions geometry of the scene. The scene is represented as a box which is divided into cells. Each cell is associated to a particle of the fluid. Each kernel of the application iterates over all the cells to perform computations.

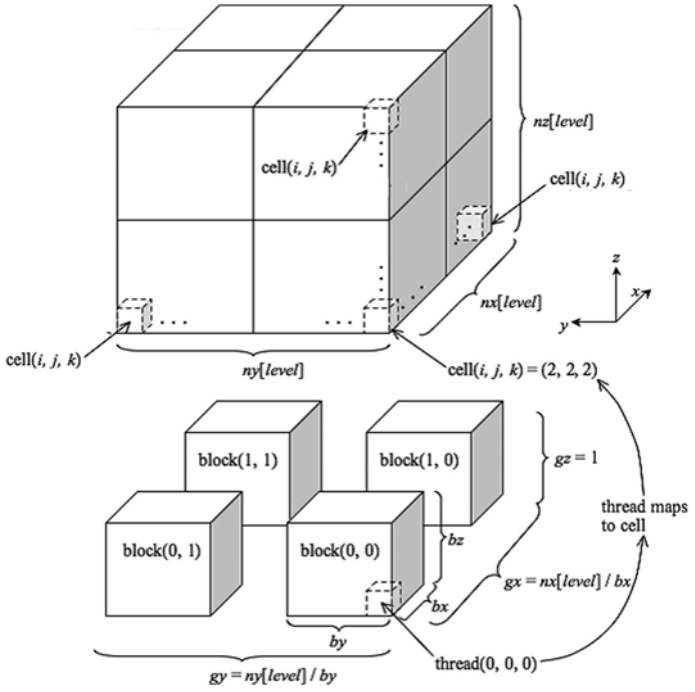


Figure 6.27: The three dimensions scene is represented as a box which is divided in small cells. These cells can be regrouped into blocks on which concurrent threads can operate.

Iteration over the cells in each kernel is implemented as a simple "*for*" loop. When parallelizing the application, these sequential "for" loops are replaced by parallel loops that execute computations on different cells or block of cells simultaneously.

6.4.1.3 Parallel Implementations

The parallel implementation of the "*fluidanimate*" application is based on a decomposition of the scene cube into blocks. Each block is a vertical grid which corresponds to a plan of cells. The application calls a function named "**AdvanceFrame**" at each simulation frame. This function invokes eight other functions which are *ClearParticlesMT*, *RebuildGridMT*, *InitDensitiesAndForcesMT*, *ComputeDensitiesMT*, *ComputeDensities2MT*, *ComputeForcesMT*, *ProcessCollisionsMT* and *AdvanceParticlesMT*. These kernels are executed in their sequential order. However, each of the kernels is executed simultaneously on different partitions (grids) of the data. Figure 6.28 gives an overview of the parallel implementation of the "*fluidanimate*" application.

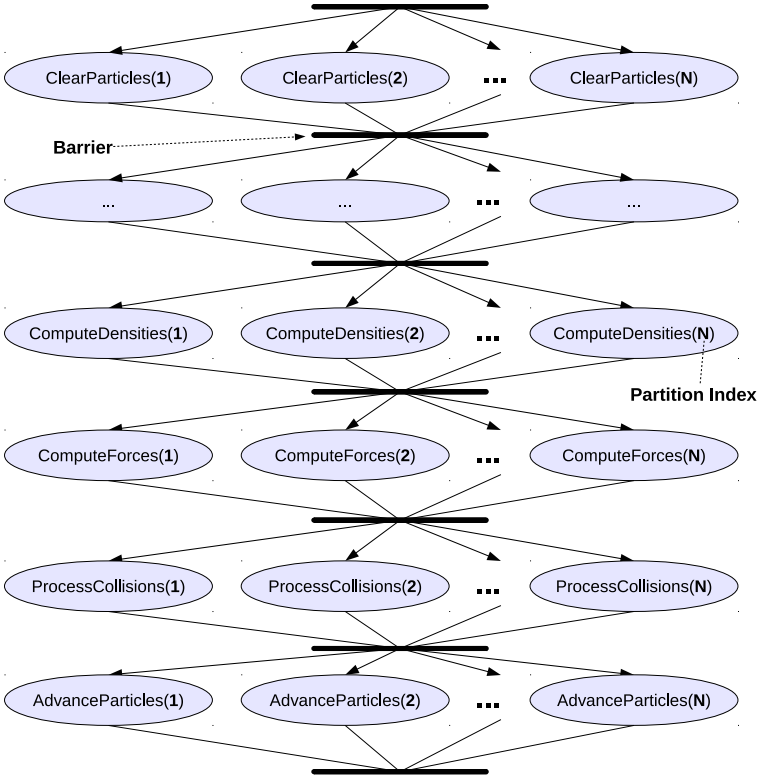


Figure 6.28: In the parallel implementation of the "*fluidanimate*" application, each kernel is executed simultaneously on multiple data partitions.

A Fluid Animation Applications In The PARSEC Benchmark

The PARSEC benchmark furnishes three parallel implementations of the "*fluidanimate*" application, the first implementation is based on the PThreads [NBF96] library and the second uses the Intel TBB [Phe08].

In the PThread version, each of the eight functions is used as a callback. At each step of the simulation, the application creates as many threads as data partitions using the corresponding function as a callback. The threads are synchronized on a common join point before executing the subsequent kernel.

The major drawback of this implementation is that the number of threads created at each kernel execution is proportional to the number of data partitions, consequently, large problem size with numerous data partitions can results in the creation of a very large number of threads. This can introduce a significant execution overhead. Moreover, the frequent creation and destruction of these short-life threads at its simulation step amplifies this execution overhead.

In the TBB version of the application, the eight functions are encapsulated in TBB task classes. Each of these classes is instantiated as many times as the data partition count. These tasks are then executed concurrently. At the opposite of the PThread implementation which create a thread to process each data partition, the TBB runtime has a persistent pool of threads and can execute multiple tasks by the same thread depending on the available processors. This reduces the execution overhead that can be introduced by thread creation and destruction.

B Parallel Implementation Of "fluidanimate" using XPU

The "*fluidanimate*" application can be parallelized using XPU in two different ways: using task parallel constructs or data parallel construct.

Task Parallel Implementation The first way of parallelizing the "*fluidanimate*" application consists in using the task parallelism API of XPU which we have seen in the previous chapter (*parallel* and *sequential*) to build the task graph shown in Figure 6.28. Listing 6.4 shows how the task graph can be built using the XPU *parallel-sequential* API. This implementation is similar to the PThread and TBB implementations provided in the PARSEC Benchmark. The XPU implementation requires much less parallelism-related extra-code than TBB and PThreads to express task parallelism. Moreover, almost all legacy code is reused without any alteration.

While offering a good programmability, the XPU task parallel implementation suffers from the same limitation of those provided in the PARSEC benchmark: the static data partitioning of the data determines the number of the concurrent tasks and thus limits the available parallelism. Consequently, the application exposes a poor scalability.

```
1
2  int main()
3  {
4     // task definition
5     task cp0(ClearParticles, 0), cp1(ClearParticles, 1), // ...
6     task rg0(RebuildGrid, 0),rg1(RebuildGrid, 1), // ...
7     task cf0(ComputeForces, 0), cf1(ComputeForces, 1) // ...
8     // ...
9
10    // task graph definition
11    xpu::task_group * simulation = sequential(parallel(&cp0, &cp1, &cp2, &cp3),
12                                              parallel(&rg0, &rg1,&rg2, &rg3),
13                                              parallel(&id0, &id1, &id2, &id3),
14                                              parallel(&cd0, &cd1, &cd2, &cd3),
15                                              parallel(&cdb0, &cdb1, &cdb2, &cdb3),
16                                              parallel(&cf0, &cf1, &cf2, &cf3),
17                                              parallel(&pc0, &pc1, &pc2, &pc3),
18                                              parallel(&ap0, &ap1, &ap2, &ap3));
19
```

```
20    // run the simulation
21    while (simulation_steps--)
22        simulation->run();
23  }
```

Listing 6.3: Parallel implementation of the "*fluidanimate*" application using XPU task parallelism primitives.

Data Parallel Implementation The data parallel implementation use the data parallelism capabilities of XPU at both thread level and instruction level. In the sequential implementation of the application, at each kernel execution, a simple "*for*" loop iterates over the data partitions and applies the kernel to each of them. We uses the XPU "*parallel_for*" construct to parallelize each of these eight loops. In addition to executing simultaneously the kernels on several data chunks, the "*parallel_for*" skeleton performs a dynamic data partitioning to achieve a high scalability: the XPU parallel loop adapts dynamically to the underlying platform and performs different data partitioning depending on the available processors to deliver a good forward scalability. Each of the eight kernels is replaced by a "*parallel_for*" loop. The eight "*parallel_for*" loops are executed successively at each simulation step.

```
1   int num_grids = 512; // partition count
2
3   /* wrapper function (required by XPU parallel for loop) */
4   ComputeForcesWrapper(int from, int to, int step)
5   {
6     for (int i=from; i<to; i++)
7         ComputeForces(i);
8   }
9
10  int main()
11  {
12    // task definition
13    task cp(ClearParticlesWrapper, 0, 0, 0);
14    task rg(RebuildGridWrapper, 0, 0, 0);
15    task cf(ComputeForcesWrapper, 0, 0, 0);
16    // ...
17
18    // task graph definition
19    xpu::task_group * simulation = sequential(new parallel_for(0, num_grids, 1, &cp),
20                                              new parallel_for(0, num_grids, 1, &rg),
21                                              new parallel_for(0, num_grids, 1, &id),
22                                              new parallel_for(0, num_grids, 1, &cd),
23                                              new parallel_for(0, num_grids, 1, &cdb),
24                                              new parallel_for(0, num_grids, 1, &cf),
25                                              new parallel_for(0, num_grids, 1, &pc),
26                                              new parallel_for(0, num_grids, 1, &ap));
```

```
27    while (simulation_steps--)
28        simulation->run();
29  }
```

Listing 6.4: Parallel implementation of the "*fluidanimate*" application using XPU parallel_for construct.

In order to compute the velocities, accelerations and various forces which are applied to the particles, the "*fluidanimate*" application uses a structure named "*Vec3*" which implements basic operations on 3-d vectors. We substituted the "*Vec3*" type with the XPU "*vec3f*" built-in type to take advantage of the XPU vectorization capabilities. The XPU "*vec3f*" implements all the operations which are implemented in "*Vec3*" using the x86 SSE2 SIMD Streaming extension. This allows us to speedup the execution of most of these operations in several regions of the application.

6.4.1.4 Performance Comparison

A Experimental Setup

We compares the average execution time of the three parallel implementations of "*fluidanimate*" on four different platforms. The TBB and PThread applications which are included in the PARSEC Benchmark are used as they are without any modification. We use the version 2 of the Intel TBB library. The XPU-based implementation of the application uses the *parallel_for* construct and the vectorized "vec3f" built-in type. We compile the three applications with the Intel Compiler v12.0.5 with SSE4.2 enabled.

We note that the PThread and TBB implementations requires explicit or "manual" specification of the desired thread number, we choose the number of threads which gives the better results in each configuration. The XPU implementation adapt dynamically to the underlying platform and therefore do not requires explicit specification of the thread number.

The applications are executed on four different platforms:

1. Multiprocessor platform with two AMD Opteron 252 K8 processors (1 x Core, 1 x Thread, 1 x L1 Cache 64KB, 1 x L2 Cache 1MB, at 2.8 GHz).
2. One Intel Core i7 Q720 Nehalem (8 x Hardware Threads, 4 x Cores at 1.6 GHz, 4 x L1 Cache 32KB, 4 x L2 Cache 256KB, 1 x L3 Cache 6MB), with Turbo Boost and Hyperthreading technologies enabled.
3. Multiprocessor and multicore platform with two Intel Xeon X5472 processors (4 x Cores, 4 x L1 Cache 32KB, 2 x L2 Cache 6MB, at 3 GHz).
4. Multiprocessor-Multicore platform with two Intel Xeon E5620 processor (8 x Threads, 4 x Cores at 2.4 GHz, 4 x L1 Cache 128KB, 4 x L2 Cache 1MB, 1 x L3 Cache 12MB), with Turbo Boost and Hyperthreading technologies enabled.

B Results

Figures 6.29, 6.30, 6.29 and 6.29 show the achieved performances on the various platform. Unexpectedly, the Intel TBB version of the application achieved lower performances than all the other version and in almost all the configurations. This can be explained by the limited performance of task stealing in communication-intensive applications: the "*fluidanimate*" algorithm uses mutual exclusion mechanisms (*mutexes*) to preserve data (particles) from *race condition*, this introduces an intensive inter-thread communication and make data locality crucial for performances. As we have shown in our experiments on scheduling strategies in the beginning of this chapter, task stealing offers theoretically good trade-off between load balancing and data locality, however our experiments on several implementations of task stealing including Cilk Plus and TBB implementations generated relatively high cache-miss rate when used with our synthetic workloads. The experiment on a real application (*fluidanimate*) outlined the same issue on various platforms.

Thanks to the XPU CHATS algorithm, the XPU implementation take advantage of the cache-aware scheduling to reduce the communication overhead and achieve good spatial and temporal data locality. In addition, the vectorization of a significant part of the computations speedups the execution of several kernels of the simulation. The XPU implementation of "*fluidanimate*" achieves higher performances than the PThread and TBB implementations in almost all the tested configurations. However, the execution times of the PThread implementation remains close to those of XPU.

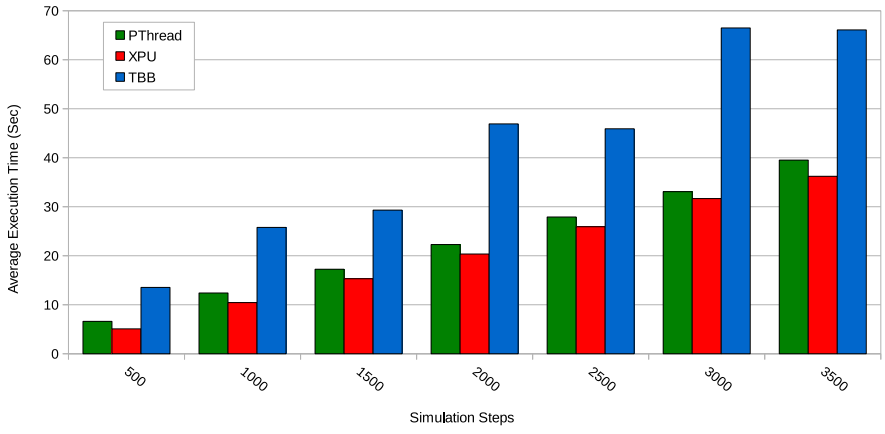


Figure 6.29: Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with two AMD Opteron 252 K8 (L1 64 KB, L2 1M, 2.8 GHz).

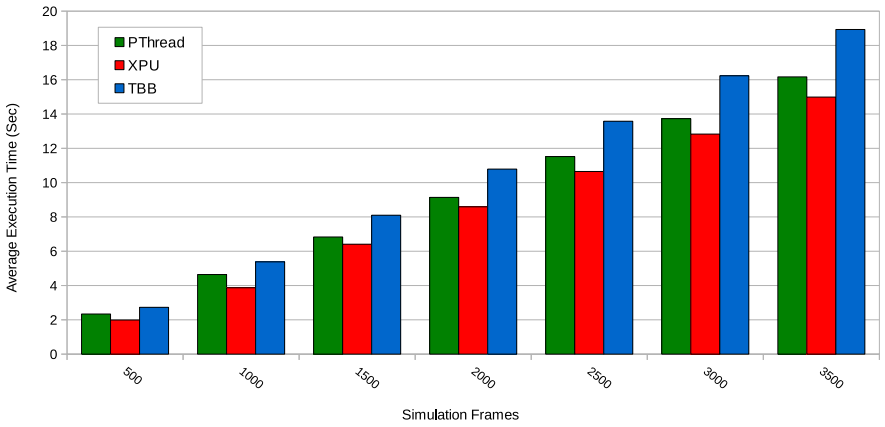


Figure 6.30: Execution time of the "Fluid Animation" application for different problem sizes on hybrid SMT-CMP Nehalem Processor (Intel Core i7 Q720) with 8 Hardware threads.

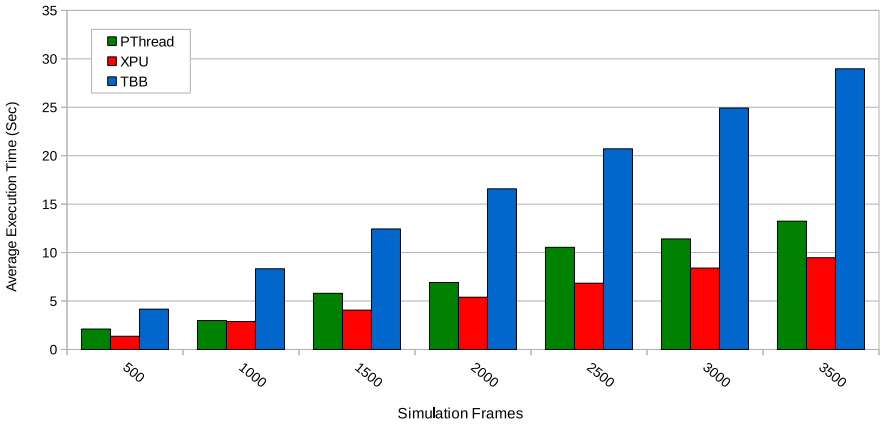


Figure 6.31: Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with 2 x Core 2 X5472 Processor (4 Cores L1 32KB, L2 6MB, 3 GHz .

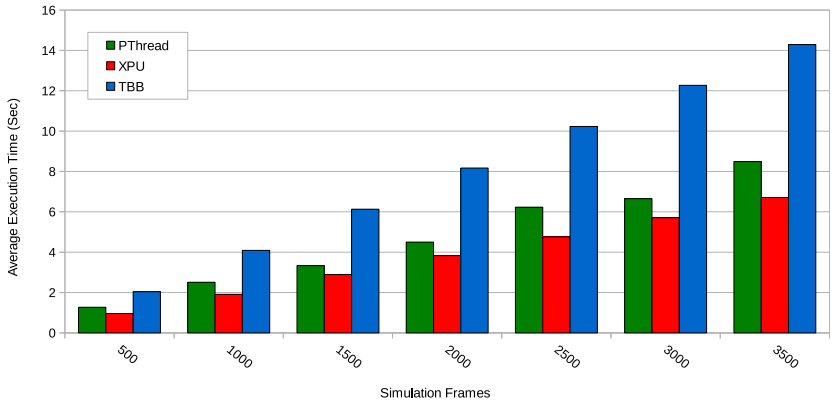


Figure 6.32: Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with 2 x Intel Xeon E5620 processors.

6.4.1.5 Programmability Comparison

Figure 6.33 shows a comparison of the line count of the sequential version and the different parallel versions of the "Fluid Animation" application. We note that XPU is vectorized while the PThread and TBB versions are parallelized at thread-level but not vectorized.

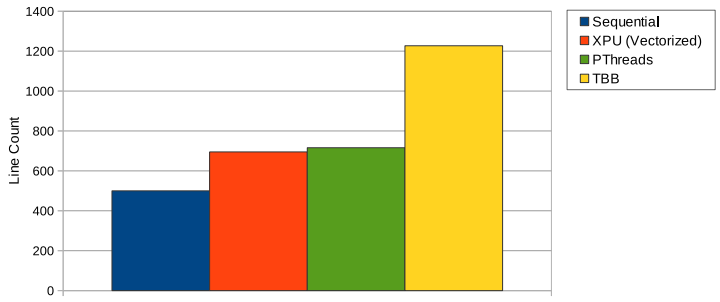


Figure 6.33: Programmability Comparison of the "Fluid Animation" application: Line count of the sequential version and the parallel versions using XPU, PThreads and TBB.

The result shows that the XPU version requires less parallel code that PThreads and TBB despite the vectorization-related code included in the XPU version. The XPU version introduce 95 lines of extra-code while the PThreads and the TBB versions requires respectively 116 and 727 lines of code. Thus, the TBB requires 6 times more lines of code than XPU to parallelize the "Fluid Animation" application while delivering lower performances than XPU on most of the tested platforms.

6.4.2 BlackScholes

The Black-Scholes model or Black-Scholes-Merton is a mathematical model of a financial market containing a set of derivative investment instruments. The Black-Scholes formula can be deduced from the model and gives the price of European-style options. The formula led to a boom in options trading and gave a scientific legitimacy to the activities of the Chicago Board Options Exchange and other options markets around the world [Mac08]. It is widely used by options market participants and many empirical tests have shown the Black-Scholes price is "fairly close" to the observed prices. When applied to a large number of options, the Black-Scholes algorithm exhibits massive data parallelism at both thread level and instruction level. We used this algorithm as another study case to evaluate our framework (XPU) and compare it to state-of-the-art parallel models.

6.4.2.1 The PARSEC "blackscholes" Application

The *blackscholes* application was included in the PARSEC benchmark to represent the wide field of analytic PDE solvers in general and their application in computational finance in particular [BKSL08]. The program is limited by the amount of floating-point computations that processor can perform. The "*blackscholes*" stores the options data in a large array. The parallel implementation of the program divides the data into a number of work units which corresponds to the number of the concurrentthreads that processes the options. Each of these threads iterates through all options and calls the "*BlkSchlsEqEuroNoDiv*" function to process each of them and compute its price.

Contrary to the "*fluidanimate*" application in which the data parallelism is limited by an intensive communication, the "*blackscholes*" application exhibits massive data parallelism and do not requires inter-thread communication (the options are processed independently). This make the parallelization of the "*blackscholes*" application relatively simple.

The PARSEC benchmark includes five parallel implementations of "*blackscholes*" in addition to a sequential one which serve as a reference or base code. The five applications are parallelized using respectively OpenMP, TBB, PThreads, OpenMP/SSE and PThreads/SSE.

6.4.2.2 "Blackscholes" Parallelization Using XPU

Similarly to the "*fluidanimate*" application, we used XPU to parallelize the popular "Black-Scholes" problem at thread level using the "*parallel_for*" construct and at the instruction level using the vectorization capability provided by XPU through the built-in vectorized type (vec4f) implemented on top of SSE to support SIMD. We used the sequential code of the "*blackscholes*" application as provided in PARSEC Benchmark Suite [BKSL08].

The main processing loop was parallelized at the cost of 3 lines of extra-code. Vectorization was introduced simply by replacing regular float type by the "vec4f" vectorized type and by setting increment step of "*parallel_for*" to 4 instead of 1. Listing 6.5 shows the original implementation of the Black-Scholes Cumulative Normal Distribution Function (CNDF) and Listing 6.6 shows it vectorized equivalent which use the vectorized XPU "*vec4f*" type. The codes are very closes since all vectorization details (often written using SSE assembly) are hidden behind the operators of the "*vec4f*" type.

```
#define ftype float

/**
 * Cumulative Normal Distribution Function
 */
fptype CNDF (fptype InputX )
{
    int sign;
    fptype OutputX;
    fptype xInput;
    fptype xNPrimeofX;
    fptype expValues;
    // ...

    xInput = InputX;
    expValues = exp(-0.5f * InputX * InputX);
    xNPrimeofX = expValues;
    xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI;

    xK2 = 0.2316419 * xInput;
    xK2 = 1.0 + xK2;
    xK2 = 1.0 / xK2;
    xK2_2 = xK2 * xK2;
    xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2;
    xK2_5 = xK2_4 * xK2;
    xLocal_1 = xK2 * 0.319381530;
    xLocal_2 = xK2_2 * (-0.356563782);
    xLocal_3 = xK2_3 * 1.781477937;
    // ...
    xLocal   = xLocal_1 * xNPrimeofX;
    xLocal   = 1.0 - xLocal;

    OutputX = xLocal;
    // ...
    return OutputX;
}
```

Listing 6.5: Original Black-Scholes Cumulative Normal Distribution Function (taken from the reference sequential application)

```
1
2  /**
3   * Cumulative Normal Distribution Function (Vectorized with XPU)
4   */
5  void CNDF(float * OutputX, float * InputX)
6  {
7      // ...
8      xpu::vec4f xInput(InputX);
9      xpu::vec4f xK2;
10     xpu::vec4f xK2_2, xK2_3, xK2_4, xK2_5;
11     xpu::vec4f xLocal, xLocal_1, xLocal_2, xLocal_3;
12     // ...
13
14     xpu::vec4f xNPrimeofX = (xInput * xInput * (-0.5f)).exp() * (inv_sqrt_2xPI);
15
16     xK2 = xpu::vec4f(0.2316419) * xInput;
17     xK2 = xK2 + xpu::vec4f(1.0);
18     xK2 = xpu::vec4f(1.0) / xK2;
19
20     xK2_2 = xK2 * xK2;
21     xK2_3 = xK2_2 * xK2;
22     xK2_4 = xK2_3 * xK2;
23     xK2_5 = xK2_4 * xK2;
24
25     xLocal_1 = xK2 * (0.319381530);
26     xLocal_2 = xK2_2 * (-0.356563782);
27     xLocal_3 = xK2_3 * (1.781477937);
28     xLocal_2 = xLocal_2 + xLocal_3;
29     // ...
30     xLocal   = xLocal_1 * xNPrimeofX;
31     xLocal   = xpu::vec4f(1.0) - xLocal;
32
33     xLocal >> OutputX;
34     // ...
35 }
```

Listing 6.6: Vectorized version of Black-Scholes Cumulative Normal Distribution Function using XPU

Listing 6.7 shows the main function of the XPU implementation. The main "*blackscholes*" kernel is encapsulated in a task which is used in the definition of the XPU "*parallel_for*" loop. We note that the progression step of the parallel loop is set to "vec4f_size" which is equal to 4 and corresponds to the 4 floats which are stored in the "vec4f" structure instead of the regular single float (line 21).

```
1
2  int bs_kernel(int from, int to, int step)
3  {
4    // ...
5    for (int i=from; i<to; i+=step)
6    {
7      BlkSchlsEqEuroNoDiv(price, NCO, &(sptprice[i]), &(strike[i]), &(rate[i]),
8                          &(volatility[i]), &(otime[i]), &(otype[i]), 0);
9      for (k=0; k<step; k++)
10         prices[i+k] = price[k];
11   }
12 }
13
14 #define vec4f_size 4
15
16 int main()
17 {
18     xpu::init();
19
20     xpu::task t(bs_kernel,0,0,0);                 // task definition
21     xpu::parallel_for p(0,numOptions,vec4f_size,&t); // parallel loop definition
22
23     xpu::timer tmr;
24     tmr.start();
25
26     for (int j=0; j<NUM_RUNS; j++)
27       p.run();
28
29     tmr.stop();
30     printf("[+] execution time: %lf \n", tmr.elapsed());
31     xpu::clean();
32 }
```

Listing 6.7: The XPU main implementation of "*blackschole*" is based on a parallel_for loop which uses the processing kernel, namely "*BlkSchlsEqEuroNoDiv*"

6.4.2.3 Performances Comparison

A Experimental Setup

We compared the performance achieved by our application to the five parallel versions which are provided in the PARSEC benchmark suite: OpenMP, TBB, Pthreads, OpenMP/SSE and PThreads/SSE. We used the Intel C++ Compiler v12.0.5 and we executed our benchmark on several multicore platforms.

In XPU, optimal thread count is determined automatically by the runtime system. The XPU scheduler exploits the CHATS algorithm to combines static workload scheduling and task stealing to exploit effectively processor caches and ensure dynamic load-balancing. For the other programming models, the number of threads is fixed manually, we choose the thread count giving the best results after several tests on each platform.

The applications are executed on three different platforms:

1. Intel Core 2 Duo E8600 Processor (2 Cores, 2 Threads, 3.3 GHz, 2 x L1 Cache 32KB, 1 x L2 Cache 6MB).

2. One Intel Core i7 Q720 Nehalem (8 x Hardware Threads, 4 x Cores at 1.6 GHz, 4 x L1 Cache 32KB, 4 x L2 Cache 256KB, 1 x L3 Cache 6MB), with Turbo Boost and Hyperthreading technologies enabled.

3. Multiprocessor-Multicore platform with two Intel Xeon E5620 processor (8 x Threads, 4 x Cores at 2.4 GHz, 4 x L1 Cache 128KB, 4 x L2 Cache 1MB, 1 x L3 Cache 12MB), with Turbo Boost and Hyperthreading technologies enabled.

B Results

Figures 6.34, 6.35 and 6.36 shows the measured execution time for each parallel version. The XPU-based application provides higher performance than the other versions and execute up to 25 % faster than POSIX Thread/SSE one. It takes advantage of the ability of the scheduler to provide both load-balancing, efficient cache utilization and low communication overhead to outperform the POSIX Thread version which use basic static scheduling achieving good cache utilization but poor load-balancing. The impact of this poor load-balancing issue becomes more visible as workload grows.

Additionally, the XPU vectorization is relatively more sophisticated than that used in the SSE PThread and OpenMP implementations: the XPU "*vec4f*" vectorizes some operation such as exponential and logarithm computation which are not available in the SSE native instruction set but can be implemented using a combination fo the SSE floating point primitives. Figure 6.37 gives an overview of the scalability of the different implementations on version of the application on several platforms ranging from to 2 to 16 hardware threads.

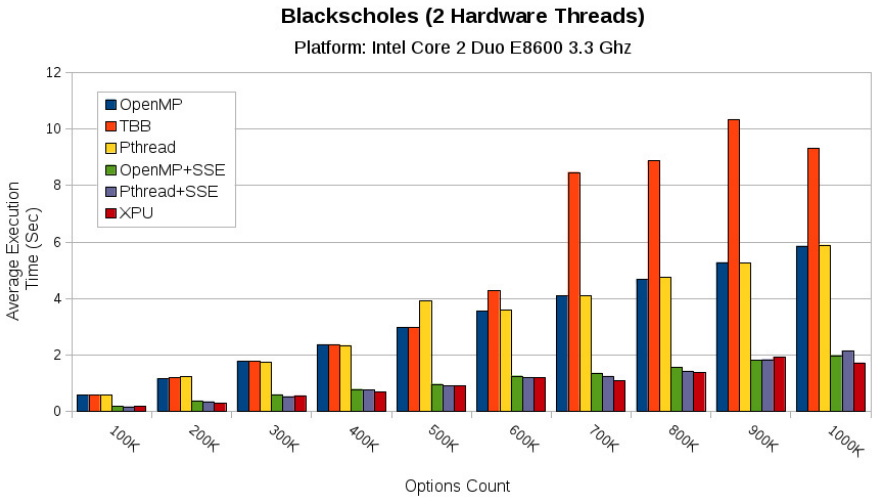


Figure 6.34: "*Blackscholes*" execution time on Intel Core 2 Duo E8600 Processor (2 Cores) for different problem size.

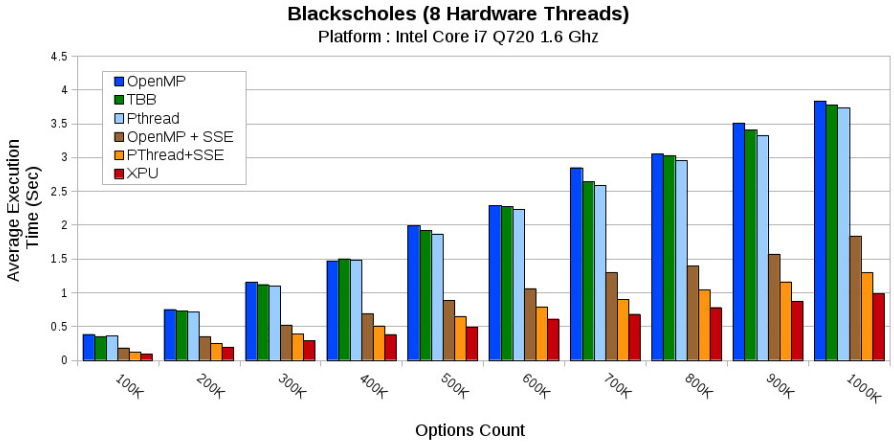


Figure 6.35: "*Blackscholes*" execution time on 8 Threads Intel Core i7 Q720 Processor

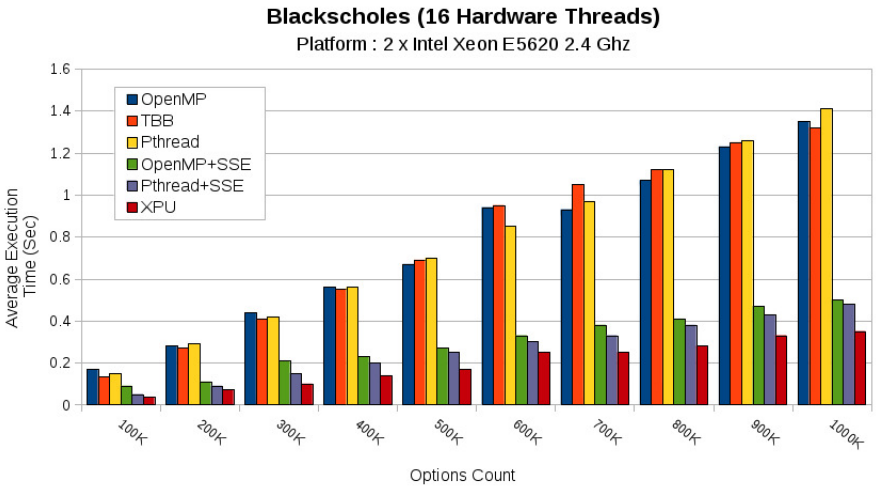


Figure 6.36: "*Blackscholes*" execution time on a 16 Threads SMP platform with two Intel Xeon E5620 Processor at 2.4 GHz

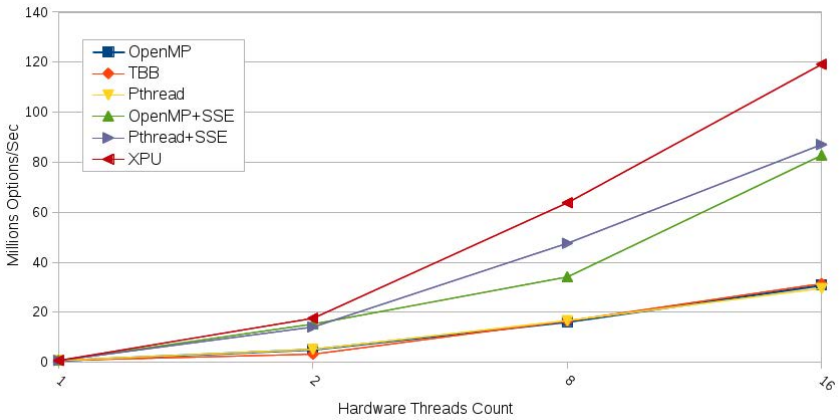


Figure 6.37: Black-Scholes Scalability (2 to 16 Hardware Threads).

6.4.2.4 Programmability Comparison

Figure 6.38 shows a comparison of the line count of the sequential version and the different parallel versions of the Black-Scholes application. We note that XPU is vectorized while the OpenMP, PThread and TBB versions are parallelized at thread-level but not vectorized. The result shows that despite the vectorization-related code, the XPU version remains very close to both the sequential version and the OpenMP ver-

sion which introduces the littlest parallelism-related code. The PThreads-based code is significantly more verbose but still less verbose than the TBB code.

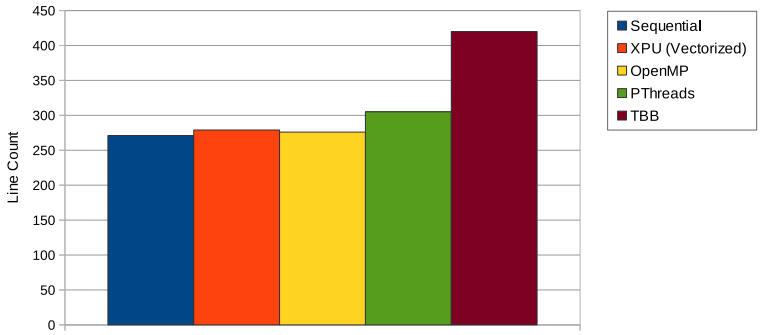


Figure 6.38: Black-Scholes Programmability Comparison: Line count of the sequential version and the parallel versions using XPU (vectorized), OpenMP, PThreads and TBB.

We reproduce the same experiment with the vectorized versions of OpenMP and PThreads. Figure 6.39 shows that XPU introduces less than 20 lines of code to parallelize the main loop and vectorize the main Black-Scholes kernel. The OpenMP version requires about 80 lines to implement the parallel loop and vectorize the code using explicitly the SSE instruction set. Finally, PThreads version introduces near 90 lines of code to do the same as OpenMP.

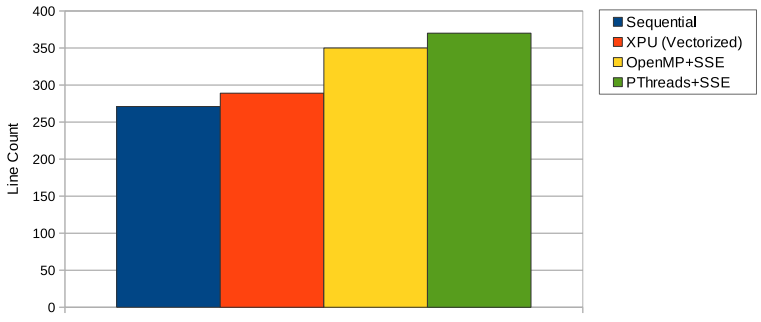


Figure 6.39: Black-Scholes Programmability Comparison: Line count of the sequential version and the vectorized parallel versions using XPU (Vectorized), OpenMP+SSE, PThreads+SSE.

6.5 Conclusion

In this chapter we have shown that data parallelism, which is crucial for todays applications, can be described easily in XPU provided that the application do not expose complex data dependencies. We have shown how we can express easily data parallelism at thread level using the XPU *parallel_for* construct and at instruction level using the built-in XPU vectorized types. We outlined the programmability provided by the XPU programming interface and shown that despite its high abstraction, it can still achieve comparable performances to lower level programming models. We described the details of implementation the XPU parallel for loop and its feature including the scalable data partitioning and cache-aware scheduling. We presented the XPU CHATS algorithm which allows the XPU parallel "*for*" loop to adapt dynamically to the underlying platform and its cache hierarchy and to perform efficient cache-aware scheduling.

The XPU "*parallel_for*" construct can parallelize only loops with no dependencies between theirs iterations. Unfortunately, in many applications such as image processing, loops expose often dependencies between their iterations and requires specific transformations to be parallelized. Since XPU cannot perform code analysis as compilers would do, the XPU runtime cannot extract these dependencies between iterations else the programmer provides them. In a future implementation, the programming interface of the parallel for construct will be extended to allow the programmer to specify dependencies between iterations. This would enable the XPU runtime to perform advanced transformations before parallelizing the loop. In the second part of this thesis, we see how such complex data dependencies analysis can be performed transparently when using the FATMA framework provided that the loop is *unrolled*.

7 Pipeline Parallelism

Pipeline execution pattern is a recurrent execution configuration in many application domains involving stream processing such as digital signal processing and data compression. Despite its popularity and recurrence in image, signal processing, graphics, data compression and many application implying real-time stream processing, pipeline parallelism is still difficult to express and implement efficiently on multicore platforms: low-level parallel programming models exacerbate the difficulties of expressing pipeline parallelism and require verbose restructuring of the code and complex scheduling techniques to perform efficient execution on modern multicore architectures.

High-level programming models are in high-demand as they reduce the burdens of programmers, ease parallelism expression and handle transparently tasks scheduling and communication. In this chapter [1] we focus on pipeline parallelism expression using XPU, we present the XPU programming interface which allows pipeline construction. We describe its internal design and the run-time implementation of the pipeline execution pattern and finally we show an example of image processing application implementing real-time adaptive edge detection algorithm. We reuse an existing sequential implementation to implement a pipelined version using both XPU and TBB. We compare the two versions in term of expressiveness and performance. We note that XPU version performs about 20% faster than TBB on a 16-threads multicore platform and requires 80% less extra-code than TBB to express pipeline parallelism. Our experiments show that our programming model provides both programmability and execution efficiency.

7.1 Pipeline Execution Pattern

Pipeline execution pattern follows a consumer/producer scheme similar to a production assembly line. An assembly line consists of a sequence of stations in which each station is responsible for assembling a part of a product. Each station may has one or multiple workers assigned and the output of each station (producer) is consumed by the adjacent station (consumer).

[1] **Publication**: Nader Khammassi and Jean-Christophe Le Lann, "***A High-Level Programming Model to Ease Pipeline Parallelism Expression On Shared Memory Multicore Architectures***", 22nd High Performance Computing Symposium, ACM HPC 2014, Tampa, FL, USA



Figure 7.1: Pipeline Execution Pattern: Consumer/Producer relationship between pipeline stages.

Analogously, as depicted in FIGURE 7.1, a pipeline is set of simultaneously active tasks called "stages" that communicate following a producer-consumer relationship: each stage is responsible of both consuming and producing some items of interest. Thus, each pair of adjacent stages forms a producer-consumer pair.

At the opposite of the serial execution pattern where completely dependent sequence of tasks are executed serially, pipeline stages are activated at the same time. However, in order to recover data coherency, sequentially dependent activities or "folds" are serialized and parallelism is exploited only on independent activities [McC10]. FIGURE 7.2 and 7.3 shows a comparison between serial and two-stage pipeline execution. It illustrates how a pipeline can exploit available parallelism by serializing only dependent activities. We note potential improvement of throughput with pipeline parallelism. This improvement may be proportional to the pipeline depth but is limited by the available computing resource particularly when the pipeline stages count exceed processors count [GPB+07]. Therefore, despite their fixed stages count limiting their scalability to large number of cores, pipelines can still provide a very useful parallelism multiplier in programs exposing strong serial dependencies between tasks.

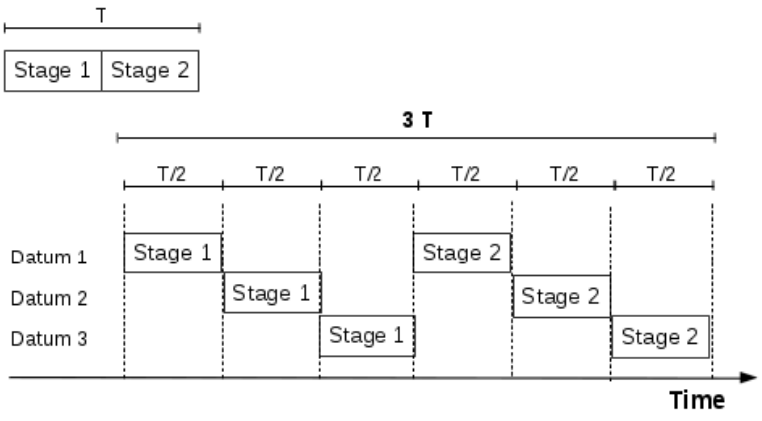


Figure 7.2: Serial Execution: Stage 1 and 2 are executed sequentially on the same processor.

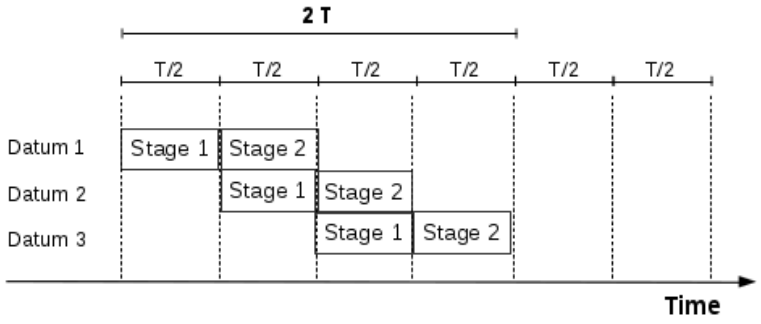


Figure 7.3: In Pipeline Execution Pattern, only dependent tasks are serialized.

For example, an image processing pipeline composed of a set of filtering stages will ensure that filters will be executed serially on each input image, while it will allows several images to be processed simultaneously by parallel stages.

7.2 Pipeline Parallelism Expression

Decomposing a program into pieces of code then specifying their execution configuration are the two major steps of the parallelization process in most parallel programming models. In low-level thread-lock programming model such as "PThread", these pieces of code are called callbacks and require often significant alteration of the targeted piece of code since it has to conform to a specific callback prototype.

In higher level skeleton-based programming model, these pieces of code are called tasks and their execution configuration are specified by selecting the appropriate skeleton or execution pattern that specifies parallelism of these tasks. Tasks allow the reuse of sequential code in less restrictive way: task code and its consumed or produced data are often encapsulated in a class such as in the Intel TBB library [Phe08] or in the PTL [LSB09]. XPU does not require to define a class with a specific prototype for each task, instead it enables programmer to define task from a piece of code which may be a function, object method or lambda expression without particular constraints on function prototype (return value, arguments count or type). Thus, it allows reuse of legacy code almost without any alteration at the cost of a single line of code. In the next few paragraphs we describe how a pipeline can be defined in XPU and TBB then we compare their expressiveness and programmability in our study case.

7.2.1 The Threading Building Blocks Pipeline Pattern

In many high-level programming model task definition may require verbose restructuring of code and significant alteration of the original sequential code. For example, in TBB, task code and its consumed or produced data should be encapsulated in a class respectively as object attributes and object methods with specific prototype. Task class

has slightly different interfaces depending on the target execution pattern. In the case of the pipeline execution patterns, a specialized class interface named "*filter*" should be implemented to host the code of each pipeline stage.

In TBB, programmer code should override the "*void * operator()(void* token)*", where the token is the item of interest which will be processed by the stage. Each intermediate filter takes an input token, process it then produces an output token. The first *filter* does not have any input tokens, it only produces tokens. Analogously, the last *filter* processes input tokens but do not produce output ones. LISTING 7.1 shows an example of TBB "*filter*" implementation.

```
1  // original function
2  int blur(char * img, int width, int height);
3  // filter implementation
4  class blur_fltr : public tbb::filter
5  {
6    public:
7
8      blur_fltr(int width, int height) : filter(serial_in_order),
9                                         m_width(width),
10                                        m_height(height)
11     {/* ... initialization ... */ }
12
13   private:
14
15     // store required parameters as attributes
16     int m_width;
17     int m_height;
18
19     // overload 'operator()' with the stage code
20     void* operator()(void*img)
21     {
22      image_t * img = (image_t*)img;
23      // call original function to process current datum 'img'
24      blur(img, m_width, m_height);
25      // return produced datum to next stage
26      return (void*)img;
27     }
28
29 };
```

Listing 7.1: Pipeline filter implementation using TBB

Once pipeline stages are defined as filters, the pipeline can be built by adding instances of the implemented filters to a pipeline object as shown in LISTING 7.2. Pipeline can then be executed by invoking the "run" method.

```
int main()
{
   // ...
   // create the pipeline
  tbb::pipeline pipe;
   // instantiate filters
   greyscale_fltr stage1(width,height);
   blur_ftr     stage2(...args...);
   threshold_fltr stage3();
   // add filters to the pipeline
   pipe.add_filter(&stage1);
   pipe.add_filter(&stage2);
   pipe.add_filter(&stage3);
   // execute the pipeline
   pipe.run();
```

Listing 7.2: Pipeline construction using TBB

7.2.2 The XPU Pipeline Pattern

Since promoting the reuse of sequential code is one of the primary design goals of XPU, we tried to overcome the previously enumerated limitations such as definition of a dedicated class for each task through a more flexible task design. In the XPU, by design, a task is basically an abstract callable piece of code which can be executed. This piece of code may consume or produce data. Task is defined generally through a single line of code. Data are passed in the form of arguments to each task.

Listing 7.3 shows how a task can be created from a function, a lambda expression or an object method disregarding its return type or its argument count or type. We note that in the particular case of the pipeline execution pattern, the first argument of the task has to be an integer. This argument is used to indicate the index of the item processed by the pipeline stage.

```
// original function
int blur(char * img, int width, int height);
// stage wrapper ('i' the index of the processed image)
int blur_stg(int i, image_t * imgs, int w, int h) {
 blur(imgs[i],w,h);
}
// task definition
xpu::task blur_t(blur_stg, 0, stream, width, height);
```

Listing 7.3: XPU Task definition using a simple function

```
int main()
{
   // ...
   task greyscale_t(greyscl_stg, 0, stream, width, height);
   task blur_t(blur_stg, 0, stream, width, height);
   task threshold_t(threshold_stg, 0, stream, width, height);
   // create the pipeline
   task_group * process_image = pipeline(n, &greyscale_t, &threshold_t, &blur_t);
   process_image->run(); // execute the pipeline
}
```

Listing 7.4: An example of a four stages pipeline construction in XPU (Tasks are used as pipeline stages)

As shown in Listing 7.4, once tasks are defined, pipeline can be constructed using these tasks as stages. Tasks are passed as argument in their sequential execution order. They will be executed simultaneously in separate threads but will be synchronized to ensure data coherency. Pipeline can be executed simply by calling its run method. Stages synchronization is transparently handled by the internal run-time system. First argument "**n**" indicate to the pipeline the number of items to be processed before stopping. For indefinite datum counts, this argument can be set to zero.

7.3 Internal Design

The TBB pipeline execution model is based on MacDonald's work [MSS04] which rethinks pipeline as object-oriented states with transformations. It follows a tasking model which attempts to perform efficient execution by providing both good load-balancing and efficient memory use. In this section we discuss the internal design of the pipeline construct in XPU. This design is based on the producer-consumer relationship between pipeline stages. Pipeline stages communicate with each others to ensure synchronization and guarantee that no stage will overlap or interfere with its relative neighbor stage(s). Each stage communicates with its adjacent stage(s) to perform one of these two following operations or both of them:

- Wait for data
- Notify its following stage that data is available for processing.

Consequently we can distinguish three different types of stages:

- *The head of the pipeline*: the head of the pipeline is the first stage which only produces items of interest so it does not wait for any other stage, instead, it processes data. Once processing is finished, it notifies the next processing stage that data is available for processing then it continues processing more data following the same scheme.

- *Intermediate stages*: an intermediate stage waits for data to be produced by its predecessor, when it receives a notification from that stage, it starts processing the new data. When finished it notifies the next stage that data is available.
- *The tail of the pipeline*: the last stage of the pipeline waits for data from its predecessor stage then processes it. Since there is no more stages, it doesn't have to notify any other stage for available data.

As we have seen, pipeline stages in our model need to communicate with each others to preserve data coherency and enforce stages synchronization. An event-based mechanism is used to perform the two needed communication actions: "**wait**" and "**notify**".

7.3.1 Event object

An event object has been designed to allow directional asynchronous communication between two concurrent threads. We note that a common event object can be used to establish communication between a couple of threads. The event object has two methods: "**wait**" and "**notify**" respectively to wait for an event from a thread or signal an event to another thread:

- The "**wait**" method blocks the calling thread until an event is signaled.
- The "**notify**" method signals an event to a peer thread waiting that same event. If no thread is waiting in the time of event signaling, signals are not lost, instead, they are accumulated in a FIFO queue so when a thread invoke the "**wait**" method later, it does not block if there is available signals in the FIFO: the waiting thread simply dequeues the FIFO queue then continues without blocking. This guarantees that an event waiter does not miss any event even when it is busy.

In the XPU framework, event object is implemented using the traditional phread's condition variables in conjunction with a thread-safe FIFO queue. Events are used internally to ensure asynchronous communications between threads.

7.3.2 The Pipeline building-blocks

Pipeline is built using three event-based building blocks:

- ***Event Notifier*** : the "**event notifier**" is a runnable object which is composed of a task and an output event. When executed, it runs the task, i.e its encapsulated code, then activate its event through calling the "**notify**" method. The first stage of the pipeline is an event notifier since it produces data without consuming.
- ***Event Relay***: the "**event relay**" is composed of a task and two events. It waits for an input event then execute its associated task and finally signal the output event. In a pipeline, intermediate stages are event relays since they both consume and produce data.

- ***Event Listener***: the "**event listener**" is the tail of the pipeline. It is composed from and input event and a task. It waits for an incoming event then execute its task.

7.3.3 Pipeline Construction

Pipeline is constructed from these three building-blocks. It is composed from an event-notifier which is the head of the pipeline, one or several event-relay which are the intermediate stages and an event-listener which play the role the last stage. Tasks encapsulating stages code as defined by the programmer are embedded in these building-blocks, additionally, events are transparently created to link the different stages and allow them to communicate.

FIGURE 7.4 describe the execution of a three-stage pipeline that processes a total of two items of interest (datum). We distinguish three different operations: "**run**", "**wait**" and "**notify**". An event is introduced between each couple of stages to enable them to communicate. We note that "stage 1" performs only "**run**" and "**notify**" operations since it is an "**Event Notifier**". "stage 2" executes "**wait**", "run" then "**notify**" operations while the last stage "stage 3" performs "**wait**" then "run" operation for each datum. We note also that the "**run**" operation is executed once for each datum by each stage. The index of the datum is passed as an argument to the "**run**" operation of the task which will update in turn the first argument of the function, object method or lambda expression before calling them to tell them which data they should process.

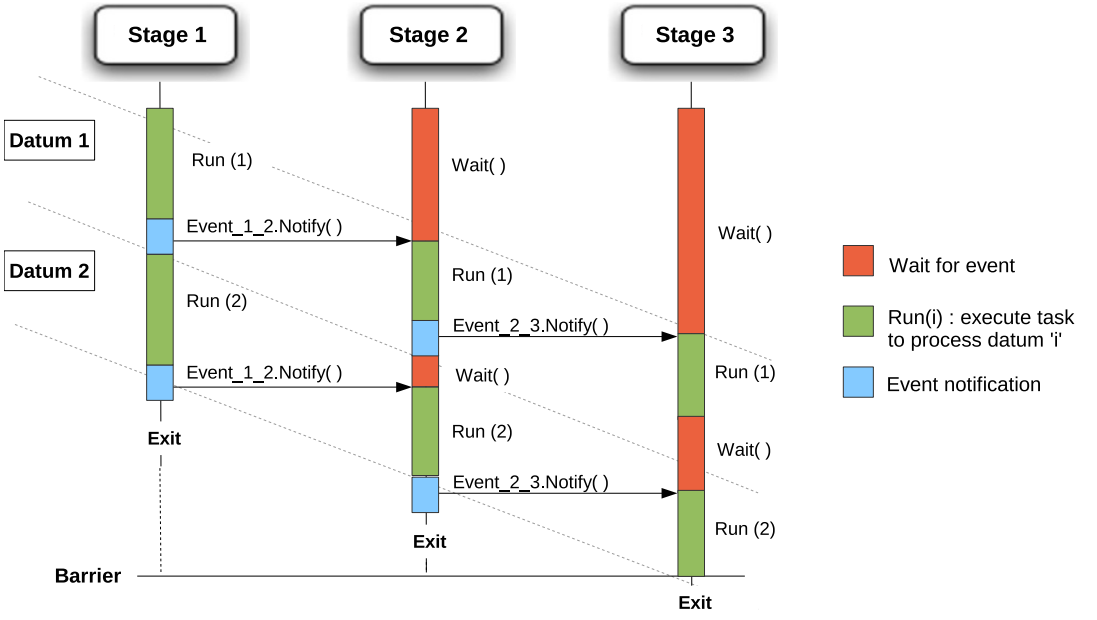


Figure 7.4: Execution of three stage pipeline which process two item of interest.

The previous sequence diagram shows that each stage execute repetitively two or three operation for each datum. We note these operations sequence as work unit. Consequently we identify three different types of works as shown in FIGURE 7.5. We define a common interface for them which we name work and implement a "**perform**" method which executes the encapsulated operations sequence. This abstraction offers a flexible mean to execute these works using different schedulers implementing different scheduling strategies and techniques as detailed in the next section. This flexible design decouples clearly workload partitioning from workload scheduling.

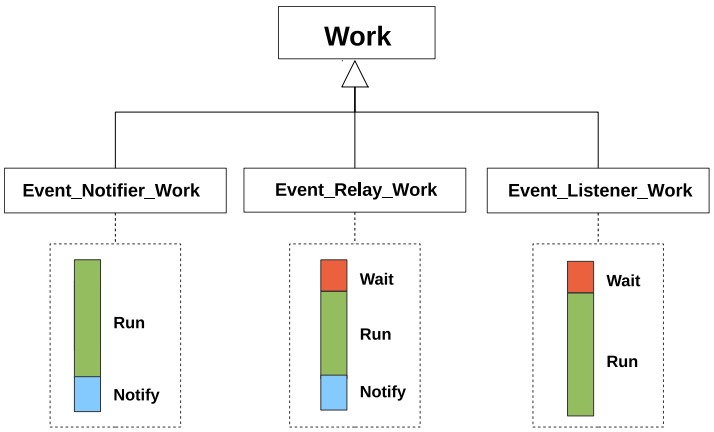


Figure 7.5: Three types of unit work: first stage execute event notifier work, intermediate stages execute event relay work and last stage execute event listener work

7.4 Run-Time System

Task scheduling is critical for execution efficiency. An efficient scheduling policy should consider two key parameters : good load-balancing and efficient cache use. The runtime system of XPU is responsible of executing the different workloads described in the previous section. These workload can be executed using different scheduling strategies. In this section we detail three of them : Thread-based scheduling, Cache-aware scheduling and Load-balanced scheduling.

We note that the programming interface as well as the internal pipeline architecture remains unchanged and common for all the schedulers since scheduling, pipeline architecture and pipeline parallelism expression are decoupled by design.

7.4.1 Thread-based scheduling

When executed the pipeline has to execute stages in separate asynchronous threads. Synchronization is performed naturally through event notifications between threads. Each thread hosts either an "**Event Notifier**", An "**Event Listener**" or an "**Event**

Relay" that will loop as long as data is available. FIGURE 7.6 gives an overview of this scheduling technique: when "**pipeline.run()**" is invoked, a thread is created for each stage of the pipeline, for N data, it will execute repetitively N times its workload wile taking care of updating the data index each time. When done thread simply exits. The main thread waits for the last stage's thread to exit.

The main advantage of this trivial approach is that it is relatively easy to implement. This implementation depends strongly on OS-level scheduling which will perform task-processor mapping. The number of created threads growing proportionally to the number of pipeline stages is one of the major limitations of this approach. A deep pipeline with big number of stages may generates too much threads for the available computing resources.

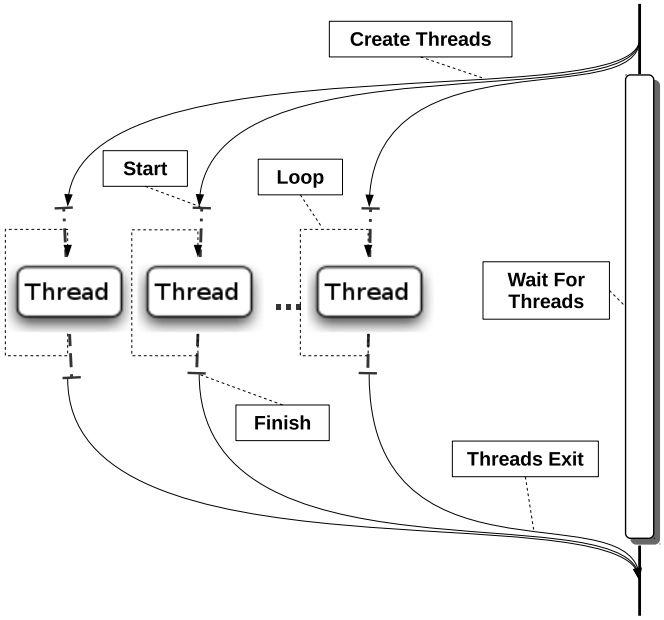


Figure 7.6: Thread-Based Run-Time System : A thread is created for each pipeline stage, the main thread wait for these threads to process all available data.

7.4.2 Load-balanced scheduling

As depicted in FIGURE 7.7, load-balanced scheduling relies on a persistent pool of threads or "*Workers*" to execute works. For each N-stage pipeline execution on M datum, $M \ X \ N$ works are generated or more precisely M "**Event Notifier**" works, M "**Event Listener**" works and $(N\text{-}2) \ X \ M$ "**Event Relay**" works. These works are submitted into a common work queue for all workers. Idle workers pick available works from this shared queue. Work ordering or serialization of dependent works is done naturally through the embedded events in each work.

This implementation provides obviously better load-balancing. This have been confirmed by experiments on the image processing application which we present in the next section. However, this approach suffers from poor spatial and temporal data locality in processor caches resulting in inefficient reuse of data in addition to potentially costly one-to many communication and "*arbitrary*" task-processor mapping.

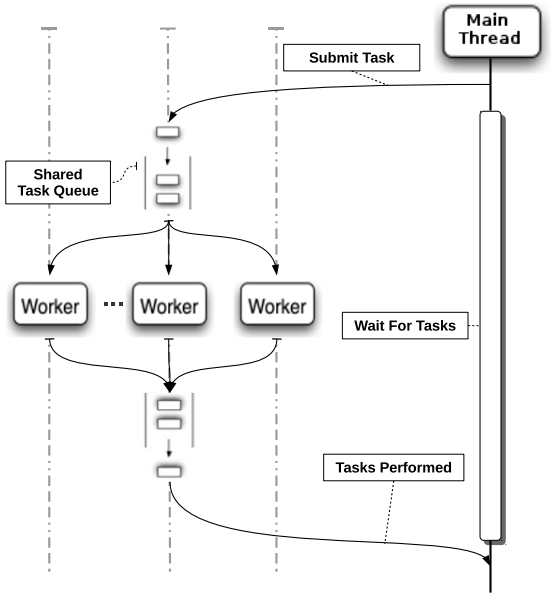


Figure 7.7: In the load-balanced scheduler implementation, task are submitted to a pool of *workers* through shared *work queue*, idle *workers* pickup the tasks from the work queue. Therefore, the load is constantly and dynamically balanced.

7.4.3 Cache-aware scheduling

FIGURE 7.8 gives an overview of cache-aware scheduling technique. Efficient use of processor caches is the primary concern in this scheduling techniques. At the opposite of the previous implementation, each persistent worker has a private work queue. As explained previously a **N**-stages pipeline processing **M** datum generates **N** x **M** works. In this configuration these works will be distributed to the available work queue quasi-fairly following a round robin scheme for each stage. This scheduling technique relies on an implicit task-data dependency information implicitly available in the pipeline execution pattern: each data will be processed serially by all pipeline stages. In this implementation we execute all stages works implying datum "**i**" on the same processor "**p**" since datum "**i**" is potentially available in high-level cache of processor "**p**". Our experiments show that this scheduling scheme reduces cache miss rate and improves data locality in caches.

This approach performs efficient use of caches in on hand by improving datum locality in caches and on the other hand by reducing communication cost through one-to-one communication when submitting works. However, simple round-robin workload distribution does not perform good load-balancing especially in the case of highly unbalanced stages. This may lead to shadow gains of efficient cache use by poor workload load-balancing.

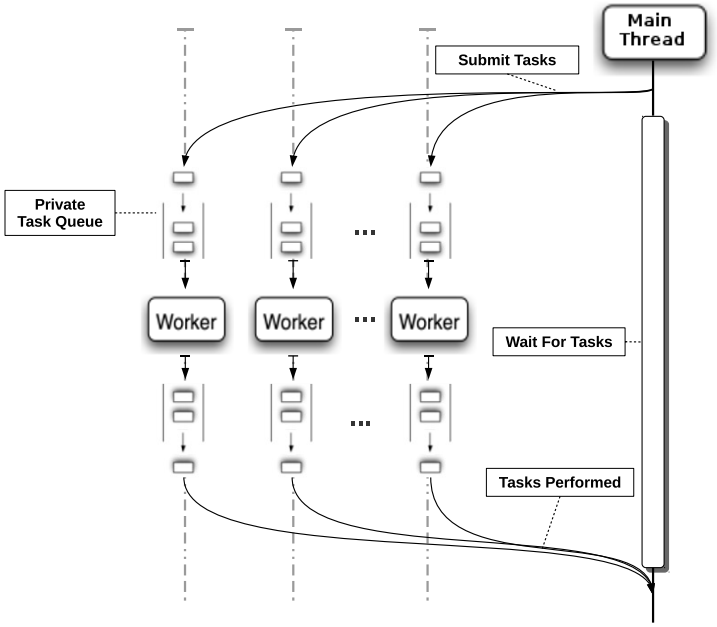


Figure 7.8: Worker Pool-based run-time system with private Work Queue.

7.5 Application: Image Processing

7.5.1 Algorithm Description

In this section we describe an edge detection algorithm named "**Adaptive Threshold Edge Detection**" that was developed for the OpenIllusionist project [Par07][Parb] specifically to help with detecting fiducials (reference points) in live video [PRZ05]. This algorithm appears to be faster and better suited to perform the task on live video stream than heavier algorithms such as the Canny Edge Detector [Can87]. The original algorithm was designed by Prof. John Robinson at the University of York and then modified and optimized for the OpenIllusionist project. We propose to implement the algorithm as a pipeline in order to speed up real-time processing of continuous images stream.

In order to evaluate our approach in terms of programmability (parallelism expressiveness) and performance in comparison with TBB, we reuse the original sequential

implementation of the algorithm, which was implemented as a Gimp plug-in written in C [Para], to implement two parallel version based on the pipeline execution pattern of XPU and TBB. FIGURE 7.9 gives an overview of the edge detection algorithm as it is implemented in the sequential target program. We note that many algorithms from many fields of applications implying stream processing such as signal processing, multimedia application and data compression can be parallelized the same way.

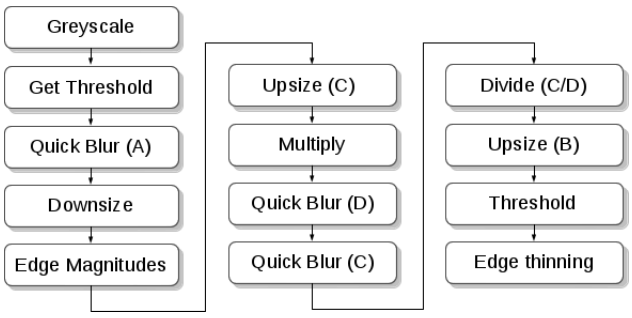


Figure 7.9: The Adaptive Threshold Edge detection Algorithm.

In the following sections, we try to evaluate our approach through this study case. Our experiment shows that the XPU-based version provides a significant performance improvement over the original implementation, performs higher frame rate than the TBB version at the cost of lesser parallelism-related extra-code, allows the reuse of the sequential code without significant alteration contrary to the TBB version.

7.5.2 Evaluation Methodology

Tension between performance and programmability are unavoidable in parallel programming model design [CCZ07b]: programming model may emphasize expressiveness and ease of use over performance, and therefore, sacrifices some execution efficiency to gain programmability and improve programmer productivity. In the other side, another parallel programming model may be designed to deliver high performance at the cost of significant programmability loss making parallel programming difficult, time-consuming and error-prone. Programming models make various performance-programmability trade-off.

In order to evaluate our programming model, we consider both of these two aspects. Performance can be evaluated by measuring execution time and throughput or performed frame rate in our study case. However, programmability or productivity are less easier to evaluate, many evaluation methodologies have been proposed to "quantify" them [TTTn+09]. One of these method relies on measuring the similarities and the differences between sequential and parallel code. Comparing the sequential and the parallelized version of the same application gives a clear view of, in one hand, the reused/altered sequential code and in the other hand, the introduced parallelism-related

extra-code required by the programming model to express parallelism.

In our study case, we consider the existing sequential application implementing the adaptive edge detection algorithm, we parallelize it using TBB and XPU then we measure the required amount of parallelism-related extra-code as well as the reused and altered sequential code. We use the "**CLOC**" [Sol] tool to count lines of code in each version ignoring all blank lines and comments. A common formatting tool limits the influence of coding style on lines count by enforcing a common C++ coding format for all versions. Moreover, to perform finer measures of similarities and differences at the character-level and not only at the line-level, we use the Levenshtein algorithm to compute the distance between sequential code string and each of the two parallel versions.

7.5.3 Programmability

The target image processing algorithm performs its task in thirteen processing step by applying several filters and transformations to the image. These operations are implemented as a set of functions. We reuse these functions without any modification or optimization of their respective code. In the case of XPU, a task is defined for each stage. In the case of TBB, a "filter" class is implemented for each stage. Pipelines are created as seen previously. FIGURE 7.10 shows a comparison between the lines count of the original sequential implementation (418 Lines) and the needed extra-code for parallelization in the XPU (65 additional lines) and TBB parallel version (351 additional lines). We note that XPU version requires 80% less lines of code to express parallelism than TBB.

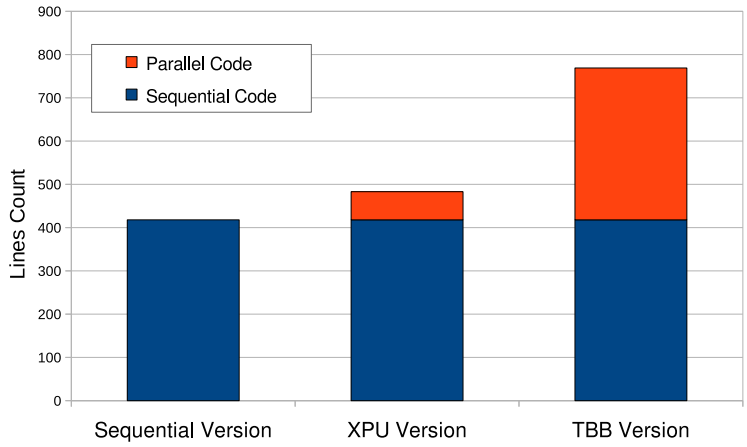


Figure 7.10: Lines count of the original sequential version and the required parallelism-related extra-code in the XPU and TBB version.

As depicted in FIGURE 7.11, by applying the *Levenshtein* algorithm, the obtained distance between XPU version and the sequential version is 3.3k while the distance between TBB version and the same sequential version is 9.6k. Therefore XPU version is about three time closer to sequential version than TBB version.

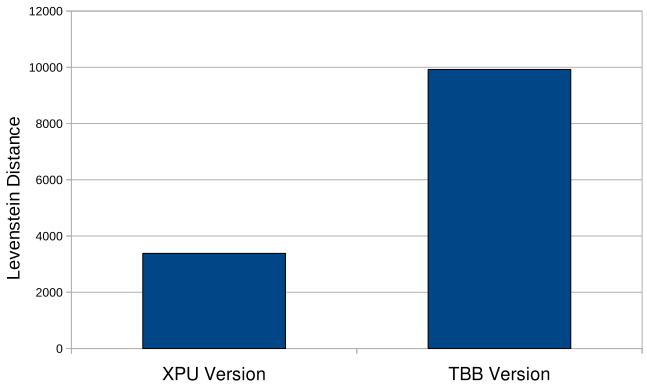


Figure 7.11: Levenshtein distance (arbitrary unit) between original sequential version and respectively XPU and TBB parallelized versions.

7.6 Performance

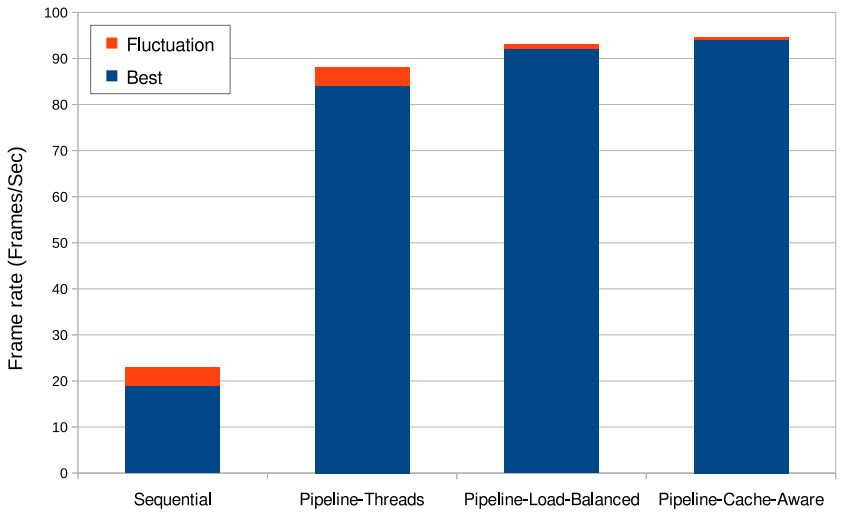


Figure 7.12: Performed frame rate (fps) by sequential and pipeline processing with different scheduling techniques on an 8 threads processor (highest best).

FIGURE 7.12 shows a comparison between these three scheduler implementation. Load-balanced and cache aware scheduling achieve slightly better frame rate than the thread-based implementation, in addition, we note also that the achieved frame rate of the later suffer from more fluctuation and is less predictable. We compared the load-balanced version of XPU with the TBB one. FIGURES 7.13 and 7.14 depict obtained results respectively on an eight-thread processor and a 16 threads bi-processor platform. We notice that XPU version performs about 20% faster than TBB and achieves about 5 times speedup of the original sequential version.

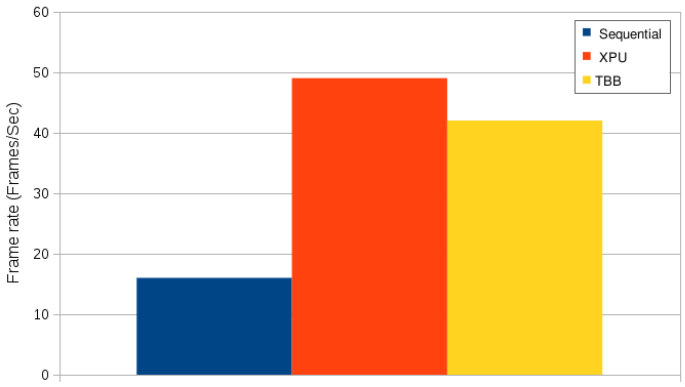


Figure 7.13: Average achieved frames rate (frame/sec) on an 8 threads processor (Intel Core i7 Q720).

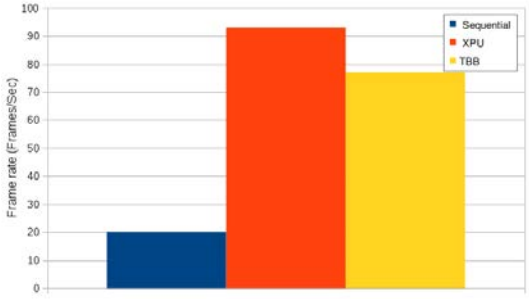


Figure 7.14: Average achieved frame rate (frames/sec) on a 16 hardware threads platform (SMP with 2 x Intel Xeon E5620 at 2.4 GHz).

7.7 Limitations and Future Works

At the time of writing these lines, the current implementation of the XPU pipeline pattern has a set of limitations which can be addressed in future works.

7.7.1 Skeleton Nesting in Pipeline

The XPU pipeline construct can be nested can be nested as a task group inside the hierarchical task group graph (HTGG), however it does not allow "direct" nesting of other patterns as pipeline stages: only simple tasks can be used as pipeline stages. The various patterns can be nested "indirectly" through using them inside the different tasks that constitute the stages of the pipeline. Allowing direct nesting would make the API more elegant and can preserve the "hierarchical" property of the HTGG and allows deeper nesting of execution patterns.

7.7.2 Data Management

In order to use the XPU pipeline skeleton to process data items, the processed data must be indexable so the stage (task) can retrieve the data simply by using its index which is continuously updated by the pipeline runtime at each stage execution. Consequently, the programmer is responsible of organizing the data into indexable structures such as arrays or vectors, the XPU pipeline skeleton guarantee only the coherent execution ordering of the different stage and does not provide any data management container such as FIFO data queue. Providing such data container can be useful for the programmer especially when processing dynamic data such as streams in real-time signal processing applications.

7.7.3 Out-Of-Order Stage Execution

Pipeline stage are often unbalanced in term of workload, as a results, some stages may execute faster than the others. Consequently multiple data can be accumulated at the input of slower stages. The XPU pipeline skeleton enforce execution ordering of the different stages for the same data but also the processing order of different data inside the same stage. So the parallelism between the different stages is exploited while the sequential execution is enforced inside the same stage. In some applications, ordering data processing inside the same stage cannot be required. Allowing out-of-order processing inside the same stage to exploit more parallelism can be an interesting feature in these particular cases. This execution mode is available in the Intel TBB pipeline pattern. It can be easily implemented in the XPU pipeline in future XPU versions.

7.8 Conclusion

The pipeline execution pattern can be a useful parallelism multiplier in many applications displaying strong serial dependencies between tasks. However this pattern exposes

two major challenges: pipeline parallelism expression complexity and pipeline execution efficiency. In this paper, we focused on these two aspects and we showed how we can ease pipeline parallelism expression without sacrificing execution efficiency. We used the traditional C++ programming language without any extension and we presented the implementation details of the pipeline execution pattern of XPU.

The image processing study case is a typical stream processing application which outlined both the programmability and performance aspect and showed that despite its emphasis on programmability, XPU provides good performances in comparison with TBB programming model. Many algorithms can be parallelized the same way as the presented study case. For instance, we are working on a pipelined implementation of another application from the digital signal processing field.

The presented pipeline design decouples clearly parallelism specification i.e. programming interface from internal pipeline design and task scheduling. This design allowed us to experiment different scheduling strategies without modifying the displayed programming interface. Good load-balancing and efficient cache use are critical for pipeline execution. The basic scheduler implementations presented in this paper focused on one of these two parameters and not both of them simultaneously. The task stealing can be investigated to design smarter scheduling policy which provides better trade-off between efficient cache use and good load-balancing.

As we will see in the automatic parallelization chapter, the pipeline pattern can be seen as a particular case of a more general producer-consumer pattern which is the super scalar task graph. We show in the next chapter how the building blocks of the pipeline can be adapted and reused to implement the super-scalar task graph pattern. XPU provides a set of skeleton to enable programmer to specify parallelism explicitly, in the next chapter we present the FATMA framewok which allows automatic parallelization of a sequential program provided that the program is represented as a sequence of tasks.

Part III

Automatic Parallelization : FATMA

8

Automatic Parallelization

In the previous chapters, we introduced XPU which is a pure C++ parallel programming model which aims to ease explicit parallelism expression. XPU provides a collection of hierarchical parallel constructs to enable the programmers to express several types of parallelism including task, data and pipeline parallelism at different levels of granularity.

While easing parallelism expression and improving programmer productivity in many application domains, XPU may be less suited to some applications exacerbating complex dependencies between very large number of tasks making explicit parallelism expression very hard even with the simplest programming interface (API). For instance, many tiled linear algebra algorithms may generate thousands of tasks with extremely complex dependencies. For example, Figure 8.1 illustrates the complexity of the task dependency graph (DAG) of the tiled Cholesky factorization algorithm that generates 35 tasks when operating on 5x5 tiles. The number of tasks grows considerably as the number of tiles become larger: as depicted in Figure 8.2, the algorithm generates 220 tasks when the target matrix is decomposed into 10x10 tiles and 1540 tasks for a 20x20 tiles configuration... As the number of tasks grows, the complexity of the DAG increases significantly. In such cases, despite the simplicity of the "parallel/sequential" programming interface provided by XPU, it is very hard to express the parallelism of such complex task graphs without the help of automatic parallelization or code generation tools...

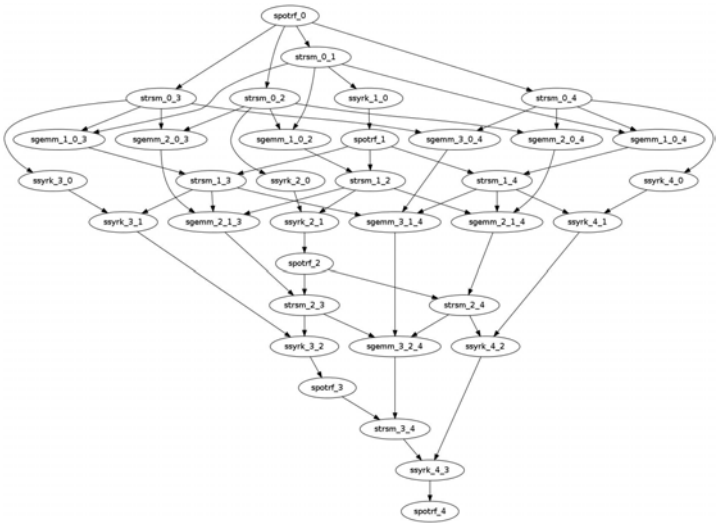


Figure 8.1: The task dependency graph of the tiled Cholesky factorization algorithm for a 5x5 tiles decomposition.

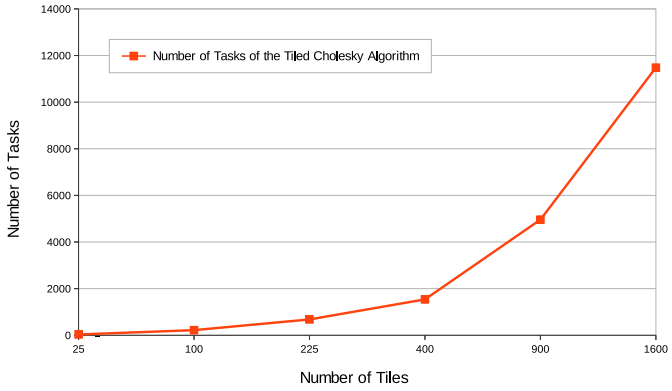


Figure 8.2: In many tiled linear algebra algorithm, the number of tasks grows as $O(n^3)$ with the number of tiles. While generating 35 tasks for 15 tiles configuration, the Cholesky factorization algorithm generates 4960 tasks for 900 tiles decomposition.

In addition to the parallelism extraction complexity, the XPU programming model expresses hierarchical task parallelism through the *fork-join* execution model which is based on spawning tasks and synchronizing them on synchronization points or *Barriers*. This execution model does not provide efficient execution of universal Task Graphs (DAG) since the used barriers may introduces unnecessary idle times that correspond to "*false dependencies*". In order to execute efficiently such Task Graphs, Tasks should be

executed in a super-scalar fashion to eliminate these idles times and false dependencies. Similarly to XPU, programming models such as Cilk Plus [Rob13], TBB [Phe08] or OpenMP [DM98] use this execution model and thus cannot execute efficiently such task graphs.

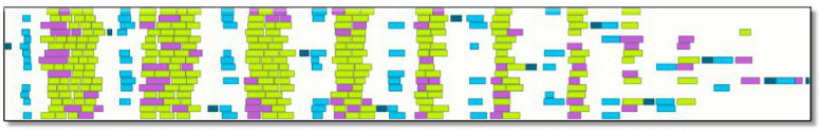


Figure 8.3: The fork-join execution model used by XPU, TBB, Cilk Plus, OpenMP... generates unnecessary idles times when executing certain Task Graphs (DAG).

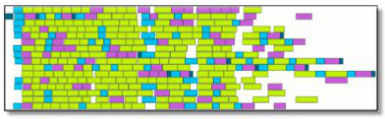


Figure 8.4: The super-scalar execution model used by FATMA, SMPSS or Quark executes asynchronously the tasks and use event-based peer-to-peer synchronization model between dependent task. This allows FATMA to eliminate unnecessary idles times when executing Task Graphs (DAG).

The Fast Multicore Application (FATMA) framework addresses the limitations of XPU by extending its capabilities and providing automatic parallelization feature that allows the programmer to parallelize transparently a large sequence of tasks by generating the corresponding task dependency graph and scheduling parallel tasks according to that graph on the available processors. The graph-driven execution is performed in a super-scalar fashion in order to reduce idle times and maximizing throughput while preserving data coherency and task dependencies.

Contrary to many programming models such as StarPU [ATNW11] and SMPSs [BHL+09] that introduce language extensions or compiler directives and require specialized compilers and tools, FATMA use exclusively pure standard C++ language and requires nothing more than a C++ compiler to be used. Instead of using compilation techniques to parallelize task sequences, FATMA relies on an intelligent run-time system that parallelize the sequences of task dynamically at run-time.

In this chapter, we present briefly the FATMA programming model then we describe examples of practical linear algebra applications and finally we show that despite its high abstraction and improved productivity, FATMA is capable to deliver comparable performances to lower level programming models.

8.1 Related Works

Several programming models addressed successfully the task-level automatic parallelization either at compile-time or at run-time or both of them. For instance, Quark, SMPSs and StarPU are prominent examples of programming models which provide automatic task dependency extraction and dynamic scheduling capabilities. FATMA falls in this category of programming models.

8.1.1 StarPU

StarPU [ATNW11] [Cou13] is being developed by INRIA Bordeau, LaBRI. It was introduced as C library designed to exploit heterogeneous parallel platforms with both general-purpose processors and specialized processors often referred to as "accelerators". While addressed run-time task scheduling issue, being a C library, StarPU display a poor, verbose and error-prone programming interface. In 2011, efforts has been started to improve its programmability by introducing new C language constructs and extending the GCC compiler suite [Cou13].

8.1.2 SMPSs

SMPSs [BHL+09] is a parallel programming framework developed at the Barcelona Supercomputer Center. It is based on a dynamic scheduler implementation which aims to automating exploitation of functional parallelism of a sequential program on multicore and symmetric multiprocessor platforms [Cen]. In SMPSs applications, task and their respective dependencies are specified through OpenMP-like preprocessor directives which annotate functions.

SMPSs relies on a C99 source-to-source compiler and a driver named "*smpss-cc*" which use the specified dependencies to generate a parallel source code. The generated object files are then linked to the SMPSs run-time library which implements a dynamic task scheduler. This scheduler relies on a dynamically generated task-dependency graph to schedule concurrent tasks.

8.1.3 Quark

Quark [Yar12] is a run-time environment for dynamic scheduling of applications that consists of precedence-constrained on multicore and multi-socket shared-memory systems. It is developed at Innovative Computing Lab, University of Tennessee. Quark is implemented as a C library which offer a programming interface for defining tasks and their dependencies. A dynamic scheduler is responsible of asynchronous scheduling of these tasks without violating their dependencies. Quark has been designed to meet the specific needs of PLASMA [KLY+14] which is a tiled linear algebra library.

8.2 FATMA Contributions

Since it exploits exclusively the traditional C++ language, FATMA does not require any specialized compiler such as SMPSs and StarPU C extension. Unlike Quark and StarPU's standard C API which expose a relatively verbose and error-prone programming interface, FATMA offers an intuitive and light-weight programming interface designed for high programmability. For instance, in most tiled linear algebra applications, a negligible modifications are required to make the transition between sequential and parallel code: as illustrated in Figure 8.5 the legacy calls to the basic BLAS and LAPACK primitives are simply encapsulated in tasks, the sequence of these tasks forms the program which is then parallelized by generating the corresponding task dependency graph a.k.a the DAG using one single line of code. Finally, a second line allows the programmer to execute concurrently the parallel tasks and exploit the available parallelism.

```
1  for (int j=0; j<nb; j++) {
2      for (int k=0; k<j; k++)
3          for (int i=j+1; i<nb; i++)
4              sgemm(a(i,k), a(j,k), a(i,j));
5      for (int i=0; i<j; i++)
6          ssyrk(a(j,i),a(j,j));
7      spotrf(a(j,j));
8      for (int i=j+1; i<nb; i++)
9          strsm(a(j,j), a(i,j));
10 }
```

Sequential Execution

```
1  program p;
2  for (int j=0; j<nb; j++) {
3      for (int k=0; k<j; k++)
4          for (int i=j+1; i<nb; i++)
5              p.add(task(sgemm, a(i,k), a(j,k), a(i,j)));
6      for (int i=0; i<j; i++)
7          p.add(task(ssyrk,a(j,i),a(j,j)));
8      p.add(task(spotrf,a(j,j)));
9      for (int i=j+1; i<nb; i++)
10         p.add(task(strsm,a(j,j), a(i,j)));
11 }
12 p.build();
13 p.run();
```

FATMA Parallel Code

Figure 8.5: The FATMA code introduce minor parallelism-related extra-code (highlighted in red) while preserving the legacy code from any significant alteration.

As we will see in the next sections, performing the same parallelization using Quark requires much more extra-code. Task creation requires explicit specification of all input and output data. These parameters are packed then unpacked within a wrapper function that calls the original function when the task is inserted. At each task insertion, dependencies of the inserted task are resolved before the execution. This online dependency analysis introduces an execution overhead. In FATMA, task dependency analysis is performed "*offline*", e.g. before the tasks execution. FATMA dependency analysis generates synchronization points that are associated to the resolved dependencies. At the execution stage, FATMA execute asynchronously the tasks and use the synchronization points to preserve data coherency and avoid violating tasks dependencies. Thus no dependency analysis-related overhead is introduced at the execution stage.

8.3 The FATMA Parallelization Process

The FATMA parallelization process consists in three steps which are:

1. **Task Sequence Specification**: the programmer specifies a set of tasks in their natural sequential order. As we have seen in the "Task Definition" chapter, task can encapsulate different pieces of code including functions and object methods.

2. **Task Dependency Graph (DAG) Construction**: The run-time system analyzes the data accesses of each task and build the Task Dependency Graph.

3. **DAG-Driven Asynchronous Task Execution**: The run-time system uses the DAG to drive the asynchronous execution of the tasks.

Figure 8.6 gives an overview of the FATMA parallelization process. Each of these steps is represented conceptually as an intermediate program representation and its associated FATMA code.

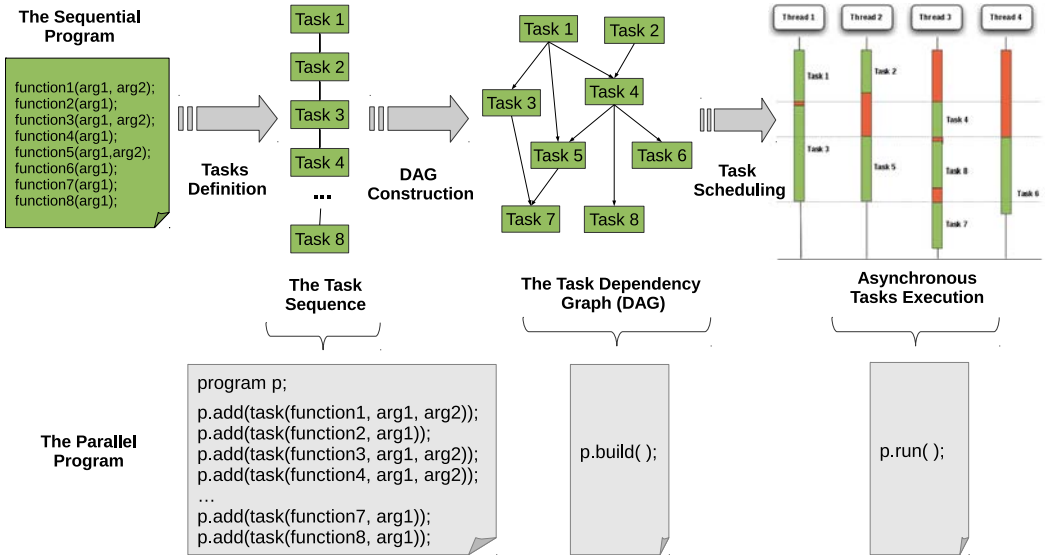


Figure 8.6: The automatic parallelization process start from a specification of a sequence of task to generate transparently the task dependency graph then use it to schedule asynchronously these tasks.

In the next sections, we present the FATMA programming interface then the implementation details of the run-time system. In order to illustrate the use of FATMA in a practical cases, we present the parallelization of the popular tiled Cholesky factorization algorithm. Analogously to the Cholesky algorithm, the LU and QR factorization and similar tiled linear algebra algorithms can be parallelized the same way.

8.4 Tiled Cholesky Factorization

In order to illustrate the use of FATMA in a practical cases, we consider the tiled Cholesky factorization algorithm. The Cholesky factorization (or Cholesky decomposition) is widely used in many scientific applications when resolving numerical solution of the linear equation Ax = b, where A is positive definite and symmetric. The Cholesky algorithm decomposes a square matrix in the form of "A = L.L^T" where L is a lower

triangular matrix with positive diagonal elements.

The cholesky factorization is implemented in LAPACK [ABB+92] by the *xpotrf* routine where the 'x' specifies the precision arithmetic. In the next paragraphs, for the sake of simplicity we omit the 'x' from the LAPACK and BLAS routine names. In tiled linear algebra algorithms, each matrix is partitioned into a set of tiles or blocks as depicted in Figure 8.7. The tiled version of the Cholesky algorithm operates on the tiles of the target matrix using basic routines of LAPACK and BLAS which are *syrk*, *potf2*, *gemm* and *trsm*.

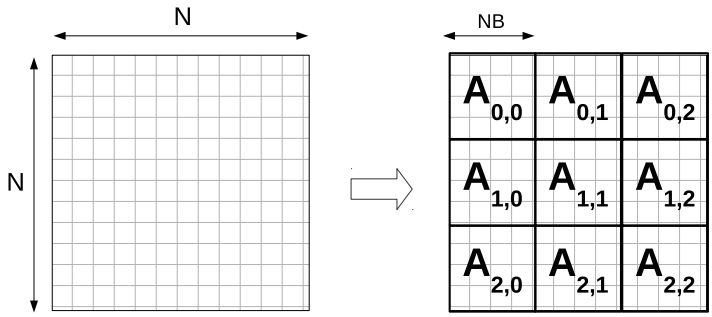


Figure 8.7: Tiled representation of N x N real matrix, in this example the matrix is decomposed into 3 x 3 tiles, each tile is a NB x NB submatrix.

Algorithm 1 gives an overview of the tiled Cholesky factorization. In this algorithm "$A(i,j)_a$" denotes the matrix tile at the position *i,j*, the *a* term specifies the access type to the tile which can be (r:read) or (rw: read-write).

Algorithm 1 Tiled Cholesky Factorization

```
for j ← 0 to N − 1 do
    for k ← 0 to j − 1 do
        for i ← 0 to j − 1 do
            sgemm(A[i,k]_r, A[j,k]_r, A[i,j]_rw)
        end
    end
    for i ← 0 to j − 1 do
        ssyrk(A[j,i]_r, A[j,j]_rw)
    end
    spotrf(A[j,j]_rw)
    for i ← j + 1 to N − 1 do
        strsm(A[j,j]_rw, A[i,j]_r)
    end
end
```

8.5 The FATMA Programming Interface

8.5.1 The FATMA Program Structure

The FATMA programming interface is based on a structure named "**program**" which hold a list of task or more precisely a sequence of tasks that programmer want to parallelize. Tasks are added to this structure in their sequential order by invoking the "**program::add(task t)**" method. Once the program is filled with the list of tasks, the programmer calls the "**program::build()**" to build the tasks dependency graph (DAG). After building this task graph, programmer can invoke the "**program::run()**" method to execute the parallel program. We note that if the programmer need to add more tasks, he should recall the "**build()**" method before executing the new program. Otherwise, programmer can re-execute the program as many times as he want without rebuilding it. Listing 8.1 shows a typical FATMA program.

```
1  program p;
2
3  // task definition
4  task t1(function1, arg1, arg2);
5  task t2(function2, arg1);
6
7  // adding tasks in their sequential order
8  p.add(t1);
9  p.add(t2);
10
11 // building the task dependency graph
12 p.build();
13
14 // executing the parallel program
15 p.run();
```

Listing 8.1: The FATMA Programming Interface.

8.5.2 Application to Tiled Cholesky Factorization

Lets consider the tiled Cholesky factorization as an example of application to illustrate the use of the FATMA programming interface. Listening 8.2 shows the original sequential code of the tiled Cholesky factorization. Implementation details of the "sgemm_tile", "ssyrk_tile", "spotrf_tile" and "strsm_tile" routines are not shown since they are simply wrappers for the original standard LAPACK and BLAS routines.

```
1
2  #define A(i,j) A[i*DIM+j] // simplifying access to the tiles
3
4  int cholesky(float * A[DIM][DIM], int NB)
5  {
6    for (int j= 0; j<DIM; j++)
```

```
7   {
8     for (int k= 0; k<j; k++)
9       for (int i = j+1; i < DIM; i++)
10        sgemm_tile( A(i,k), A(j,k), A(i,j), NB);
11    for (int i = 0; i < j; i++)
12      ssyrk_tile( A(j,i), A(j,j), NB);
13    spotrf_tile( A(j,j), NB);
14    for (int i = j+1; i < DIM; i++)
15      strsm_tile( A(j,j), A(i,j), NB);
16  }
17 }
```

Listing 8.2: Sequential code of the tiled Cholesky factorization.

We note that LAPACK and BLAS routines are implemented in many libraries such as Intel MKL, GotoBLAS, OpenBLAS , Atlas... etc. These libraries are tuned to offer high performances on multicore and multiprocessor platforms in single-threaded or multi-threaded mode. We note that we use always the single-threaded implementations of these routines when using them with FATMA to preserve data locality in caches and allows FATMA to control the task-processor mapping through cache-aware and load-balanced scheduling strategies. We note that there is also other implementations on BLAS that support GPU computing such as cuBLAS. The later implementation use CUDA from NVIDIA to support GPU whele preserving the standard function prototypes of the BLAS routines allowing easy switching between CPU or GPU implementation.

```
1
2  int cholesky(float * A[DIM][DIM], int NB)
3  {
4    program p; // the structure that will hold the sequence of tasks
5    for (int j= 0; j<DIM; j++)
6    {
7      for (int k= 0; k<j; k++)
8        for (int i = j+1; i < DIM; i++)
9          p.add(sgemm_tile, A(i,k), A(j,k), A(i,j), NB);
10     for (int i = 0; i < j; i++)
11       p.add(ssyrk_tile, A(j,i), A(j,j), NB);
12     p.add(spotrf_tile, A(j,j), NB);
13     for (int i = j+1; i < DIM; i++)
14       p.add(strsm_tile, A(j,j), A(i,j), NB);
15   }
16   p.build(); // build the task dependency graph (DAG)
17   p.run();   // use the DAG to execute asynchronously the tasks
18 }
```

Listing 8.3: Parallelization of the tiled Cholesky factorization using FATMA.

8.5.3 Programmability Comparison

In this section, we try to compare FATMA to QUARK and SMPSs in term of programmability. We present briefly the programming interfaces of QUARK and SMPSs then compare them to FATMA by quantifying the required effort to make the transition between a common sequential application to its parallel version using the different parallel programming models.

We defines a set of metrics to quantify this programming effort. QUARK and SMPSs are chosen for this comparison because they represent two different approaches of programming interface implementations: QUARK uses standard C language and does not require any specialized compiler while SMPSs extends C language with OpenMP-like directives to ease parallelism expression but introduce a dedicated compiler or preprocessor named "*smpss-cc*". The first versions of StarPU were based on a C library similarly to QUARK, more recent implementations of StarPU extends the GCC compiler suite and thus is compiler-based similarly to SMPSs. Consequently, we have not included StarPU in our comparison.

8.5.3.1 QUARK

The QUARK programming model defines an API for task insertion that allows the programmer to schedule tasks for execution.The QUARK run-time system is responsible of analyzing tasks dependencies and resolving data hazard conflicts to determine whether a task can be executed or should wait for another task to terminate before it can be executed.

Unlike FATMA, QUARK does not allow direct reuse of functions as Tasks. Instead, when reusing a function as a task, the programmer needs to write two extra functions : a wrapper function and a task insertion function that are associated to the actual target function. In order to illustrate the use of the QUARK task insertion API, lets consider the "dgemm" BLAS routine that we want to use as a task. The signature of this routine is shown in Listing 8.4.

```
1 void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
2                  const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
3                  const int K, const double alpha, const double *A,
4                  const int lda, const double *B, const int ldb,
5                  const double beta, double *C, const int ldc);
```

Listing 8.4: The protoype of the standard CBLAS routine "dgemm" for double percision matrix multiplication.

A Task Scheduling

In the task insertion function, the programmer is responsible of specifying explicitly the access type (*INPUT*, *OUTPUT*, *INOUT* or *VALUE*) to each parameter of the target

function. The later information is used by the run-time to analyze dependencies when the task is inserted. We note that "*CORE_ dgemm_ quark*" is a the task wrapper that encapsulate the target function. The wrapper function is called by the run-time system when executing the task. We note also that the "*QUARK_ Insert_ Task*" invokation is performed within a separate function for the sake of code clarity and can be invoked directly inside the main function.

```
1  // Quark task insertion function
2  void QUARK_CORE_dgemm ( Quark* quark , QuarkTaskFlags * taskflags, int order, int transA, in
3                          int m, int n, int k, int nb, double alpha, const double *A, int lda,
4                          const double *B , int ldb, double beta, double *C , int ldc)
5  {
6     QUARK_Insert_Task( quark , CORE_dgemm_quark , taskflags ,
7           sizeof ( enum CBLAS_ORDER  ), &order
8           sizeof ( enum CBLAS_TRANSPOSE ), &transA, VALUE,
9           sizeof ( enum CBLAS_TRANSPOSE ), &transB, VALUE,
10          sizeof ( int ), &m, VALUE,
11          sizeof ( int ), &n, VALUE,
12          sizeof ( int ), &k ,VALUE,
13          sizeof ( double ), &alpha, VALUE,
14          sizeof ( double )*nb*nb, A, INPUT,
15          sizeof ( int ), &lda, VALUE,
16          sizeof ( double ) *nb*nb, B, INPUT,
17          sizeof ( int ), &ldb , VALUE, sizeof ( double ), &beta, VALUE,
18          sizeof ( double )*nb*nb, C, INOUT ,
19          sizeof ( int ), &ldc, VALUE, 0);
20 }
```

Listing 8.5: QUARK Task insertion function

B Task Specification

The wrapper function play the role of a proxy that encapsulates the actual target function. The wrapper is called by the run-time when executing the corresponding task. This wrapper is required because the run-time cannot call directly the original function since a preliminary step is required before the call: when specifying access types to the different parameters of the function, these parameters are packed in a specific QUARK structure and thus parameters should be extracted ("*unpacked*") from that structure to be able to pass these parameters as arguments to the original function when calling it. Listing 8.6 shows a QUARK wrapper for the "dgemm" BLAS function.

```
1  // wrapper routine for the original C BLAS routine
2  void CORE_dgemm_quark(Quark * quark)
3  {
4     int order, int transA, transB, m, n, k, lda, ldb, ldc;
```

```
5     double alpha, beta;
6     double *A, *B, *C;
7
8     quark_unpack_args14(quark, order, transA, transB, m, n, k, alpha,
9                         A, lda, B, ldb, beta, C, ldc);
10    cblas_dgemm(order, transA , transB, m, n, k, (alpha), A, lda, B, ldb,
11                        (beta), C, ldc);
12 }
```

Listing 8.6: Example of QUARK wrapper function that unpack arguments before calling the actual target function

```
1  int main()
2  {
3    // set values of parameters: m, n, k, lda . . .
4    QUARK_CORE_dgemm(quark, taskflags, transA, transB, m, n, k, alpha, A,
5                     lda, B, ldb, beta, C, ldc);
6  }
```

Listing 8.7: In order to insert a task in the task queue for execution, the task insertion function is called.

The QUARK programming interface is less verbose than the PThreads API since the paradigm of Task insertion relieve the programmer from handling low-level threads details. Yet, the QUARK API still requires substantial rewriting of the target application code by introducing significant amount of extra-code and modifying the legacy sequential code. This make the QUARK API more verbose than the SMPSs programming interface that offer more compact programming interface. However, while SMPSS API relies on a C/C++ language extension with "pragmas" and a custom compiler, QUARK is implemented using standard C language that offers limited expressiveness and programmability but still offers the advantage of portability across a wide variety of C compilers and systems.

8.5.3.2 SMPSs

The SMPSs programming model address the lack of expressiveness of standard C language by extending it with OpenMP-like preprocessor directives. These directives are used to define tasks and specifies their data dependencies. SMPSs directive are also used for tasks scheduling .

A Task Specification

As shown in Listing 8.8, SMPSs tasks are conceived in a form of a function without return value, i.e. a procedure [Cen11]. Functions are converted into tasks by annotating its definition or declaration with an SMPSs directive (**pragma**). The programmer specifies through this directive the input and the output of the target function.

```
#pragma css task input(A[NB][NB], B[NB][NB], NB) inout(C[NB][NB])
void smpss_dgemm(double *A, double *B, double *C,
                 int NB, double alpha, double beta)
{
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasTrans,
                NB, NB, NB, alpha, A, NB, B, NB, beta, C, NB);
}
```

Listing 8.8: Specification of a task and its input/output using SMPSS.

B Task Scheduling

Once tasks have been specified, they can be used in the main program simply by calling the annotated functions. When a function is called within a block surrounded by the directives "# **pragma css start**" and "# **pragma css finish**", the corresponding task is scheduled by the SMPSs runtime. We note that the later block can be used only once in the program. Listing 8.9 shows how to schedule the previously defined "smpss_sgemm" task in an SMPSs program.

```

void main()
{
   #pragma css start

   smpss_dgemm_tile(A1, B1, C1, NB, alpha, beta);
   smpss_dgemm_tile(A2, B2, C2, NB, alpha, beta);

   #pragma css finish
}
```

Listing 8.9: Task in SMPSS program are scheduled by invoking the corresponding annotated function. The two scheduled tasks in the example do not expose any depdencies between them and thus will be excuted concurrently by the SMPSs run-time.

8.5.4 Comparison

We compare FATMA, QUARK and SMPSs in term of programmability by determining the required effort to make the transition between a sequential application to its parallel version. This transition consists mainly in the specification of tasks and their data dependencies then scheduling them for execution by the different run-time systems.

8.5.4.1 Methodology

Programmability and productivity evaluation is not an easy task, many evaluation methodologies have been proposed to "quantify" them [TTTn+09]. One of these meth-

ods relies on measuring the similarities and the differences between sequential and parallel code. Comparing the sequential and the parallelized version of the same application gives a clear view of, in one hand, the reused/altered sequential code and in the other hand, the introduced parallelism-related extra-code required by the programming model to express parallelism.

In order to compare FATMA, QUARK and SMPSs, we consider two examples:

- A simple program using a single task (previous "dgemm" task example).
- The tiled cholesky factorization example.

In both case studies, we consider the sequential code and we parallelize it using FATMA, QUARK and SMPSs then we measure the required amount of parallelism-related extra-code as well as the reused and altered sequential code. We use the "**CLOC**" [Sol] tool to count lines of code in each version ignoring all blank lines and comments. A common formatting tool limits the influence of coding style on lines count by enforcing a common C++ coding format for all versions.

8.5.4.2 Results

A Simple Program

In the case of the single-task program, we measure the required parallelism-related extra-code (line count) and the altered/reused legacy sequential code (line count). The fact that there is a single task to define then execute allows us to measure the programming overhead relatively to a single task. The used task is the BLAS "dgemm" function that we have seen in the few previous paragraph. The line length is limited to 100 characters to avoid carriage return and line splitting interference with our measures. Table 8.1 summarize our measures.

	QUARK	SMPSs	FATMA
Extra-Code (lines)	24	6	3
Altered Code (lines)	1	1	1
Reused Code (lines)	N-1	N-1	N-1
Standard C/C++ Compiler	Yes	No	Yes

Table 8.1: Programmability comparison between QUARK, SMPS and FATMA in the case of single task program: lines count of the required parallelism-related extra-code and altered/reused legacy sequential code.

In the case of QUARK, about 14 lines of extra-code are introduced by the task insertion routine, 8 lines are introduced by the wrapper definition and 2 lines are required for run-time initialization and stopping. One single line is altered since the original function call is replaced by a call to the task insertion routine. Te remaining code can be reused without alteration. Finally, we note that QUARK use standard C language

and thus can be used with standard C/C++ compilers.

In the case of SMPSs, generally a single line directive is required to specify the task, but often a function wrapper (3-4 lines) is required since SMPSs requires that the target function has a void return value which is not usually the case in legacy code, especially when reusing external library such as the C BLAS library in our case. Another two extra-lines are required for the start/finish directive. Altered/reused lines of legacy code is similar to QUARK. Contrary to QUARK, a specialized compiler is required to compile SMPSs programs.

In the case of FATMA, there is no need for wrappers to encapsulate functions when using them as tasks, any function, including external library routines such BLAS ones, can be used as a task. Task definition and insertion are performed within a single line of code. This line is the one that replace the traditional function call in the legacy code. The remaining code is reused without any alteration. FATMA is a pure C++ library and does not require any specialized compiler or tools excepting a standard C++ compiler.

B The Tiled Cholesky Application

In the previous example, we measured the programming overhead for a single-task program, in this section, we apply the same methodology for a more realistic application which is the tiled Cholesky factorization. Many others tiled linear algebra algorithms such as LU and QR factorization or linear system resolution should give similar results and proportions.

	QUARK	SMPSs	FATMA
Extra-Code (lines)	86	18	3
Altered Code (lines)	4	4	4
Reused Code (lines)	N-4	N-4	N-4
Standard C/C++ Compiler	Yes	No	Yes

Table 8.2: Programmability comparison between QUARK, SMPS and FATMA in the case of the tiled cholesky factorization program.

8.5.4.3 Conclusion

As we have seen in the previous comparisons, FATMA requires lesser programming effort to specify and schedule tasks than SMPSs and QUARK. SMPSs offers higher programmability than QUARK but requires a custom compiler. Contrary to SMPSs, FATMA is a based on standard C++ language and does not require any specialized compiler or tool apart a standard C++ compiler. Therefore, FATMA offers a better programmability-portability tradeoff in comparison to QUARK and SMPSs.

8.6 Dynamic Task Dependency Graph Construction

In the FATMA programming model, the programmer specifies a sequence of tasks and the FATMA run-time is responsible of parallelizing them through constructing the task dependency graph (DAG) that drive the asynchronous tasks execution. In order to parallelize a sequence of tasks, dependencies between these task are analyzed to identify the consumer-producer relationships between the different tasks. Tasks dependencies analysis at run-time is a complex process that can introduce a significant execution overhead even when performed "*offline*" (before tasks execution). For instance, task dependency analysis using the Polyhedral Model [BHRS08] can be too time consuming to be performed at run-time, therefore, it is often implemented at the compilation stage within compilation tool chains such as the PLUTO tool [BHRS08]. In order to reduce as much as possible this potential overhead, we use a custom task dependency analysis algorithm designed particularly to meet the need of FATMA. Our algorithm exploits exclusively the task-data dependency information which are extracted transparently by FATMA at run-time. The algorithm consists mainly in two stages which are:

- Virtual Task Execution Tracing (VTET)
- Task Execution Back-Tracing (TEBT)

In the first stage the sequential task execution is simulated while their execution traces and more particularly their access to data (read/write) are collected. The second stage exploits the collected traces to build the task dependency graph (DAG). Once the DAG is fully constructed, it can be used to drive the concurrent execution of tasks. In the next paragraphs, we describe the VTET and TEBT algorithms.

8.6.1 Virtual Task Execution Tracing

When executed, a task accesses data in read, write or read-write mode. In addition to these elementary access types, if we consider the sequential execution order of a set of tasks, we can identify five types of data accesses that generate producer-consumer relationships between the different tasks:

- Initial Read (IR) : the task A read a data which has never been accessed.
- Initial Write (IW) : the task A read a data which has never been accessed.
- Read After Read (RAR) : the task A read a data which was read by a previous task B. In this case A and B are independent and can be executed simultaneously without violating of any dependency.
- Read After Write (RAW) : the task read a data which was written by a previous task. In this case, A is dependent on B and cannot be executed until B execution is terminated.

- Write After Write (WAW) : the task write a data which was written by a previous task. Similarly to the previous case, A is dependent on B and cannot be executed until B execution is terminated.

FATMA tasks are very similar to XPU tasks and inherit the transparent task data dependency detection feature that analyzes the arguments of each task to extract its data dependency including the consumed and produced data. Hence, for each task, we know which data is accessed by the task, in addition, we know the access type to that data (*Read* or *Read-Write*). This feature is extensively discussed in the Chapter 2 ("Task Definition") of this thesis.

Given a sequence of tasks, the Virtual Task Execution Tracing (VTET) aims to simulate the execution of a these tasks in their sequential order while recording their traces including the read and write accesses to data and thus the precedence of theses accesses (RAR, RAW and WAW). We note that task traces recording does not requires executing tasks, instead, their execution is just simulated, e.g. data accesses are recorded without executing any computation. Thus, thanks to this abstraction of the task execution details, the VTET execution overhead depends only on the task count (O(n)) and not on the workloads of the different tasks.

In order to illustrate the VTET process, lets consider a simple application composed of a sequence of 6 tasks (namely T_1, T_2, T_3, T_4, T_5 and T_6) that perform basic computations on a set of data (namely '**a**', '**b**', '**c**', '**d**', '**e**' and '**f**'). The computations consist in simple additions, each addition takes two data as input and write the result into a data output. Tasks are implemented as follow:

1. Task T_1 : a = b + c
2. Task T_2 : d = c + e
3. Task T_3 : e = a + b
4. Task T_4 : f = c + d
5. Task T_5 : c = f + e
6. Task T_6 : c = b + b

These FATMA tasks can be created using the function "*add*" as shown in Listing 8.10. The first parameter of the function "*add*" is the *output* of the addition (*non-constant pointer*) and the two remaining parameters are the *inputs* (*constant pointers*).

```
1 void add(int * res, const int * x, const int * y)
2 {
3   *res = *x + *y;
4 }
5
```

```
6  void main()
7  {
8    int a,b,c,d,e,f;
9    task t1(add, &a, &b, &c); // T1 definition : a = b + c
10   task t2(add, &d, &c, &e); // T2 definition : d = c + e
11   // ...
12 }
```

Listing 8.10: Task definition using the function "*add*".

If we apply the VTET process to this sequence of tasks, we obtains the traces depicted in Figure 8.8. These traces specifies the precedence of task accesses to data and allow us to retrieve the dependencies or the consumer-producer relationship between them. Task dependency extraction is performed in the next step which is Task Execution Back-Tracing (TEBT).

	T1 a=b+c	**T2** d=c+e	**T3** e=a+b	**T4** f=c+d	**T5** c=f+e	**T6** c=b+b
a	*IW*		*RAW*			
b	*IR*		*RAR*			*RAR*
c	*IR*			*RAR*	*WAR*	*WAW*
d		*IW*		*RAW*		
e		*IR*	*WAR*		*RAW*	
f				*IW*	*RAW*	

Figure 8.8: Result of the Application of the Virtual Task Execution Tracing process to the sequence of tasks.

Figure 8.8 represents the result of the application of the VTET algorithm. The result is represented as a table of traces. Each column of the table corresponds to a task and store its accesses to data. Each row of the table correspond to a data and row contains the successive access traces of the different tasks to that data.

8.6.2 Task Execution Back-Tracing and DAG Construction

In order to be able to construct the task dependency graph, the dependencies between the different tasks are deduced from the simulated execution traces that we obtained after the application of the VTET algorithm. The exploitation of these traces is performed using a new algorithm named the Task Execution Back-Tracing (TEBT).

8.6.3 Task Execution Back-Tracing

Task Execution Back-Tracing (TEBT) exploits the recorded traces of the simulated sequential execution to extract dependencies between tasks. Considering the sequential execution order of the tasks and the consumed and produced data, dependencies between tasks are extracted from the traces following the next two rules:

1. A reader task is dependent on the last writer if the later exists.

2. A writer task is dependent on the last writer and the last reader if the later exists.

Based on these rules, Figure 8.9 shows how the simulated execution traces are exploited to extract dependencies between tasks in our example. Dependencies between tasks are represented through the red directed edges between columns that correspond to the different tasks. We can observe that concurrent read accesses to data does not introduce dependencies between the readers and thus allows parallel execution of multiple readers. We note that the IW and IR traces does not introduce any dependencies on predecessor tasks since the data is accessed for the first time.

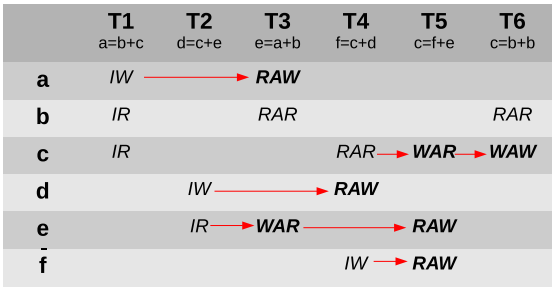


Figure 8.9: Task Execution Back-Tracing exploits the simulated execution traces to extract dependencies between tasks represented by the red direct edges.

8.6.4 DAG Construction

Once dependencies are extracted using TEBT, they are stored into a structure named the task dependency graph. Since such graph is directed and acyclic, it is often referred in the literature as the Directed Acyclic Graph (DAG). The nodes of the graph represent the tasks while the directed edges between these nodes represent the dependencies between tasks. We note the direction of the edges indicates the ordering of dependent tasks.

Many run-time environment such as QUARK [Yar12] resolve dependencies between tasks dynamically when tasks are queued and thus the DAG is implicitly defined and never explicitly constructed at run-time [Yar12]. In this dynamic scheduling approach, dependency analysis is performed "*online*" when executing tasks and thus introduces an execution overhead when "inserting" tasks.

In our approach, both tasks dependencies analysis and DAG construction are performed "*offline*" before the actual task execution. This allows us to avoid introducing any significant execution overhead when executing the task and enable us to perform many optimizations before the tasks execution such as cache-aware task-processor mapping to improve data locality and reduce communication overhead. Also, its possible to

perform DAG transformations such as node fusion to adjust task granularity.

The FATMA approach allows us to perform static scheduling, dynamic scheduling or hybrid scheduling since task-processor mapping can be specified explicitly before tasks execution or dynamically when executing tasks to allow better load balancing.

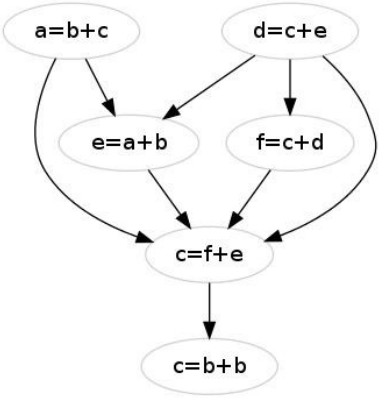


Figure 8.10: The Directed Acyclic Graph (DAG) corresponding to the example program.

In order to execute the parallelized program without compromising data coherency, the constructed DAG is used to drive the execution of the tasks. The tasks are executed in a super-scalar fashion to maximize throughput and reduce idle times.

8.7 Static and Dynamic DAG-Driven Scheduling

8.7.1 Execution Infrastructure

The simplest way to execute a Task within a peer thread is to create a new thread and setup the target Task as the callback of the that thread. While easy to implement, this approach became quickly ineffective as the number of tasks grows. In addition to the thread creation overhead and the frequent context switching, a significant communication overhead can be introduced. this approach is also limited by the available physical resources in term of available threads and memory... Our experiments on the initial implementation of DAG-driven scheduler using threads showed a significant performance degradation when the tasks number grows.

The Task abstraction allows us to delegate task execution to a set persistent generic workers (PGW) instead of creating threads. The PGW consist mainly in pool of threads that are able to execute tasks independently of the execution pattern in with the tasks running. The PGW is a generic execution infrastructure that provide a good hardware abstraction. In addition, it offer a finer control of data locality since tasks can be scheduled on a specific core or processor without need of low-level thread configuration

or pining. We use the PGW infrastructure to execute the tasks or the "nodes" of the task graphes generated by FATMA.

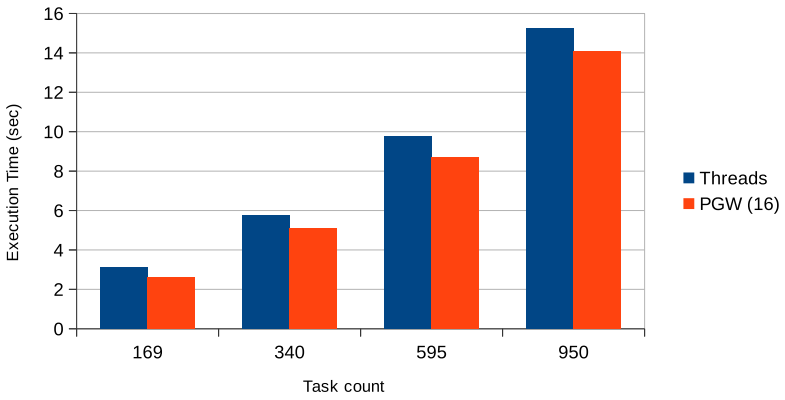


Figure 8.11: Comparison between the parallel implementation of the tiled linear system solver (dgesv) using the Thread-based infrastructure and the Intel Core i7

8.7.2 Tasks Synchronization an Coordination

As we detailed in the first Chapter of this thesis, the Super Scalar Task Graph (SSTG) can be seen as a generalization of the Pipeline pattern and also as a composition of both the pipeline pattern and the fork-join pattern. While the pipeline pattern specifies a linear consumer-producer relationship between consecutive pair of stages where each single consumer depends on a single producer, the STTG can specify more generalized consumer-producer relationship where multiple consumers can depend on a multiple producer. For this reason, we reused the building blocks of the XPU Pipeline skeleton to implement the SSTG construct.

The execution infrastructure of the Pipeline skeleton is based on a worker pool and thus is similar to the PGW infrastructure which we described in the previous paragraph. The reader can refer to the "Pipeline Parallelism" chapter to find more information about the implementation of the XPU Pipeline skeleton.

The XPU Pipeline implementation are based on three building blocks which are the *Event Notifier*, the *Event Relay* and the *Event Listener*. These three types of "*nodes*" corresponds respectively to a pure producer, a consumer/producer and a pure consumer. Each of these "nodes" consist in a structure that encapsulates a task and an input and/or an output *Events*. *Events* corresponds to the dependencies between the nodes and provide the communication between the nodes. Figure 8.12 gives an overview of the Event-based implementation of a simple 4-stages Pipeline.

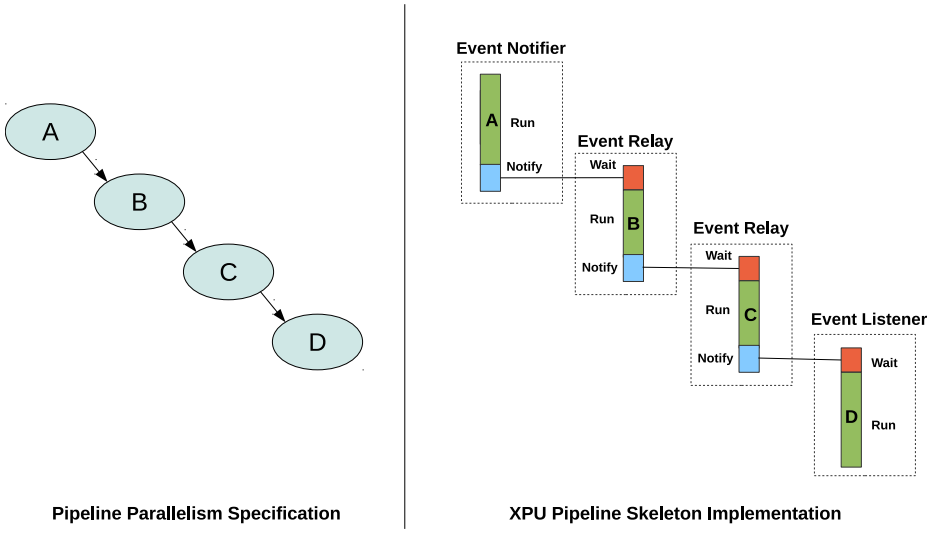


Figure 8.12: Event-based implementation of the XPU Pipeline skeleton.

In the Pipeline pattern, the "*nodes*" are linked to each others to form a consumer-producer chain that starts with a pure producer and terminates with a pure consumer while all the intermediate nodes are both consumers and producers at the same time. Thus, consecutive "*nodes*" are connected by pairs (one-to-one).

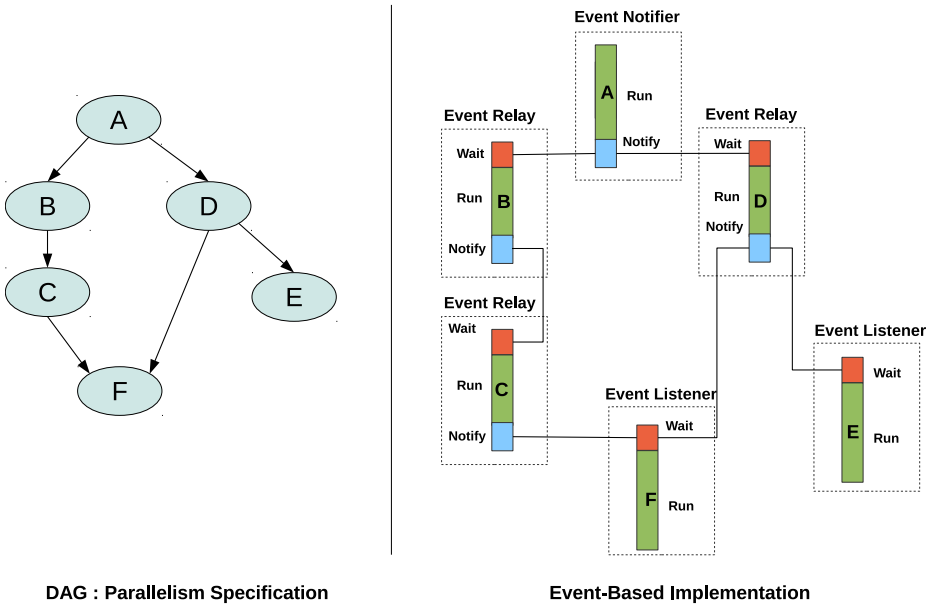


Figure 8.13: Event-based implementation of the FATMA super scalar task graph.

The SSTG has a similar consumer-producer structure except that a "*node*" can be connected to more than one node. Therefore, we derived new implementation of the *Event Notifier*, *Event Relay* and the *Event Listener* implementation to support multiple output/input *Events* and allows a node to connect to multiple nodes to form a graph instead of a simple chain. This new implementation meet the specific needs of the FATMA run-time and allows the asynchronous execution of the Super Scalar Task Graph. Figure 8.13 gives an overview of the event-based implementation of the FATMA Task Graph.

8.8 Application to Tiled Linear Algebra Algorithms

In order to evaluate the performance of the FATMA framework, we use it to parallelize two algorithms: the first is the tiled Cholesky factorization algorithm and the second is the tiled solver for linear equation systems. While being mainly a "static" scheduler, the FATMA run-time uses dynamic scheduling techniques to offer a better tradeoff between cache use efficiency and load-balancing. In the first application, we compare the performance of FATMA to two dynamic schedulers which are Quark and SMPSs. In the second application, we compare the performance of the FATMA-based implementation to a statically scheduled implementation from the PLASMA library.

8.8.1 Tiled Cholesky Factorization

8.8.1.1 Experimental Setup

We use FATMA, Quark and SMPSs to parallelize the tiled Cholesky factorization. We use a basic sequential implementation as a common base code. This common implementation is taken from the program examples which are provided in the SMPSs v2.4 package. The used tile size is 200 for all the configurations.

The target cholesky program uses basic LAPACK and BLAS routines and thus requires the LAPACK and the BLAS libraries. We use the Intel MKL library version 10.3 which implement the BLAS and LAPACK routines. We use the Intel Compiler "*icc*" version 12.0.5 when compiling each of the three parallel implementations. We note that we configure the SMPSs "*smpss-cc*" compiler to use the Intel Compiler when compiling the final code.

We execute the three parallel implementations of the Cholesky algorithm on an Intel Core i7 Q720 (8 hardware threads and 4 physical cores) platform. We note that the Turbo Boost and Hyperthreading technologies are both enabled.

8.8.1.2 Result

Figure 8.14 shows the achieved performances by the three parallel implementations. The FATMA framework takes advantage of the low execution overhead of its static

scheduler to achieve relatively higher performance than QUARK and SMPSs when the number of task is low. However, as the number of the tasks grows, we notice a relative performance degradation of the FATMA performances in comparison with Quark and SMPSs. This can be explained by the domination of static scheduling in the current FATMA scheduling strategy. Static scheduling improves spatial and temporal data locality and allows effective cache use but suffers from poor load balancing due to limited dynamic scheduling decisions. Being dynamic schedulers, Quark and SMPSs provide more efficient load-balancing and achieve better performances in the configurations where the task number is relatively large.

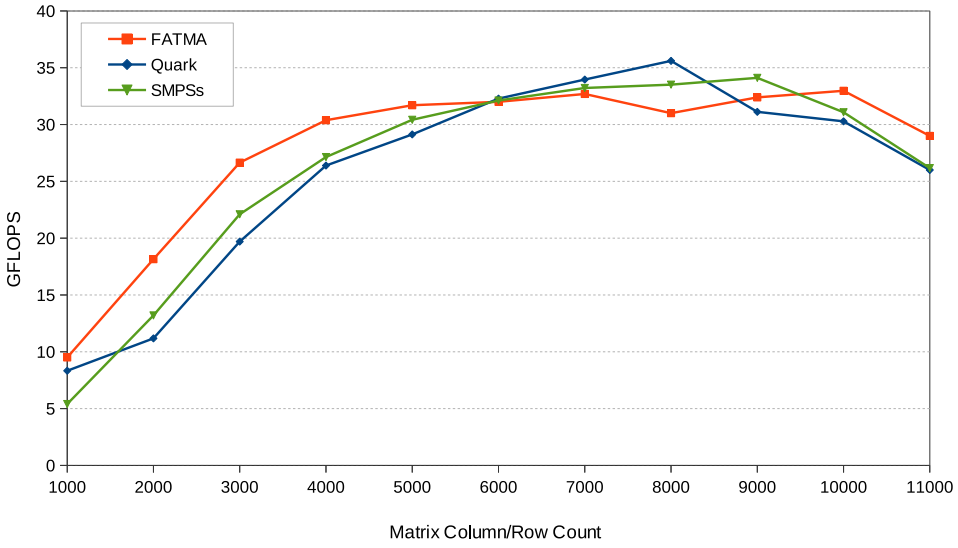


Figure 8.14: Comparison between FATMA, QUARK and SMPSs implementations of the tiled Cholesky factorization on and 8 Threads Intel Core i7 Q720 processor.

8.8.2 Tiled Linear System Solver (DGESV)

The tiled DGESV routine solve the linear system of equations "$A.X = B$" where A is a square matrix, the columns of matrix B are individual right-hand sides and the columns of X are the corresponding solutions. This computations are performed in two steps: the matrix A is factorized using the LU decomposition with partial pivoting and row interchanges method, then the factored form of A is used to solve the system.

8.8.2.1 Experimental Setup

We use the same software setup as the first application. We implement the tiled DGESV algorithm using FATMA and we compare our parallel implementation to the PLASMA implementation of the DGESV kernel. We use the PLASMA v2.0 which uses a static

scheduler and not the Quark dynamic scheduler). We use the Intel MKL implementation of BLAS and LAPACK routines for both FATMA and PLASMA implementations. We execute the two implementations on an SMP platform with two Intel Xeon E5620 at 2.4 GHz (16 hardware threads and 8 physical cores).

8.8.2.2 Discussion and Result

The PLASMA library uses a static scheduler which make static scheduling decisions and do not perform any task dependency analysis at run-time, instead, the asynchronous parallel algorithm is implemented "manually". The use of static scheduling techniques in the PLASMA library eliminates the need for run-time task dependency-analysis and thus eliminate the execution overhead related to such dynamic analysis. The efficiency of static scheduling in PLASMA came at the cost of the implementation complexity which make its code hard to read and maintain. This increase the difficulty of implementing new and innovative algorithms and this is one of the major motivation for implementing dynamic scheduler such as Quark or SMPSs.

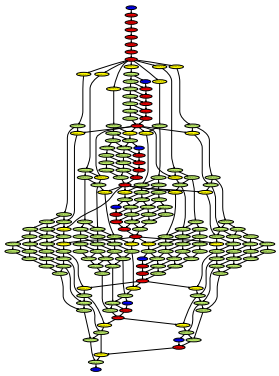


Figure 8.15: LU Factorization

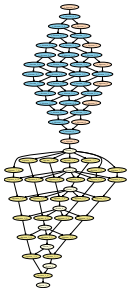


Figure 8.16: Linear Solver

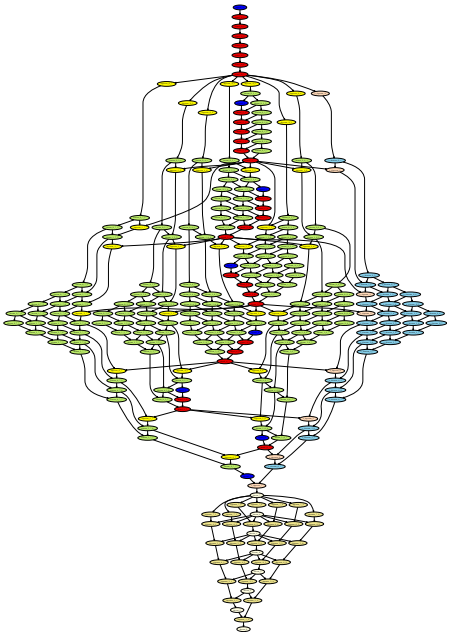


Figure 8.17: Merged Algorithms

While allowing effective use of processor caches through improving data locality, PLASMA's static scheduler exposes certain limitations when used in advanced algorithms such as the DGESV algorithm. For instance, the PLASMA implementation does not allow algorithms merging which can improve the performances in many algorithms that are composed of several parallel sub-problems such as the DGESV. As

we explained earlier, the DGESV algorithm performs an LU factorization then solves the linear system. PLASMA executes the parallel implementation of the tiled LU factorization (DGETRF) (cf. Figure 8.15) and wait for it to terminate before executing the parallel system solving (cf. Figure 8.16). FATMA overcomes this limitation by enabling algorithm merging and dynamic scheduling while still allowing static scheduling. FATMA merges the LU factorization (DGETRF) task graph with the linear solver task graph to form one single task graph (cf. Figure 8.17) which removes the barrier that the separate the two stages of DGESV algorithm and allows the extraction of more parallelism. In term of programmability, the FATMA code is much simpler and easier to read than the PLASMA code.

Figures 8.18 shows the achieved performances by the PLASMA and the FATMA implementations of the tiled DGESV algorithm. The PLASMA implementation take advantage of its static scheduler to achieve relatively better performances for small number of tasks. However, the FATMA implementation achieve higher performances for heavy workload with large number of tasks and take advantage of algorithm merging to extract more parallelism and keep the processors busy during the execution without any idle time between LU factorization and linear system solving. However, in the current implementation of FATMA, the scheduler perform very limited cache-aware scheduling, thus our implementation suffer from poor data locality which reduces the achieved performances and shadows the gained performances from algorithm merging. Consequently the performances of the FATMA and PLASMA versions remains relatively close for large problem sizes.

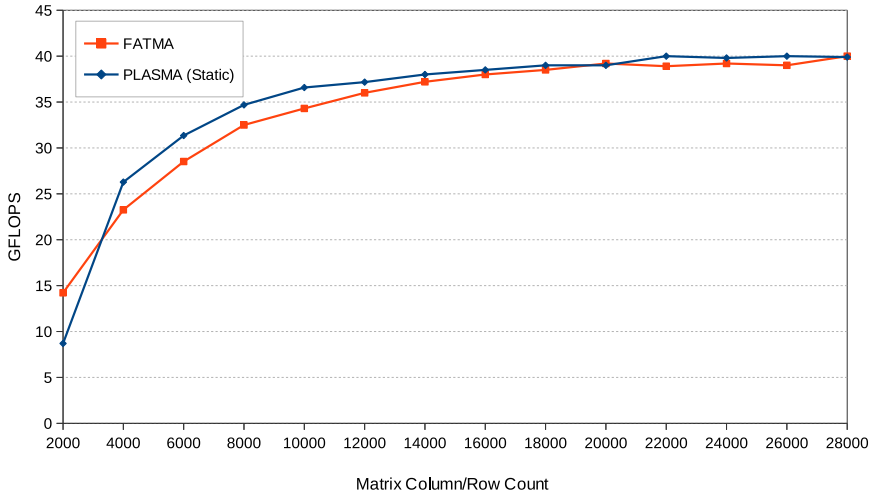


Figure 8.18: Comparison between FATMA and PLASMA (Static Scheduling) implementations of the tiled *dgesv* on an SMP platform with 2 x Intel Xeon E5620 at 2.4 GHz (16 Hardware Threads).

In the current version of FATMA, the extracted task data dependency information is exploited to build the task graph. In future versions of FATMA, we will exploit this information to perform smarter cache-aware scheduling to improve spatial and temporal data locality. Such scheduling techniques has been implemented successfully in the Quark dynamic scheduler which allows the programmer to assign "*locality flag*" to data items in order to improve data locality in processor caches [Yar12].

8.9 Conclusion

In this chapter we presented the FATMA framework which allows automatic task parallelism extraction under certain conditions. FATMA has been designed to offer high programmability and competitive performances. Contrary to many runtime systems such as SMPSs or StarPU which extended the C programming language and introduced specialized compilers, FATMA exploits exclusively the traditional C++ language to extract parallelism. Yet it provides an intuitive and light weight programming interface contrary to other libraries such as Quark which uses standard C programming languages but expose verbose programming interface. Our experiments on the parallelization of tiled linear algebra algorithms have shown that the FATMA programming interface allows the reuse of the legacy code almost without any alteration.

The FATMA programming model isolates parallelism extraction from task execution and thus eliminates online task-dependency analysis and allows both static and dynamic scheduling. The FATMA scheduler is at its very early development stages, yet it offers comparable performances to established state of the art schedulers such as SMPSs, Quark or StarPU. The FATMA scheduling techniques can be greatly improved to achieve higher performances.

The PLASMA linear algebra library uses static scheduling techniques which achieve high performance at the cost of the complexity of implementing new algorithms which limit its ability to prototype innovative algorithms[Jos13]. The FATMA framework can offer a good alternative to the PLASMA static scheduler since it is able to perform both static and dynamic scheduling while easing algorithm prototyping through its light-weight programming interface.

Part IV

Applications

9 Radar Signal Processing Application

General-purpose shared memory multicore architectures are becoming widely available. They are likely to stand as attractive alternatives to more specialized processing architectures such as FPGA and DSP-based platforms to perform real-time digital signal processing. In this chapter, we describe a parallel implementation of radar signal processing application. This study case shows how we can improve programmer productivity through easing parallel programming without sacrificing performances.

9.1 Algorithm Overview

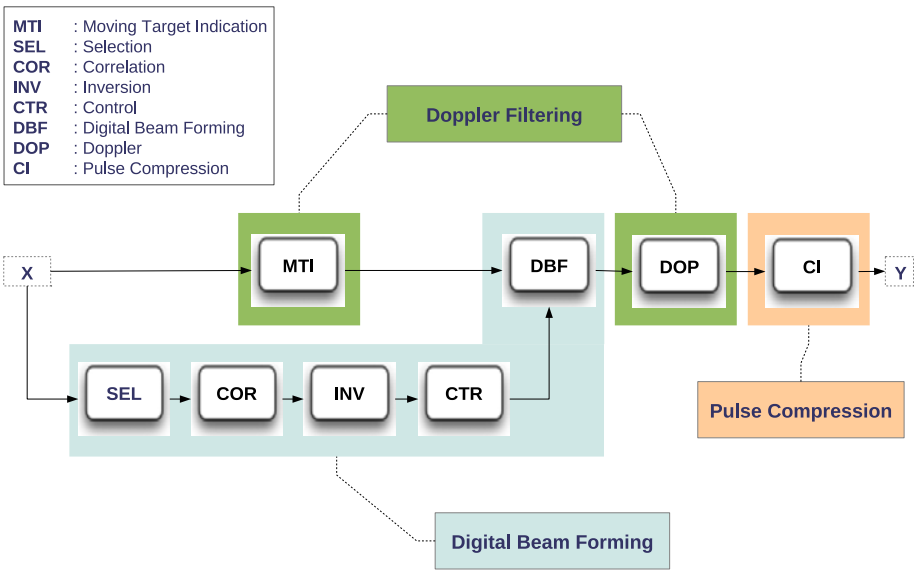


Figure 9.1: Overview of the Radar signal processing algorithm.

The target application is a radar signal processing algorithm which processes a digitized signal of a phased array radar system. Data volume grows significantly as enabled channels count grows. As shown in Figure 9.1, the algorithm perform its task in eight steps. These steps may be summarized in three major stages which are:

1. Digital Beam Forming (Green)
2. Doppler Filtering (Blue)
3. Pulse Compression (Red)

Echos are received as periodic bursts. For each burst, the received signal's samples feeds the signal processing chain which must perform all required operations before the next burst is received in order to meet real-time processing.

The first task "RCV" is responsible of data reception from the radar simulator and introduces thus negligible workload. The four following blocks: SEL (Selection), COR (Correlation), INV (Inversion) and CTR (Control) perform a correlation of the receiver channels and guide digital beam-forming. The MTI (Moving Target Indication) block allows the discrimination of moving targets against stationary clutter. DBF (Digital Beam Forming) forms beams using the processed input channels. Doppler processing is then performed using these beams by the DOP (Doppler) block and finally Pulse compression is completed by the last block PC (Pulse Compression).

Listing 9.1 show the basic sequential C++ implementation of the radar signal processing chain. The different blocks of the algorithm are implemented as simple functions that are called sequentially. This code is used as a reference code and has been parallelized at different granularity levels using several XPU constructs to express different types of parallelism. The next paragraphs details the progressive parallelization process and the achieved performance at each step of this process.

```
1  // functions implementation
2  void rcv(/* args */) { ... }
3  void sel(/* args */) { ... }
4  void cor(/* args */) { ... }
5  void inv(/* args */) { ... }
6  void ctr(/* args */) { ... }
7  void inv(/* args */) { ... }
8  void inv(/* args */) { ... }
9
10 // sequential radar signal
11 // processing algorithm
12
13 int main()
14 {
15   // serial processing of each burst
16   while (burst_available) {
17    rcv(/* args */);
18    sel(/* args */);
19    cor(/* args */);
20    inv(/* args */);
21    ctr(/* args */);
```

```
22      mti(/* args */);
23      dbf(/* args */);
24      dop(/* args */);
25      pc (/* args */);
26    }
27  }
```

Listing 9.1: Sequential Radar Signal Processing Implementation

The target Radar can be configured to use different number of input channels. Input data volume as well as computing load is proportional to the enabled channels count. While Figure 9.2 shows the theoretical computing load of each processing block for one 64 channels-burst, Figure 9.3 depicts the generated input/output data volume for/by each processing block by the same configuration.

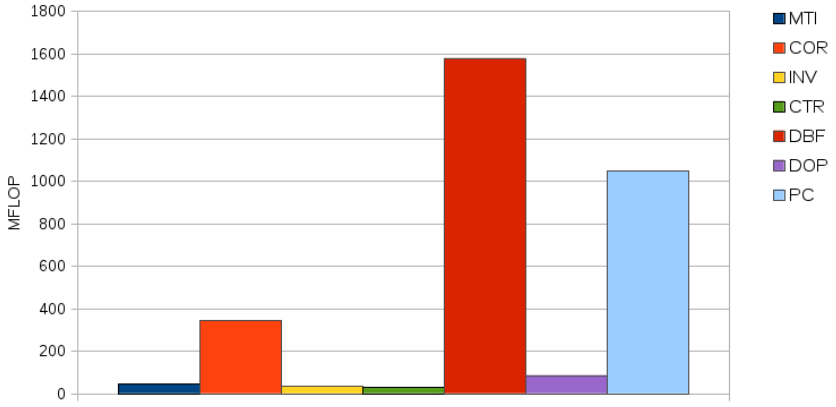


Figure 9.2: Computing load of each processing block for a 64 channels-burst (Floating-point Operations).

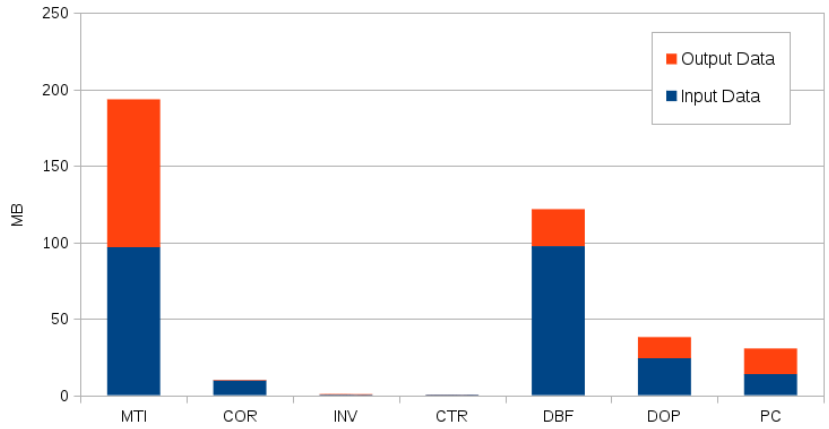


Figure 9.3: Input/Output Data volume per burst for each processing block in a 64 channels configuration.

9.2 Experimental Setup

9.2.1 Hardware Setup

In the next sections, we use two different execution platforms with one and two general-purpose multicore processors:

1. The first platform is an Intel Core i7 Q720 1.6 GHz (Max Turbo Frequency: 2.8 GHz), it contains 4 Physical Cores and 8 Threads and has a 45W Maximum TDP (Thermal Design Power) as specified by the constructor [Intb].

2. The second platform is an SMP platform with two Intel Xeon E5620 2.4 GHz (Max Turbo Frequency: 2.66 GHz). Each processor has 4 Physical Cores and 8 Threads. Its constructor declares a maximum TDP of 80W per-processor, i.e., about 160W for our target platform [Intc].

9.2.2 Software Setup

We consider mainly three radar configurations of 64, 32 and 16 channels. Total processing time must be equal or lower than 20 ms in the worst case in order to meet real-time processing requirements without loosing any data. Since "DOP" and "PC" tasks performs many FFT (Fast Fourrier Transform) in their processing, we use the free FFTW 3.3 library [FJ05] in the case of the PC task. DOP's FFT is much smaller and is written by us.

Serial FFTW is used in the sequential version of PC task and threaded FFTW is used in the parallel one The GNU compiler v4.6.3 is used to compile the different version of the target application. Finally, applications are executed on a Linux Debian OS with the 3.0 Linux Kernel.

9.3 Serial Execution

The initial basic C++ implementation of our processing chain is fully serial. Figure 9.4 shows the execution time of the entire serial processing chain for different workloads (16, 32 and 64 Channels-Burst) and depicts the spent execution time by each processing block of our algorithm. As expected, the application's execution-time is dominated by DBF, PC and COR tasks. Parallelization should target particularly these blocks in order to reduce the over-all execution time. This basic sequential implementation takes about 7 seconds to process a 64 channels-burst with a fixed beam count on an SMP platform with two Intel Xeon E5620 at 2.4 GHz. So, it runs 350 times slower than the required real-time processing time (20 ms). We outline that the digital beam forming task "DBF" generates the same beam count disregarding the input channels count. This explains the near-constant execution time of the two last blocks "DOP" and "PC" which have a constant workload in all configurations.

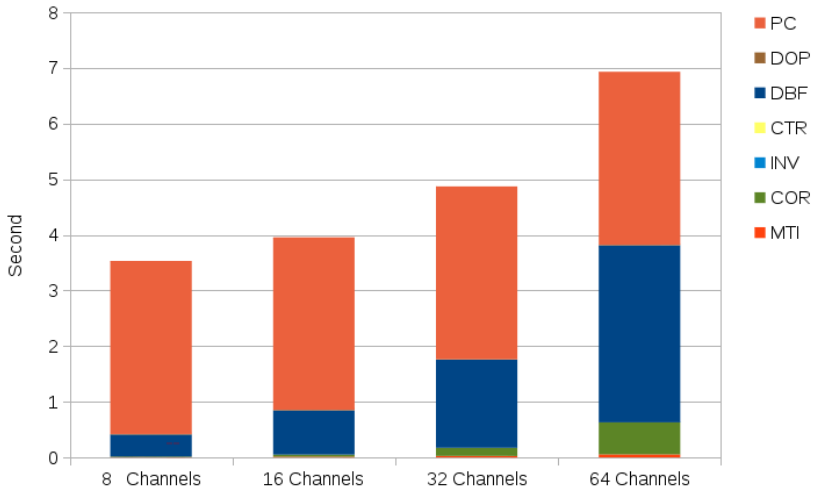


Figure 9.4: Execution time of the initial serial implementation for 16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz (16 Threads / 8 Cores).

9.4 Parallelization Methodology

In order to parallelize our application we follow a simple parallelization methodology which we detailed in previous work [KLLDS12]. This parallelization technique pass through three main steps which are:

1. Decomposing our application into a set of tasks
2. Specifying the parallelism of these tasks.
3. If possible, parallelizing each of these tasks: each task may be decomposed into finer grain tasks to express finer grain parallelism.
4. Finally, instruction-level parallelism (SIMD) may be expressed using vectorized types or concurrent GPU vectors .

Figure 9.5 gives an overview of our parallelization methodology. We outline that XPU allows hierarchical expression of several types of parallelism including task parallelism, data and temporal parallelism at different level of granularity at both thread level and instruction level.

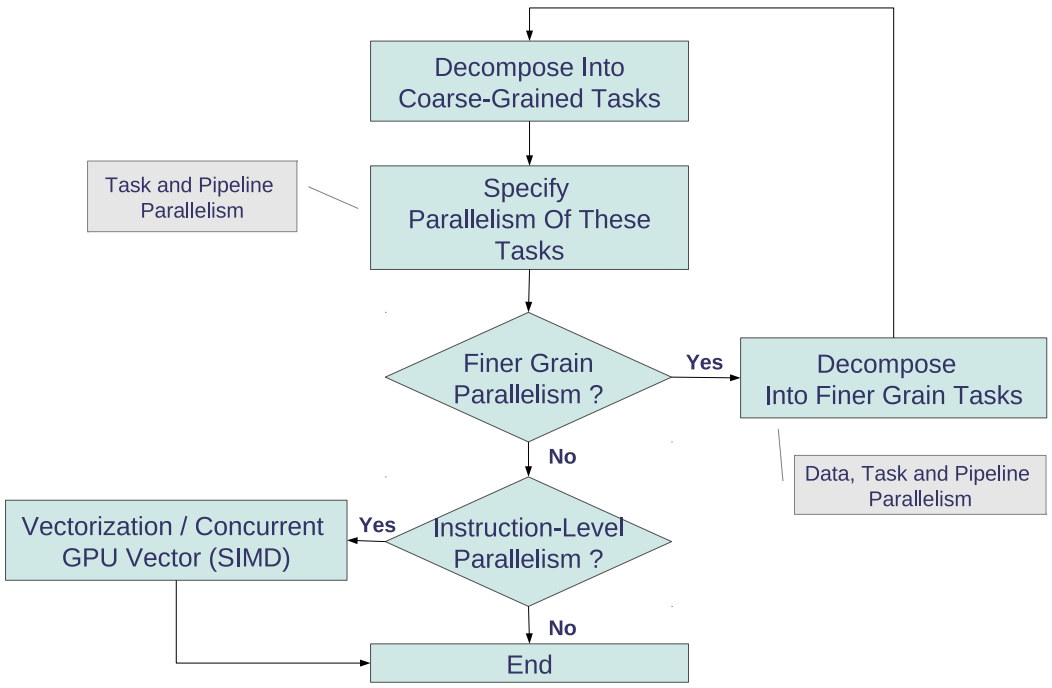


Figure 9.5: Application Parallelization Methodology: Application is progressively parallelized at different levels of granularity by expressing different types of parallelism.

9.5 Task Parallelism Extraction

A first look to the algorithm enables us to distinguish clearly an inherent parallelism between the MTI (Moving Target Indication) and the four following blocks: SEL (Selection), COR (Correlation), INV (Inversion) and CTR (Control). Exposed task parallelism in our study case is relatively weak since the five targeted tasks are not much time-consuming. Consequently, expected speedup is relatively low but nevertheless important. We note that task parallelism may be much more available and effective in other study cases.

Assuming we have a function which represents each of these processing blocks, an XPU task is defined for each of these serial functions through a single line of code. Once task defined, their trivial execution configuration, described in Figure 9.1, can be expressed at the cost of another single line of code as shown in Listing 9.2.

Since MTI and SEL accesses to a common input (the burst) respectively in write and read mode, XPU run-time will transform both of them into critical sections in order to protect the burst data from conflictual concurrent accesses (also known as "race condition") to ensure a safe and coherent execution. This may annihilate parallelism in our case, so, we choose to duplicate input data to preserve parallel execution of MTI and the other four concurrent blocks.

```
1  void main() {
2   // task definition
3   task rcv_t(&rcv, burst_count),
4        mti_t(&mti, args), ...;
5   task_group * burst_processing; // task graph
6   burst_processing = sequential(rcv_t,
7                                 parallel(sequential(sel_t,cor_t,inv_t,ctr_t),mti_t),
8                                 dbf, dop, pc);
9   while (burst_available)
10      burst_processing->run(); // parallel processing of each burst
11  }
```

Listing 9.2: Coarse-Grain Task Parallelism Implementation Using XPU

As expected, parallelization of the sequential processing tasks does not provide a significant speedup since MTI, SEL, COR, INV and CTR are not time-consuming in comparison with DBF and PC. Nevertheless, this unavoidable parallelization step reduces slightly the over-all execution time as shown in Figure 9.6. The gained execution time grows proportionally to the input data size. In order to make this task-parallelism speedup significant, COR, DBF and PC execution times must be significantly reduced to be as closer as possible to the MTI execution time.

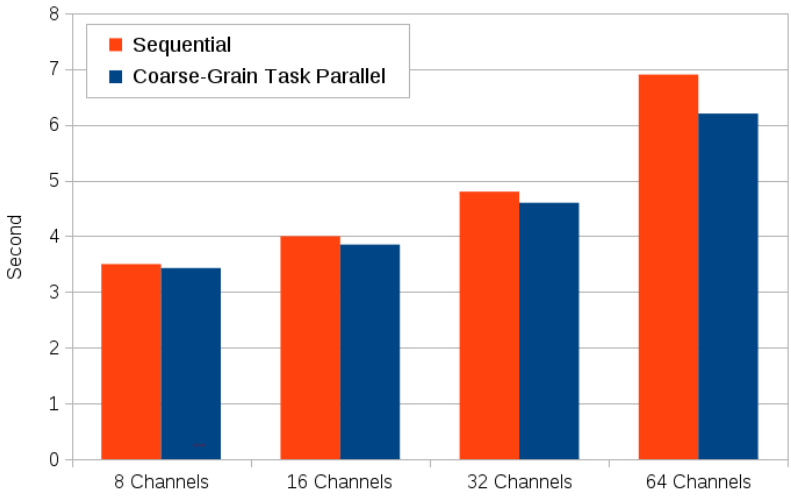


Figure 9.6: Execution time of both the sequential version and the task-parallel version for 8,16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz.

In the next sections, we try to parallelize each block at thread and instruction level through XPU data parallel execution patterns.

9.6 Data Parallelism

As we stated earlier, this algorithm processes simultaneously all data (signal's samples) received from several channel of the phased array radar system. Many blocks of the algorithm performs the same operation on each channel. This can be exploited to implement data parallelism in each of these stages at instruction-level by using the vectorization capabilities of XPU and at thread-level through replacing sequential loops by parallel ones using the "parallel _for" execution pattern available in XPU. COR, DBF and PC should be particularly targeted by this parallelization.

9.6.1 Vectorization

Vectorization can be implemented by replacing regular simple precision float by the "xpu::vec4f" built-in type which translates transparently simple operations on floating-point into SIMD operations on four floats simultaneously. This instruction-level parallelism is implemented on top of x86 SSE streaming intrinsics and may be extended in future works to support other SIMD extensions such as AVX or Altivec. We note

that data should be aligned in memory to make vectorization efficient. This can be performed through the XPU's aligned allocator which is compatible with standard C++ STL containers.

Vectorization has been used in MTI, COR, DBF and PC tasks. Figure 9.7 and Figure 9.8 show that instruction-level parallelism can be a great parallelism multiplier for data parallel tasks such as MTI and COR where the same operation is performed on large vectors of samples. We note that SIMD impact vary depending on the workload size implying both data size and computing load: the vectorized COR code performs 10 times faster than the original sequential code while the vectorized MTI code achieve only about 20% execution speedup.

9.6.2 Parallel Loop

Instruction-level parallelism offers limited scalability since SIMD may take advantage of processor frequency increasing but cannot benefit from processor count increasing without thread-level parallelism. Parallel loop implements data parallelism at thread-level and provides good scalability on multicore architectures. We use XPU's "parallel_for" execution pattern in conjunction with vectorization to parallelize several tasks. The XPU's "parallel_for" construct adapts dynamically to the underlying architecture to exploit all available processors.

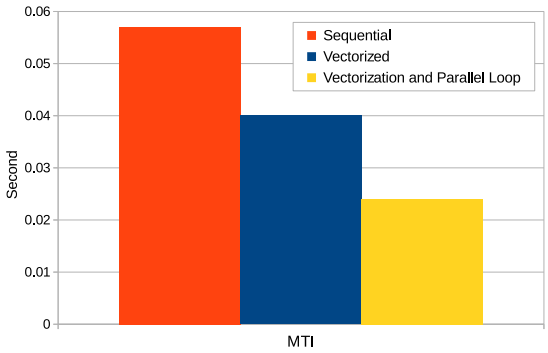


Figure 9.7: Parallelization of the MTI block at instruction and thread level using the XPU vectorization and the parallel loop.

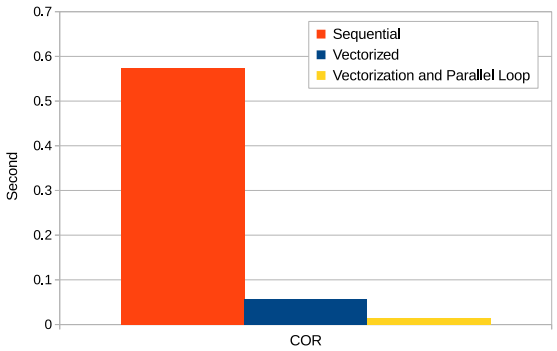


Figure 9.8: Parallelization of the COR block at instruction and thread level using the XPU vectorization and the parallel loop.

Listing 9.3 shows an example of parallel for implementation to perform simultaneously several FFT on several pulses (pulse: a set of samples). The "parallel_for" construct has been used to parallelize MTI, COR, DBF, DOP and PC tasks. The vectorized code has been reused in the parallel loops. Figure 9.7 and Figure 9.8 depicts a significant speedup in comparison with the initial serial version: the parallelized COR version runs about 48 times faster than the original sequential version and 5 times faster than the vectorized version on the bi-processor 16 Threads SMP platform.

```
1  int fft(int from, int to, int step, float ** pulses)
2  {
3    for (int i=from; i<to; i+=step)
4        cplx_fft(pulses[i]);
5  }
6
7  void pc(float ** pulses, int pulse_count)
8  {
9    task fft_t(fft, 0,0,0, pulses);
10   parallel_for p(0, pulse_count, 1, &fft_t);
11   p.run();
12 }
```

Listing 9.3: A simplified example of parallel for loop use in the PC task. Using XPU

9.6.3 Performances

As shown in Figure 9.9, after parallelizing the most time-consuming blocks of our processing chain at thread and instruction-level, the over-all processing time of a 64 channels-burst on the dual Intel Xeon E5620 platform dropped from 6.9 seconds (initial sequential version) to 0.045 second (in the worst case) for the parallel version. The 64 channels configuration does not satisfies the real-time requirement (20 ms in the

worst case). However, we are able to process input data in real-time in the 32,16 and 8 channels configurations on the same platform.

With a maximum TDP of 160 Watts, the first platform may not be embeddable due to potential energetic constraints. Thus, a platform based on one Core i7 Q720 with a maximum TDP of 45 Watts may be more suitable. We executed our application on this platform. Figure 9.10 shows that our application run twice as slow as on the SMP platform. Consequently the 32 channels configuration does not allow real-time processing on that platform. However, real-time processing can be performed in the 8 and 16 channels configurations.

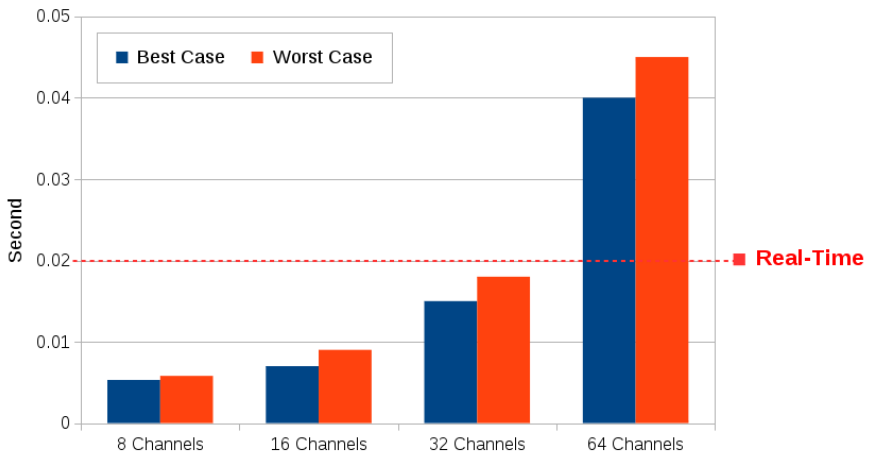


Figure 9.9: Execution time achieved by the parallelized version for 8,16,32 and 64 channels-burst on 2 x Intel Xeon E5620 2.4 GHz.

As shown in Figure 9.9 and Figure 9.10, the current application displays a good scalability. We believe that real-time processing in all our four configurations may be achieved on faster platforms with more processing cores and higher clock frequency.

As we have seen, data parallelism can offer significant execution speedup especially when coupled with task parallelism. However, available parallelism may be limited by producer-consumer dependencies between tasks. In this case, temporal parallelism (a.k.a. pipeline parallelism) can be a very useful parallelism multiplier.

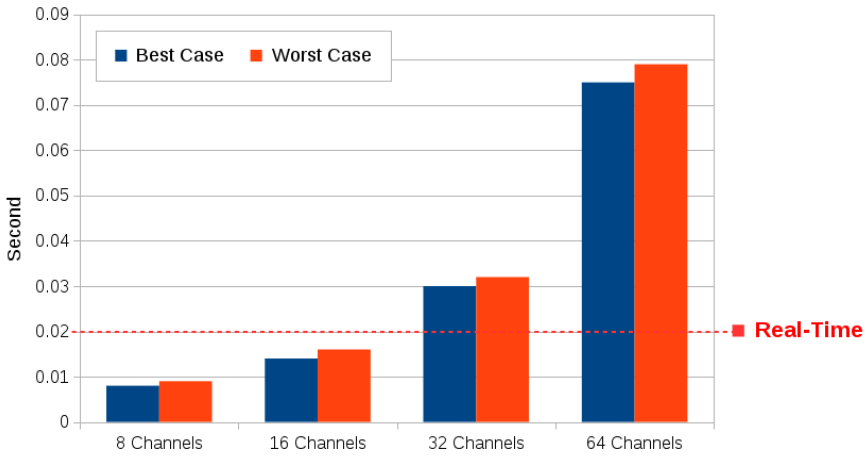


Figure 9.10: Execution time achieved by the parallelized version for 8,16,32 and 64 Channels-burst on an Intel Core i7 Q720 1.6 GHz

9.7 Pipeline Parallelism

At the opposite of serial execution pattern where all tasks are executed sequentially, pipeline exploits the available parallelism by executing simultaneously all its processing stages and serializing only the dependent activities. This allows potential throughput improvement especially in the case of applications involving real-time continuous stream processing such as in our case.

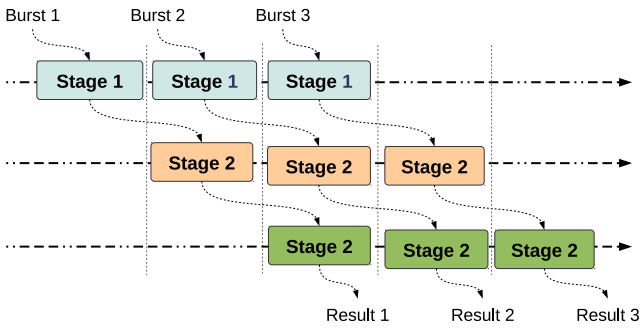


Figure 9.11: The radar processing algorithm is split into three balanced stages which executes concurrently in a pipeline execution pattern. Concurrent pipeline stages works on different bursts simultaneously.

In our application, a single burst must be processed by the different tasks in the specified order to preserve data coherency: these tasks are acting as a consumer-producer chain. However, multiple independent bursts can be processed simultaneously by different processing stages without violating the producer-consumer dependencies as shown in Figure 9.11. This consumer-producer relationship allows us to use the pipeline execution pattern to exploit the available parallelism.

We use the "pipeline" execution pattern of XPU to implement a three-stages pipeline execution configuration. For more information on the pipeline parallelism and the XPU pipeline skeleton, reader can refer to chapter 6 of this dissertation. Figure 9.11 illustrates how the three stages of the pipeline can process simultaneously three different bursts without violating producer-consumer dependencies. The different stages may be executed differently depending on the workload of each stage however stage's ordering is preserved for each received burst. As illustrated in Figure 9.12, we regroup our tasks into three processing stages:

1. **Stage 1** holds MTI, SEL, COR, INV and CTR tasks. We note that the "parallel" construct specifies parallelism between MTI and the other four tasks which are executed sequentially. In addition, we use two "parallel_for" loops in MTI and COR in conjunction with vectorization.

2. **Stage 2** contains the parallel version of DBF : a "parallel_for" loop enables parallel processing of different channels.

3. **Stage 3** is composed of DOP and PC tasks. Both of them uses "parallel_for" loops to process simultaneously several beams.

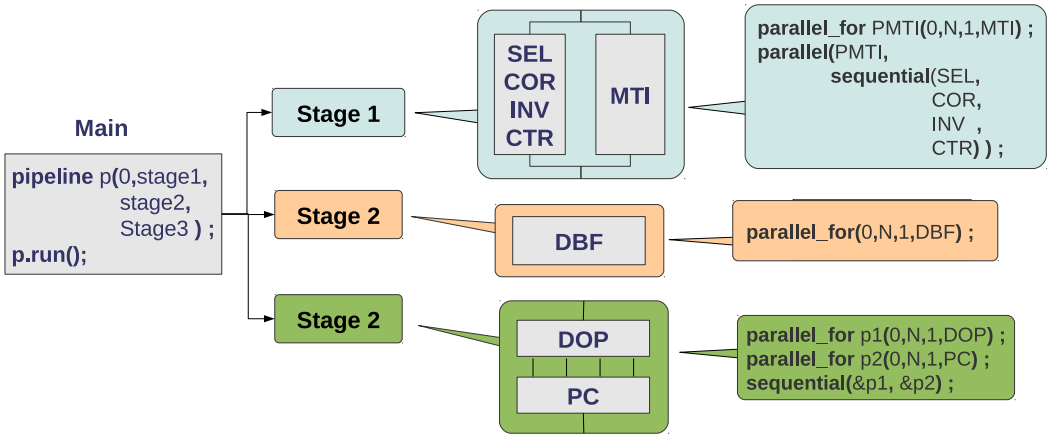


Figure 9.12: The final parallel application is parallelized using different types of parallelism at different levels of granularity.

This task regrouping pattern aims to load-balance the pipeline stages since the various processing blocks have different workloads. Each pipeline stage regroup a set of processing blocks. The regrouping is guided by the execution time of the blocks to obtain a pipeline with load-balanced stages.

A FIFO (First In First Out) queue ensure data transfer between the different stages. The FIFO sizes are limited by the available memory and may have a non-negligible impact on the achieved performances.

As shown in Figure 9.13, pipeline parallelism allows real-time processing in all our four configurations on the 8 threads platform (Intel Core i7 Q720). By observing the load of the different processor's cores when executing the application as well as the achieved execution times, we can conclude that the pipeline-based version exploits the computing resources more efficiently than the previous parallel version.

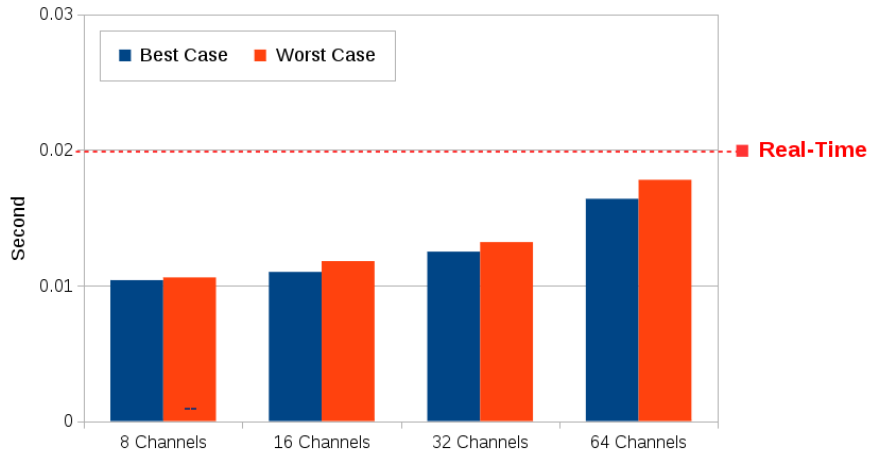


Figure 9.13: Worst and best execution time of the parallel version with pipeline parallelism version execution for 8,16,32 and 64 Channels-Burst on the Intel Core i7 Q720 1.6 GHz (8 Threads)

9.8 Conclusion

As we have seen previously, we parallelized our target application progressively by expressing several parallelism types at different granularity levels starting from coarse grain tasks to finer grain ones. Use of both task, temporal and data parallelism allowed us to extract a significant amount of parallelism and to achieve high performance and good scalability. Through this application, we outline the programmability of the XPU

framework which enabled us to easily express various types of parallelism at all levels of granularity at the cost of a little amount of parallelism related extra-code and in the same time. Moreover, XPU enabled us to reuse most of the legacy sequential code without significant alteration. This programmability can improve significantly programmer's productivity.

By easing parallelism expression on general purpose multicore architectures such as the x86 architecture, many real-time digital signal processing applications are likely to be implemented efficiently on such architectures making them an attractive alternative to expensive specialized processing architectures such as FPGA and DSP based platforms.

10

Polyphase Filter Bank Processing Application

In digital signal processing, the Discrete Fourier Transform (DFT) [PM07] is the computational basis of spectral analysis. Particularly, the DFT is a fundamental tool to analyze the frequency spectrum of a signal. The DFT transforms a signal, represented by an equally spaced time series or "samples" (in the time domain), into a frequency spectrum (in the frequency domain) which consists in a set of equally spaced coefficients or "bins" which represent the energy of each frequency present in the signal.

Unfortunately, the straightforward direct application of the DFT on a signal exposes two significant drawbacks known as energy leakage and scalloping loss. Depending on the sampling rate and the number of points of the DFT, a strong input tone may appear in more than one frequency bin and may consequently shadows or "drown out" signals of interest in the nearby bins, this phenomenon is known as "leakage". Figure 10.1 shows a example of frequency spectrum that illustrates the "leakage" phenomenon that affect the bin of interest and its sidelobes. DFT scalloping loss refer to the fluctuations which affect the overall magnitude response of a DFT due to windowing. The windowing shape is visible in Figure 10.1.

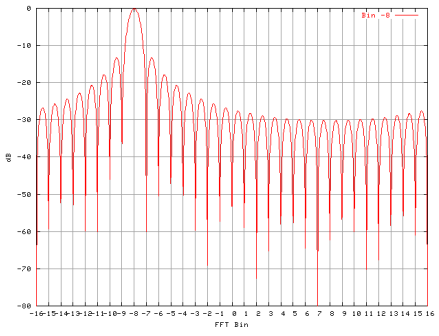


Figure 10.1: Non-Windowed FFT exposes poor out-of-band rejection due to "leakage" phenomenon which affect the nearby of the bin of interest.

The Polyphase Filter Bank (PFB) processing alleviates the aforementioned disadvantages of the straightforward DFT. As shown in Figure 10.2, the application of the

PFB before the DFT produces a flat magnitude response across the analyzed frequency channel and provide efficient suppression of out-of-band signals.

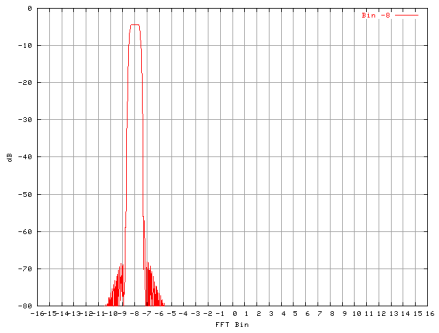


Figure 10.2: Applying PFB processing before the FFT provides a flat response on the bin of interest with an excellent out-of-band rejection and alleviates both the "leakage" and "scalloping loss" phenomenons.

10.1 The Quadrature Polyphase Filter Bank Technique

The Quadrature Mirror Filter Bank (QMFB) [PM07] [Lyo10] is a powerful signal processing technique that offers a fine spectral decomposition of a signal : a wide frequency band signal is decomposed into a set of narrow frequency sub-band. The QMFB is implemented as a set of band-pass filters that isolate each of these sub-bands. Several filter banks can be used as cascaded filtering stages to perform hierarchical spectral decomposition. In this case, the main wide frequency band is decomposed into several narrow sub-bands, each of these sub-bands is then decomposed in turn into narrower sub-bands, etc... Figure 10.3 gives an overview of a sub-bands decomposition using a filter bank. The QMFB filtering has many applications such as frequency spectrum analysis, audio equalization, signal compression or speech recognition.

While improving the magnitude response across the analyzed frequency band, the PFB implementation is often computationally intensive and consumes more resources than a simple DFT. Consequently, efficient implementation of the PFB can be critical to make its qualities outweigh resources consumption issue especially in real time signal processing applications.

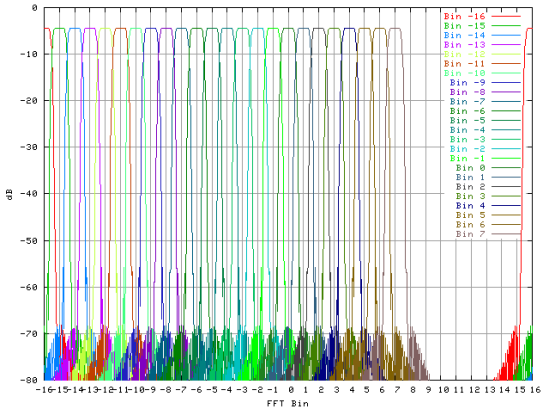


Figure 10.3: The frequency response of a Polyphase Filter Bank composed of a set of overlapping filters Polyphase Filter Bank Processing.

In addition to signal filtering within a bank of receivers, filter bank has many applications including equalization, re-sampling (up-sampling or down-sampling) and signal compression. Bank filtering is also widely used in some applications such as speech recognition and frequency spectrum analysis.

10.2 Context

In our context, the Electronic Warfare (EW) department of Thales Airborne Systems designed an application using the PFB technique to analyze signals within a specific frequency band in order to detect particular signal types and analyze their behavior. The application is required to processes a huge amount of data in real time. Thus, the application is both computationally intensive and data-intensive.

Traditionally, specialized parallel hardware such as FPGA and DSP boards are instinctively used for such heavy real-time signal processing applications. Such hardware may offer high performances but at the cost of poor programmability and consequently low productivity: programming FPGA or DSP is a heavy process and a hard task which requires a deep understanding of the target hardware architecture in addition to strong parallel programming knowledge and skills and requires the use of specialized compilers, languages and tools... Moreover, such hardware may reduce significantly application portability and forward scalability.

Recent general-purpose multicore architectures (GPMA), such as the widely available x86 architectures, are becoming widely available at a relatively low cost in comparison to DSP and FPGA boards. GPMA can offer an attractive alternative to those specialized processors since they can achieve acceptable performances at the cost of

less programming effort. Particularly, stringent real time requirements for most digital signal processing applications seem still beyond reach of today GPMA: these DSP applications require supplemental dedicated accelerators or even fully dedicated SoC architectures. However, GPMA show such promises and it is likely that in a near future, DSP applications could be modeled appropriately and executed in such GPMA.

Through this Thales EW application, we tried to investigate how we can use XPU to implement the PFB-based algorithm on a general-purpose multi-core processor. A sequential C++ version of the application was already available. This was an ideal occasion to evaluate the both the programmability and the performance of XPU. The programmability was evaluated in term of reused sequential code and required extra-code to express different types of parallelism. The application has been progressively parallelized while measuring the delivered performance and the achieved performances.

10.3 Algorithm Description

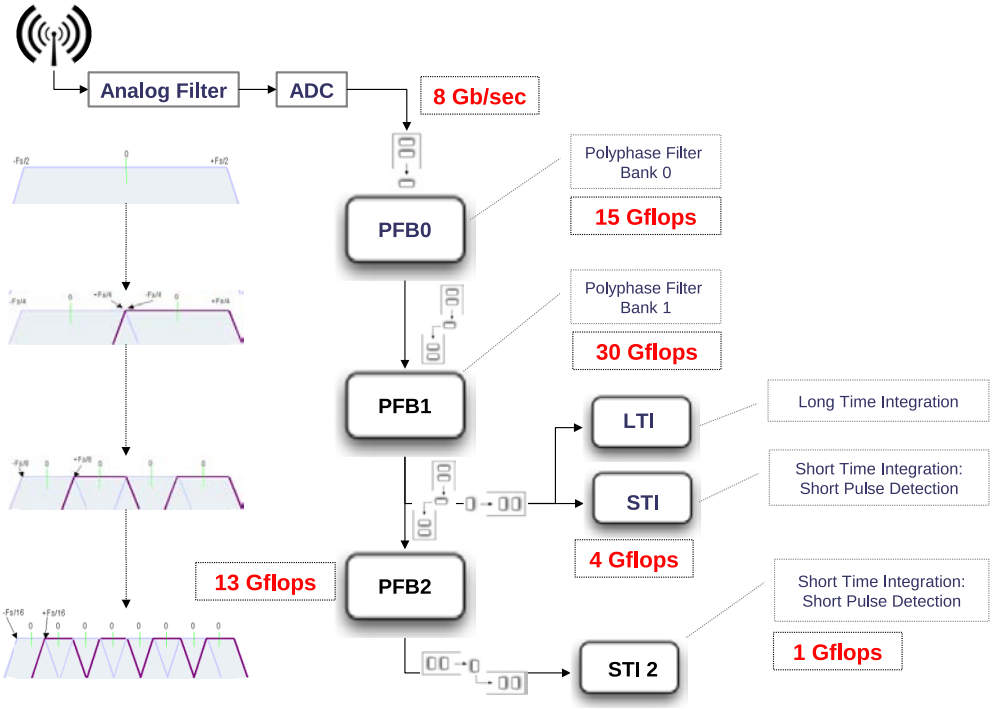


Figure 10.4: Overview of the polyphase filter bank application in its single channel configuration.

The target application aims to analyze an arbitrary signal within a specific frequency band using three cascaded polyphase filter bank stages, namely PFB0, PFB1 and PFB2.

An intermediate detection of long and short pulses is realized after the second filtering stage and a final short pulse detection is performed at the last stage.

The application is designed to process one or multiple channels simultaneously. In the later configuration, the same filtering chain is duplicated N times while the detection blocks are shared between these N filtering chains. Multi-channel processing is required to realize digital beam forming and direction finding. The more the channels, the better the angular precision of direction finding. In our case, ideally, a four channels configuration is required to achieve a good angular precision.

Figure 10.4 gives an overview of the algorithm for a single channel processing. As depicted in the figure, an hierarchical spectral decomposition is performed progressively at each filtering stage. The output of each filtering stage is used as the input of its following stage. The intermediate signal detections exploits the outputs of second filtering stage "PFB1", while the last detection use the output of the last PFB stage.

Each channel of the Analog to Digital Converter (ADC) has a throughput of around 8 Gb of raw samples per second and feeds the processing chain continuously. The later is computationally intensive: in order to meet real-time processing requirement, the application must perform more than 60 Billions Floating Point Operation (GFLOP) per second for each channel. The four channels configuration expose 237 GFLOPS of computing load. Achieving real-time processing in the later configuration on a GPMA platform become challenging.

In the next paragraphs, after exposing the profile of the application and identifying the most time-consuming operations, we try to parallelize progressively the application at different level of granularity to achieve the best possible speedup over the sequential version.

10.3.1 Application Profile

The sequential version of the application is composed of three filtering blocks or stages, namely "PFB0", "PFB1" and "PFB2" and three detection blocks which are "LTI", "STI1" and "STI2".

At each stage, the filter bank processing consist in performing filtering operations on each sub-band of our signal. The processing of each sub-band is mainly composed of three operations which are:

- Filtering
- FFT Processing
- Down-Sampling or Decimation (Except for the last PFB stage)

The FFT are performed on a relatively small number of samples, especially those which are in the two first stages and thus FFT processing is not the most time-consuming operation. The decimation is simply a selective data copy which pick one sample out of N samples. Hence, decimation is not a computational operation. Consequently, the filtering part dominates the filter bank processing in term of computation intensity. Filtering consists in traditional convolution of the processed samples with a pre-computed filter.

The remaining processing blocks ("LTI", "STI1" and "STI2") perform various operations on the intermediate outputs of the different processing stages. These operations includes the detection of short and long pulses, i.e., detection of present signals, their levels and frequencies. In the multi-channel configuration, detection blocks performs additionally the direction finding through phase shifting and digital beam forming. We note that in the multi-channel configuration, the detections blocks are shared between all channels and only the PFB blocks are duplicated. Thus, the detection blocks introduce less computational load than the PFB blocks.

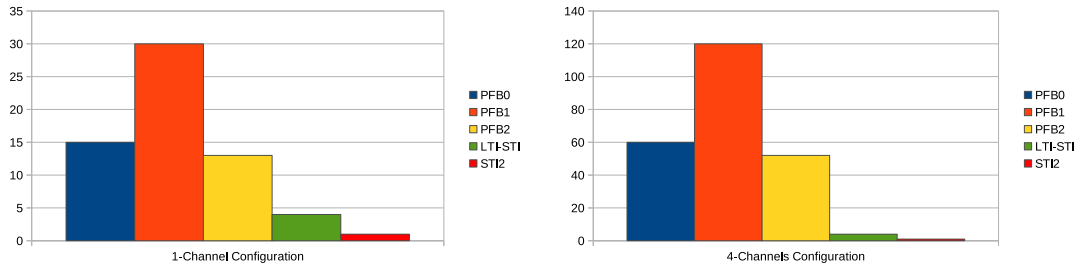


Figure 10.5: Computing Loads (GFLOP/S) respectively in the 1-Channel and 4-Channels Configurations

Figure 10.5 shows the relative computation loads (GFLOP/S) of the different computation blocks in two configurations: the one-channel and the four channels configuration. We can see that PFB processing blocks start to dominate the over-all computation load as the channels count increases. In the single channel configuration, the PFB processing blocks represent more than 90 % of the processing load with 58 GFLOPS out of 63 GFLOPS. In the four channels configuration, the PFB processing load raise to about 98 % with 232 GFLOPS out of 237 GFLOPS while the computing load of the detection blocs does not raise far from the initial 5 GFLOPS.

10.4 Parallelization

In order to parallelize the target sequential application, we follow the simple parallelization process that we have introduced at the first chapters of this thesis which can be summarized as follows:

1. Decomposing the program into Tasks.

2. Task parallelism extraction (If available).

3. Data parallelism extraction (If available).

4. Pipeline parallelism Extraction (If available).

5. The Previous operations can be reproduced at finer grain to capture more parallelism when available.

10.4.1 Task Definition

In the target sequential application, filtering operations inside PFB0, PFB1 and PFB2 are implemented and encapsulated into separated C++ classes as object methods. XPU allows us to reuse these object methods as tasks and thus isolate the different processing blocks of the application before reasoning about the parallelization. You can refer the "Task Defintion" chapter to learn more about reusing object methods and functions as tasks in XPU.

Once the program is decomposed into small tasks. These tasks can be organized using the different XPU patterns to specifies the different types of parallelism. Since filtering is one of the most time-consuming tasks, we try to parallelize it using data parallelism pattern at both task and instruction levels. But before starting to parallelize each PFB at fine grain, we can capture pipeline parallelism at coarser grain between the PFB stages since these stages form a consumer producer chain.

10.4.2 Pipeline Parallelism Extraction

The three PFB stages expose a consumer-producer relation between adjacent stages. The stream nature of the input data that "flows" through these three stages allows us to use the XPU *pipeline* pattern to make these stage running concurrently while preserving the data coherency. Reader can refer to chapter 7 for more information about pipeline parallelism and th XPU pipeline skeleton.

Figure 10.6 depicts the first pipeline configuration that include not only the PFB processing stages but also the detection stages.

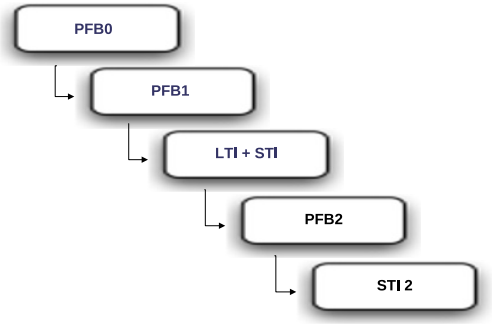


Figure 10.6: Both the filter bank stages and the detection stages are integrated as processing stages within a pipeline execution pattern.

We note that the detection blocks ((LTI,STI) and STI2) has been included in the pipeline as processing stages but can be executed asynchronously out of the pipeline stages since their dependencies are limited to specific PFB stages without generating dependencies over the adjacent PFB stage. More precisely, the (LTI+STI) detection block depends on the output of PFB1 stage, however the PFB2 is not dependent on that detection block, hence a false dependency between them is generated if we simply include the (LTI+STI) in the pipeline as a stage.

We used this simple pipeline configuration in our initial implementation for the sake of simplicity and rapid prototyping, but we switched to a more efficient configuration with four stages pipeline that include the three PFB stages and the last detection stage (STI2) in the pipeline while making the first detection block (LTI+STI) executes concurrently with the pipeline. The (LTI+STI) block is executed asynchronously and gets its input data from the second PFB stage when available. The later configuration is depicted in Figure 10.7.

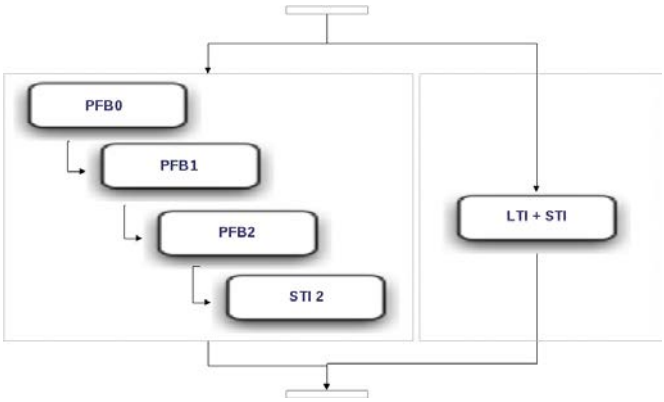


Figure 10.7: The first detection block (LTI+STI) is executed outside the pipeline in an asynchronous fashion to avoid false dependency between (LTI+STI) and PFB2.

In order to evaluate the pipelining effect without additional task parallelism between the pipeline and the (LTI+STI) block, we measured the execution time of the initial-pipeline only version. As depicted in Figure 10.8, the pipeline configuration performs three times faster than the original sequential version.

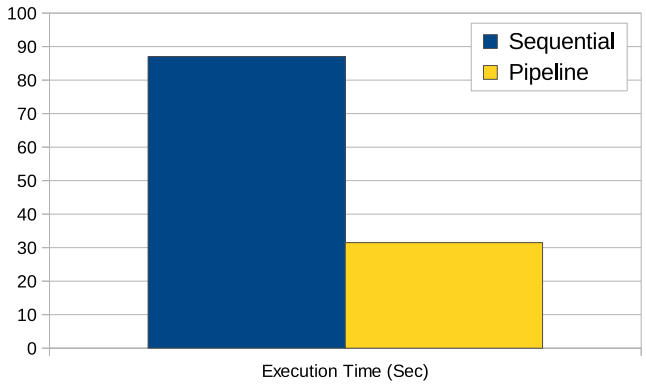


Figure 10.8: The initial pipeline-only configuration perform about three times faster that the original sequential configuration on an Intel Xeon W3680 with 12 Threads at 3.3 GHz.

While improving considerable the processing time, the pipeline execution pattern suffer from inherent limited scalability due to limited stage count: the pipeline provide a good forward scalability until the the number of processing stages. In order to extract more parallelism, we tried to extract data parallelism using the XPU "parallel_for" pattern and the vectorization capabilities of XPU.

10.4.3 Data Parallelism Extraction

Data parallelism can be extracted both at task level through parallelizing loops and at instruction level through using the XPU vectorization capabilities. The following paragraph details both of them.

10.4.3.1 Vectorization

We used the vectorization capability of XPU to speedup the filtering process. At each processing stage (filter bank), the main filtering operation implies uniform multiplication of large number of signal samples by the filter coefficients then the accumulation of their sum (FIR applicationd).

Both signal samples and filter coefficients are stored as tables of floats. Vectorization can be naturally implemented by processing the regular float buffers as XPU "vec4f"

buffers. This allows us to process four pairs of floats at once at each operation instead of one single pair float at a time and thus increasing filtering throughput.

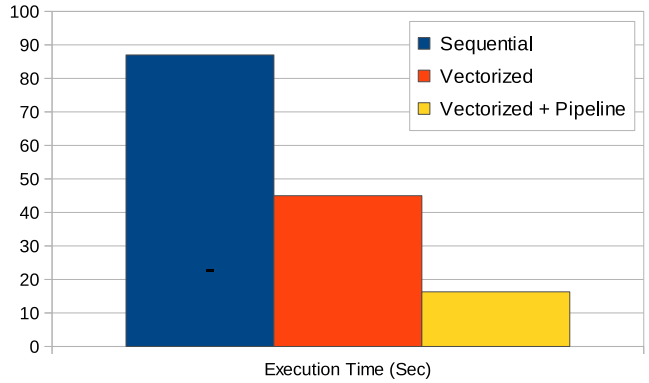


Figure 10.9: Effect of vectorization in comparison on the original sequential version on an Intel Xeon W3680 at 3.3 GHz

As depicted in Figure 10.9, the vectorization acted as a great parallelism multiplier and offered a two time speedup over the original sequential version on a an Intel Xeon W3680 at 3.3 GHz (2 x 6 Cores) in a single channel configuration. The vectorization offers a good speedup, however we are still far from real-time processing for a single channel. Also, we note that vectorization exposes very limited scalability, since it can only take advantage from increasing processor frequency but not processing unit multiplication. Thus, vectorization need to be combined with thread level parallelism to achieve higher speedup and better scalability. This can be done by trying to parallelize the numerous loops within the different PFB blocks and also detection blocks.

10.4.3.2 Parallel Loop

The application reads a large number of samples before starting the processing. The size of this input buffer is configured by the user. Each filter has a fixed length and operates on a limited set of sample at a time. Thus, several small subsets or partitions of the main samples buffer can be filtered simultaneously.

Parallelization of each filter at each PFB stage can be easily done using the XPU *parallel_for* pattern. Instead of iterating over the samples and performing filtering sequentially on each subset of them, multiple iterations can be executed concurrently. The number of concurrent filtering is fixed by the *parallel_for* dynamically at run-time and will depends on the available processing units (processors and cores). This guarantee a good forward scalability when switching from one platform to another with larger processors count.

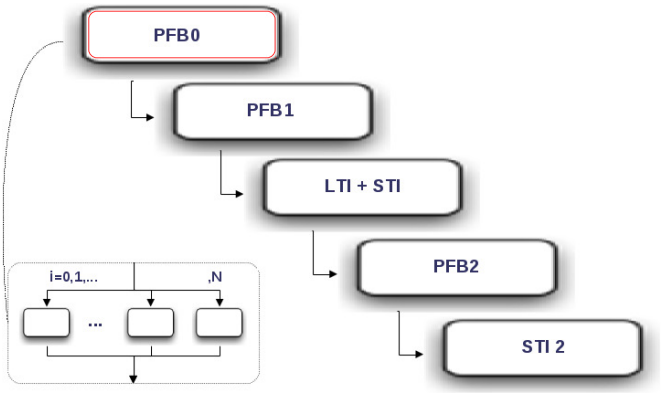


Figure 10.10: Data parallelism in PFB0 is implemented through an XPU parallel for loop.

The most time-consuming loop in PFB0 has been parallelized using the XPU *parallel_for* loop . This parallel loop is included in the first stage of the pipeline as shown in Figure 10.10. Many other loops inside the different PFB processing stages and the LTI detection blocks are parallelized the same way.

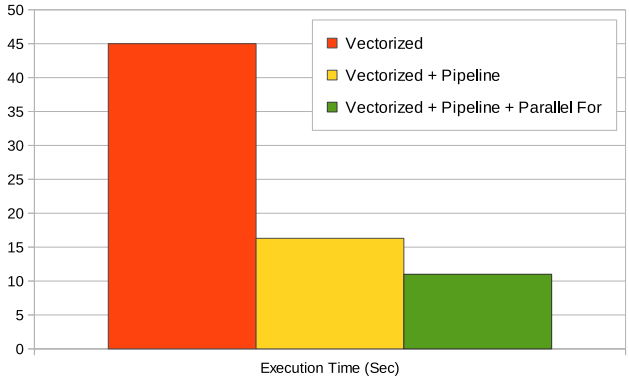


Figure 10.11: Parallelization of loops in PFB0 stages and LTI reduced execution time by more than 30% on an Intel Xeon W3680 with 12 Threads at 3.3 GHz.

As shown in Figure 10.11, the parallelization of the for loops in the different PFB stages and the LTI blocks allowed the application to execute 32.5% faster. We note that some of parallelized loops appeared to be not "enough" time-consuming and has been disabled since their sequential version achieved better performances.

10.4.4 Task Parallelism Extraction

In each PFB stages a set of filters are applied to the input signal (samples). These filters correspond to the adjacent frequency subbands. The filters are applied to the same common input: it consumes a common read-only data and write the result into separate output data. Hence filters are independent and the different subbands can be filtered simultaneously.

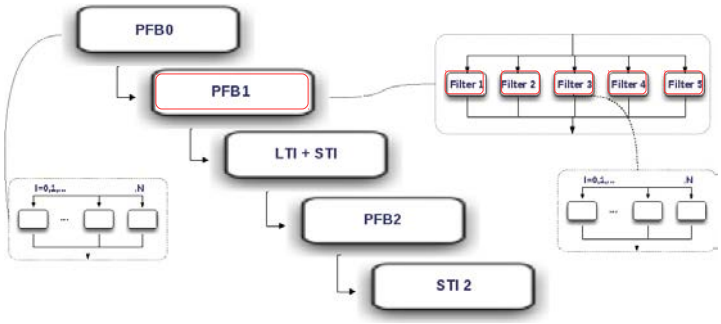


Figure 10.12: Both Task and Data parallelism are implemented in PFB1 by specifying parallelism between five filters and implementing each of these filters as a parallel for loop.

Figure 10.12 shows parallelism specification of the five filters of the PFB1 stage. Analogously, task parallelism can be specified for the twenty filters of the PFB2 stage. In addition to task parallelism, data parallelism in each filtering operation is specified through parallel loops and vectorization as we have seen in the previous paragraphs. In addition to concurrency between independent filters, the (LTI+STI) block can be split into two parallel blocks LTI and STI since they are independent. As depicted in Figure 10.13, the task parallelism between LTI and STI can be specified inside the LTI+STI stage.

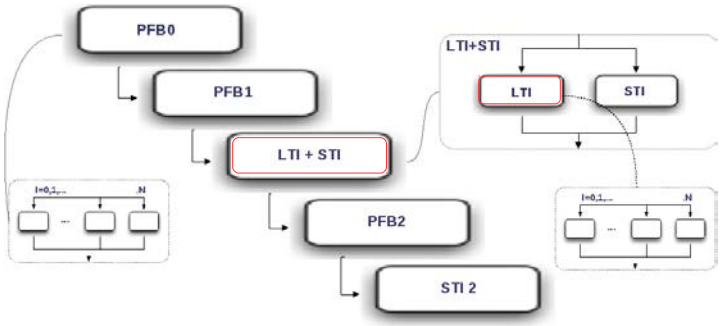


Figure 10.13: LTI and STI can be executed as parallel independent tasks. Additionally, data parallelism inside LTI can be specified through an XPU parallel for loop.

In term of code, a generic filter kernel is encapsulated in an XPU task for each subband, the filter coefficients corresponding to each subband as well as the input and output buffers are passed as argument to each task. In order to specify parallelism between the concurrent filters, we use simply the *parallel* keyword to parallelize their execution sa shown in Listing 10.1.

```
1  // the main filter kernel
2  int filter(const float * input, float * output, float * filter, int length)
3  {
4    // filter kernel ...
5  }
6
7  // PFB0 stage
8  int polyphase_filter_bank_0()
9  {
10   // ...
11   // tasks definition
12   xpu::task filter_1_t(filter, input, output_1, f1, length);
13   xpu::task filter_2_t(filter, input, output_2, f2, length);
14   xpu::task filter_3_t(filter, input, output_3, f3, length);
15   xpu::task filter_4_t(filter, input, output_4, f4, length);
16   xpu::task filter_5_t(filter, input, output_5, f5, length);
17
18   // running parallel filters
19   xpu::parallel(&filter_1_t, &filter_2_t, &filter_3_t, &filter_4_t, &filter_5_t)->run();
20
21   // ...
22 }
```

Listing 10.1: Parallelism specification of five filters inside PFB0.

Figure 10.14 shows the effect of task parallelism implementation on the over-all execution time. Parallelization of filters execution allowed the application to run 31% faster. Hence, to be closer the real-time execution requirement.

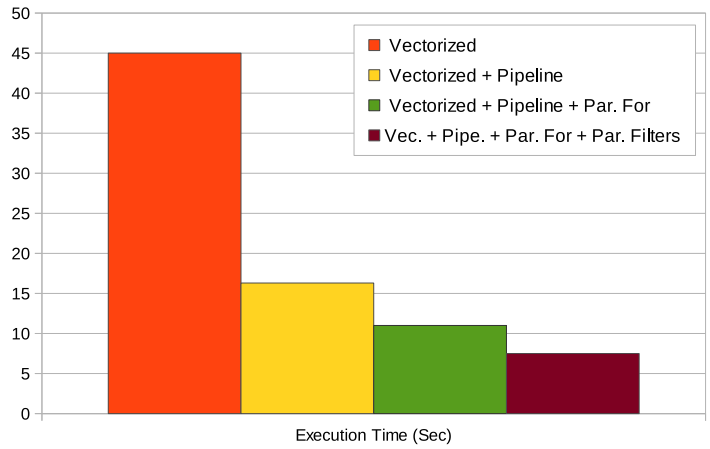


Figure 10.14: Implementation of Task Parallelism through parallelization of filters execution at each PFB stage allowed us to achieve a 31% execution speedup on the same platform (Intel Xeon W3680 with 12 Threads at 3.3 GHz).

10.5 Conclusion

As we have seen along this chapter, the quadrature mirror polyphase filter bank application has been parallelized progressively by expressing several parallelism types at different granularity levels starting from coarse grain tasks to finer grain ones. The data parallelism expression at thread and instruction level allowed us to exploit a significant amount of parallelism to speedup the execution. Additionally, the stream processing nature of the target application allowed us to exploit temporal parallelism through the pipeline execution pattern to achieve significant execution speedup. Finally, task parallelism expression captured even more concurrency to maximize throughput and reduce idle times in all pipeline stages.

Again, this application outlined the programmability provided by the XPU programming interface since it enabled us to express parallelism at the cost of little amount of extra-code and allowed us to reuse the legacy sequential code without alteration. This programmability improved significantly our productivity and reduced the development time spent on parallelization.

This case study demonstrated that by easing parallelism expression on general purpose multicore architectures such as the x86 architectures, many real-time signal processing applications are likely to be implemented efficiently on such architectures making them an attractive alternative to expensive specialized processing architectures such as FPGA and DSP based platforms.

11 Conclusion

11.1 Summary

The continuous proliferation of multicore processors in all segments of industry has placed software developers under great pressure to parallelize their program to take advantage of these platforms. Unfortunately parallel programming using low-level programming model is still a hard task for the average sequential programmer. The industrial context (*Thales Airborne Systems*) in which our researches took place confirmed this observation and outlined the need for a high-level parallel programming model which improves programmer productivity while delivering reasonable performances. This formulated the main motivation of our research work which is "*easing parallel programming without sacrificing performances*". The goal was to design a structured parallel programming model which offer a high programmability while delivering comparable performances to lower level approach. Design goals included:

1. Using a traditional programming language (C++) without any extension to provide good portability.
2. Easing the parallelization of existing sequential application while reusing the legacy code without alteration.
3. Easing parallelism expression by offering an intuitive and compact parallel programming interface.
4. Designing a flexible parallelization methodology which allows parallelization of general-purpose application which may expose different types of parallelism at different levels of granularity.

In this thesis we presented two C++ parallel programming frameworks namely XPU and FATMA which have been designed to satisfy the previously enumerated needs and to address the limitations of many state of the art parallel programming models in term of programmability and performance. XPU and FATMA exploit exclusively the potential of standard C++ programming language and its metaprogramming capabilities to ease parallelism expression while delivering performances which are comparable to low-level programming models.

XPU has been designed to ease explicit parallelism expression. It proposes a flexible parallelization methodology and allows the programmer to express several types of parallelism including task parallelism, data parallelism and pipeline parallelism at different levels of granularity. XPU expose a compact and intuitive programming interface (API) composed of very few keywords ("*task*", "*parallel*", "*sequential*", "*parallel_for*" and "*pipeline*"). Yet, thanks to extensive use of C++ metaprogramming techniques, XPU allows the programmer to easily express parallelism at the cost of few lines of code while reusing his legacy sequential code without almost without any alteration.

In the XPU programming model, the programmer is responsible of extracting parallelism before expressing it using the XPU API. In many applications, parallelism extraction can be a very hard task since the target application can exposes complex dependencies between a very large number of tasks. Tiled linear algebra algorithms are examples of such applications. Even when succeeding to extract parallelism using advanced task dependency analysis tools, explicit parallelism expression can be very challenging even with the simplest API. FATMA has been designed to address this limitation of XPU by extending its capabilities and providing automatic tasks parallelization at run-time. FATMA allows the programmer to parallelize transparently a large sequence of tasks by generating automatically the corresponding task dependency graph then using it to schedule the parallel tasks on the available processors. The FATMA API is even simpler that of XPU, it is composed of few keywords which corresponds to the three parallelizing phases (*program.add(task)*, *program.build()* and *program.run()*)

XPU has been used to parallelize several popular applications from the PARSEC Benchmark [BKSL08]. Our experiments have shown that despite its high abstraction, XPU can still deliver comparable performances to lower-level programming models. XPU has been also used to parallelize industrial applications such as a Radar signal processing application which has been presented in this thesis, int this application XPU allowed us to achieve real-time execution on a general-pupose multicore platform which can be used as an alternative to expensive specialized processing architectures such as FPGA and DSP platforms which are traditionally used in this application field.

We used FATMA to parallelize two tiled linear algebra algorithms and compared our implementation to different state of the art approaches. Despite being at its early development stages, FATMA achieved performances that are very close to specialized linear algebra libraries while displaying high programmability.

11.2 Limitations

While easing explicit parallelism expression, XPU exposes several limitations. A subset of these limitations has been addressed by FATMA. Since it reuses several building blocks of XPU, FATMA inherits some of the XPU limitations. These limitations can be summarized as follow:

- Limited skeleton nesting: XPU represents a program as a structured hierarchical task graph, each task of the graph can be replaced in turn by another hierarchical task graph ("*parallel/sequential*" skeleton) to express parallelism at finer grain. It can be also replaced by a skeleton that implement data parallelism (*parallel_for* construct) or a skeleton that implements pipeline parallelism such as the *pipeline* skeleton. While the hierarchical task graph allows hierarchical nesting without any limitation on the hierarchy depth, the two later skeletons do not allow "direct" nesting of other skeletons inside them.

- Task data dependencies extraction mechanism analyzes the signature (the parameters) of the target function or object method to determine its data dependencies. When the code of the function access to global variable instead of specifying that data in its arguments, the dependencies can be hidden from the XPU and FATMA runtime systems. The same problem can happen if the programmer hide a pointer inside a structure even when passed as an argument: the pointer to the structure can be seen by the runtime but the pointers hidden inside the structure cannot.

- Task granularity: in both XPU and FATMA programming models, the programmer is responsible of decomposing a program into of a set of task. The easiest way of decomposition is to define task using the existing functions. However these functions must have a reasonable granularity to be able to extract the available parallelism. In our experiments, most of C applications which are structured as a set of functions have often a satisfying granularities. However, in several cases, the program can exposes a too coarse-grained function. We note that too fine-grained functions do not expose the same problem since they can be encapsulated into tasks then grouped easily to form a coarse grain task-group.

- Static scheduling in FATMA: the current implementation of FATMA uses a static scheduling technique to schedule the task graphs, this allow FATMA to perform parallelism extraction "offline", e.g. before the execution, and thus to avoid introducing any task-dependency analysis overhead at the execution of the tasks. This is the major advantage of "static scheduling" technique. However, this scheduling mode has a limitation since it requires that no "inline" code is introduced between tasks, i.e, all code must be encapsulated into task and added to the task list in their natural sequential order. For example, if an code inline code must introduced due to data-dependent execution of a given application, the task list should be split into several sub-lists which can be then parallelized. Dynamic scheduling can alleviate this limitation but introduce an execution overhead which is related to "online" data dependency analysis. The late scheduling mode will be introduced in future version of FATMA.

11.3 Perspectives

The first and the last limitations of our frameworks are already technically resolvable using exclusively the C++ language and thus will be addressed in the future versions

of XPU and FATMA. However, addressing the second and the third limitation can be hard to resolve using exclusively the C++ language without making the API verbose. In this case, external tools such as syntax analyzer or source-to-source compilers can be used to perform the tasks that cannot be accomplished using exclusively the C++ language : syntax analysis tool can be used to perform advanced task data dependency analysis and address the case of "hidden dependencies", task granularity adjustment can be performed based on profiling data (execution time) or syntax analysis metrics (code length) using source-to-source compiler and custom profiling tools. These tools can address the limitation exposed earlier, and can relieve the programmer from adapting his code manually especially in the case of complex algorithms and applications. For instance, a memory access pattern analyzer such as the QUAD tool [OMGB10] can be used to extract automatically the data dependencies at functional level and track any "hidden" data dependency. The programmers code can then be completed by the explicit specification of missing dependencies in XPU or FATMA tasks.

In future works, XPU and FATMA will be merged to form a single programming model allowing both explicit and automatic parallelization, the resulting framework will be extended to support static and dynamic distributed memory systems. Our first experiments with the "Remote Task" implementation[1] appeared to be very promising, a deeper study of the communication, scheduling and performance models will be studied deeply. The (CHATS)[2] algorithm which we designed to perform cache hierarchy-aware scheduling on shared memory systems can be extended and generalized to perform memory hierarchy-aware scheduling (MHATS) on distributed memory systems.

Despite being at its early development stages, FATMA achieved encouraging results when used with popular tiled linear algebra algorithms. Particularly, FATMA addressed successfully easing static scheduling of super-scalar task graph on shared memory systems. Yet the current version of FATMA suffers from several implementation bottlenecks which can be fixed in later versions and we expect a significant performances improvement. FATMA performances can be improved by implementing advanced scheduling techniques such as cache-aware scheduling or using "*sliding window*" to limit the number of scheduled tasks when scheduling large task graph... Many of these techniques have been used successfully in concurrent approaches and appear to be very promising.

Finally, XPU has been used to implement industrial application such as the real-time "Radar Signal Processing" and the "Polyphase Filter Bank Processing" applications presented in this thesis [3]. XPU succeeded to achieve real-time execution in the first

[1] For more details, see Chapter 4 "Task Definition"

[2] Nader Khammassi and Jean-Christophe Le Lann, "***Design and Implementation Of A Cache Hierarchy-Aware Task Scheduling For Parallel Loops On Multicore Architectures***", Third International Conference on Parallel, Distributed Computing Technologies and Applications, PDCTA 2014, Sydney, Australia

[3] For more details, see *Part 4*: Chapter 9 and Chapter 10

application allowing the use of general-purpose multicore processors as an alternative to FPGA and DSP platforms. However, this kind of applications exposes hard real-time execution constraints which cannot be guaranteed by XPU, the provided wort-case execution time is fully based on experiments. In addition, performances of XPU in other platforms cannot be predicted easily before experimentation. In future implementation of XPU, a performance prediction model will be designed to provide a reliable performance prediction without need to heavy experimentation.

A

C++ Template Metaprogramming

A.1 C++ Templates

C++ Templates are Meta-Classes or Meta-Functions which constitute an efficient mean to handle type and parameter variability when respectively instantiating classes or calling functions that specifies common operations for different possible types of parameters. Templates allow the programmer to avoid duplicating a common code for different types or parameters, instead Templates allows him to specify a common reusable code for different types. When using Templates the compiler adapts transparently the template code to the selected type. Moreover, the compiler can perform various compile-time optimizations to produce efficient code.

A.1.1 Class Templates

Using a Meta-Class consists in defining a class which performs a set of operations on a generic abstract type. The programmer specifies the type when instantiate the template class for a specific type. For example, the C++ Standard Template Library (STL) [SL94] provides a set of containers which are class templates. These classes allow the programmer to use a container and its associated routines and algorithms disregarding whether it is a container of integers, of floats or of any other simple or complex type... Available containers in the C++ STL includes *vector*, *list*, *set*, *map*... etc

Listening A.1 shows an example of a generic class named table implemented as a template class. Thanks to template programming this class can implement a table of integer, floats, or other compatible types using a common code and interface.

```
1
2  /**
3   * generic table
4   */
5  template<class T, int N>
6  class table
7  {
8    T data[N];
9
```

```
10    T max()
11    {
12      T m = data[i];
13      for (int i=1; i<N; ++i)
14        if (data[i]>m)
15          m = data[i];
16      return m;
17    }
18  };
19
20  int main( )
21  {
22    table<float,16> float_table; /* table of ten float */
23    table<char,128> char_table; /* table of 128 characters */
24
25    float f = float_table.max(); /* find the maximum in the float table */
26    char  c = char_table.max(); /* find the maximum in the character table */
27
28    return 0;
29  }
```

Listing A.1: Example of generic "table" implementation using template class allows the use of a common interface and code for different types of data : float, charcter or other types.

Template Classes do not adapt only to type specification but also to value specification. For example the C++ STL library implements a container named *bitset* which allows a programmer to define the number of bits in that bit set through an integer template argument. So, *bitset<16>* is a *bitset* containing 16 bits while *bitset<32>* is a bitset containing 32 bits etc... Analogously, in our example in Listening A.1, the second parameter of the "table" template class specifies explicitly the size of the table. This allows the compiler to perform many optimization at compile time such as stack allocation for table data or loop unrolling in the "max" routine.

A.1.2 Function Templates

Similarly to class templates, Meta-Functions are functions which are able to operate on arguments disregarding their types. As shown in Listening A.2 a common meta-function which perform an addition between two parameters or arguments is able to perform this operation whenever the two passed arguments are integers, float or double... The compiler adapts automatically the function according to the arguments passed by the programmer when calling that function.

```
1
2  /**
```

```
 * generic addition function
 */
template<typename T>
T add(T x, T y)
{
   return (x+y);
}

int main( )
{
  float f, xf, yf;
  int   i, xi, yi;

  f = add(xf,yf); /* adding two floats */
  i = add(xi,yi); /* adding two integers */

  return 0;
}
```

Listing A.2: Template function "add" is able to perform and addition between integers, floats or other types.

A.1.3 Template Specialization

In addition to supporting varying types, template classes can be specialized to behave differently for specific types or values of the template arguments, i.e., it can be customized for specific types. Without specialization, the compiler use the same code for different types. When a template is specialized for specific type and that type is specified by the programmer, the specialized template is used instead of the original template. The specialized template can be different from the original template in many ways : a specialized template class can have different attributes or methods.

```
// default template class
template <class T>
class test
{
  public:
   test()
   {
       std::cout << "default template object \n";
   }
};

// specialized template for the "int" type
template <>
class test <int>
{
```

```
16 public:
17    Test()
18    {
19        std::cout << "specialized template object\n";
20    }
21 };
22
23
24 int main()
25 {
26
27     test<char> a;  // uses the default template class
28     test<int>  b;  // uses the specialized template class
29     test<float> c; // uses the default template class
30     return 0;
31 }
```

Listing A.3: An example of template class specialization

A template class can be specialized by fixing one or more of its template arguments and overloading completely or partially its routines to modify its behavior for the case of specific type. Listening A.3 shows an example of template class specialization. In this example, the default template class implemented for general types is specialized for the "int" type. When the "int" type is used as template parameter, the specialized implementation is used.

A.1.4 C++ Template Metaprogramming

C++ template Meta-Programming is a powerful programming technique which allows the programmer to perform static programming, i.e. programming the program. In other words, thanks to templates, programmer can tell the compiler to execute some operations or computation at compile-time when all the parameters of these computations are known at compile-time. These compile-time evaluations can reduce significantly many potential execution overheads which can be introduced if these evaluations are made at run-time.

In addition to producing highly efficient code, Meta-Programming techniques provide a great programming flexibility particularly for library designers who design generic template libraries for a wide variety of users. By using various meta-programming techniques, they make their libraries adapt easily to various context while maintaining a high execution efficiency. The provided templates can be considered as reusable skeletons which can be easily parametrized through template arguments. The C++ STL library is a great example of such libraries: for instance, STL containers and algorithms are widely used and highly efficient.

Template Meta-Programming can be used to perform many useful operations at compile-time such as : partial or complete evaluation (factorial computation), design meta-control structure (if,then...), compile-time recursive operation (Lanczos, Factorial, Prime...), loop unrolling ("for" loop unrolling), compile-time type identification (CTTI), introspection... etc

Listening A.4 shows the static implementation of the factorial computation using meta-programming techniques including template recursion and template specialization. The factorial of a given number is evaluated at compile-time by the compiler. At the opposite of the traditional implementation using a recursive function, this implementation do not introduce any execution overhead at run-time. Thus, this static implementation is useful when the number that we want to compute its factorial is known at compile-time.

```
#include <iostream>

// recursive template
template <int N>
struct factorial
{
    enum { value = N * factorial<N-1>::value };
};

// template specialization for the terminal state
// to stop recursion
template <>
struct factorial<1>
{
    enum { value = 1 };
};

// example use
int main()
{
    const int fact5 = Factorial<5>::value; // factorial of 5 is computed at compile-time
    std::cout << fact5 << endl;
    return 0;
}
```

Listing A.4: Static implementation of the "factorial" computation using C++ metaprogramming technique: the factorial is evaluated at compile-time by the compiler.

A.1.5 Limitations and Drawbacks

While offering great flexibility and execution effectiveness, C++ Template Meta-Programming can lead to unreadable, verbose and error-prone code and may result into poor programmability. In fact, when not properly used, C++ templates are known to generate

extremely verbose compilation errors. Moreover, extensive use of templates can make the exposed programming interface "unnatural": for instance, when using some modules of the Boost library [Sch11], it can be hard to recognize the resulting code as C++ code.

For these reasons, when using C++ metaprogramming in the implementation libraries, the programmer should take care of making the exhibited programming interface elegant and natural to take advantage from C++ Template Metaprogramming benefits without sacrificing programmability and intuitiveness.

B

Platforms Used For Testing XPU and FATMA

B.1 Single Multicore Processor

- Intel Core i7 Q720 (4x Cores, 8x Threads, 4x L1 Cache 32KB, 4x L2 Cache 256KB, 1x L3 Cache 6MB, 1.6 GHz). This platform is equipped with a GPU ("NVIDIA GeFORCE GT240M" with 48 CUDA Cores and OpenCL support).

B.2 Bi-Processor Platform

- 2 x AMD Opteron 252 K8 (1x Core, 1x Thread, 1x L1 Cache 64KB, 1x L2 Cache 1MB, 2.8 GHz).
- 2 x Intel Xeon X5472 (4x Cores, 4x Threads, 4x L1 Cache 32KB, 2x L2 Cache 6MB, 3 GHz).
- 2 x Intel Nehalem Xeon E5620 (4x Cores, 8x Threads, 4x L1 Cache 128KB, 4x L2 Cache 1MB, 1x L3 Cache 12MB, 2.4 GHz). This platform has two GPUs (2 x ATI Radeon 6970. Each GPU has 2GB GDDR5 memory and a theoretical compute power of 2.7 TFLOPS (Simple Precision) and OpenCL Support)

B.3 Quad-Processor Platform

SMP Supermicro station with 4 x CPU AMD Opteron 6274 16-Core 2.2 Ghz 16MB making a total of 64 Cores. The platform is equipped with a GPU "MSI GTX 280 2 GB GDDR5". The platform has 32 GB of RAM (ECC Registred at 1600 Mhz).

B.4 The CAPARMOR Supercomputer

A part of our test has been performed on the different CAPARMOR nodes. The CAPARMOR Supercomputer is an SGI Altix ICE 8200 EX composed of 294 nodes and 5 Racks. Most of the Nodes (256) are bi-processors board with 2 x "Intel Xeon X5560 Quad Core at 2.8 GHz". The remaining 38 Nodes are bi-processors board with 2 x "Intel Xeon X5677 Quad Core at 3.46 GHz". Therefore the CAPARMOR Supercomputer is

composed of 1352 Cores and display a theoretical compute power of 27 TFLOPS. CAPARMOR Nodes are connected through an Infiniband network (4xDDR and 4xQDR). Each node has a 24 GB memory, so the global memory size of CAPARMOR is 7 Tera Bytes.

B.5 The BADIANE Cluster

The Badiane Cluster is composed of 8 Nodes, each Node is a bi-processors board with two "Intel Xeon Sandy Bridge E5-2670 at 2.6 Ghz" (8 Cores, 20MB L3 Cache). Nodes are connected through an Infiniband network (40 Gb/s).

Bibliography

[AAB+02] E. Alba, F. Almeida, M. J. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. M. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 927–932, London, UK, UK, 2002. Springer-Verlag.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[ACD+09] E. Ayguade, N. Copty, A Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, March 2009.

[ACE+12] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, G. Péan J. O. Mcmahon, F. Pasquier, and P. Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing, Jan 2012.

[AD07] M. Aldinucci and M. Danelutto. The Cost of Security in Skeletal Systems. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, pages 213–220, Washington, DC, USA, 2007. IEEE Computer Society.

[ADD07] M. Aldinucci, M. Danelutto, and P. Dazzi. MUSKEL: An Expandable Skeleton Environment , 2007.

[ADK09] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic Management of Non-functional Concerns in Distributed & Parallel Application Programming. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[ADKT12] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, page 13, 2012.

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Gener. Comput. Syst.*, 19(5):611–626, July 2003.

[AG03] M. Alt and S. Gorlatch. Using Skeletons in a Java-Based Grid System. In H. Kosch, L. Boszormenyi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 742–749. Springer Berlin Heidelberg, 2003.

[AG05] M. Alt and S. Gorlatch. Adapting Java RMI for Grid Computing. *Future Gener. Comput. Syst.*, 21(5):699–707, May 2005.

[AM11] Khronos Group A. Munshi. The OpenCL specification version 1.1, 2011.

[AMD09] AMD. AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions, 2009.

[App20] Appentra. Parallware, http://www.appentra.com/products/parallware/, 20.

[ATNW11] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, 2011.

[BBK+08] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In Laurie Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin Heidelberg, 2008.

[BCD+97] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An Environment for Structured Parallel Programming. In Lucio Grandinetti, Janusz Kowalik, and Marian Vajtersic, editors, *Advances in High Performance Computing*, volume 30 of *NATO ASI Series*, pages 219–234. Springer Netherlands, 1997.

[BCGH05] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with Eskel. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 761–770, Berlin, Heidelberg, 2005. Springer-Verlag.

[BDF+06] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Apr 2006.

[BDM09] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[BDPV99] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: A Heterogeneous Environment for HPC Applications. *Parallel Comput.*, 25(13-14):1827–1852, December 1999.

[BE92] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(6):643–656, November 1992.

[BEF+95] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 141–154, London, UK, UK, 1995. Springer-Verlag.

[Ben99] S. Benkner. VFC The Vienna Fortran Compiler. *Sci. Program.*, 7(1):67–81, Jan 1999.

[BHC+93] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-parallel Language. *SIGPLAN Not.*, 28(7):102–111, July 1993.

[BHL+09] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[BHRS08] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[BK96] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 243–, Washington, DC, USA, 1996. IEEE Computer Society.

[BKSL08] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[Ble90] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[Ble96] G. E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[BLMP97] S. Breitinger, R. Loogen, Y. O. Mallen, and R. Pena. The Eden Coordination Model for Distributed Memory Systems. In *Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, HIPS '97, pages 120–, Washington, DC, USA, 1997. IEEE Computer Society.

[BMA+02] S. Bromling, S. MacDonald, J. Anvik, J. Schaefer, D. Szafron, and K. Tan. Pattern-based parallel programming. In *Proceedings of the International Conference on Parallel Programming*, ICPP'2002, pages 257–265, New York, NY, USA, August 2002. ACM.

[BMBW00] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGOPS Oper. Syst. Rev.*, 34(5):117–128, November 2000.

[BRS07a] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, Mar 2007.

[BRS07b] U. Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, Oct 2007.

[Can87] J. F. Canny. Readings in Computer Vision: Issues, Problems, Principles, and Paradigms. chapter A Computational Approach to Edge Detection, pages 184–203. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[CCZ07a] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[CCZ07b] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[Cen] Barcelona Supercomputing Center. SMP Superscalar, http://www.bsc.es/computer-sciences/programming-models/smp-superscalar.

[Cen11] Barcelona Supercomputing Center. SMP Superscalar (SMPSs) User's Manual Version 1.0, 2011.

[CGH13] Q. Chen, M. Guo, and Z. Huang. Adaptive Cache Aware Bitier Work-Stealing in Multisocket Multicore Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 24(12):2334–2343, December 2013.

[CGK+07] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

[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. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[CKK+08] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and Tong Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP '08. 37th International Conference on Parallel Processing, 2008.* , pages 536–545, Sept 2008.

[CL07] D. Caromel and M. Leyton. Fine Tuning Algorithmic Skeletons. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 72–81, Berlin, Heidelberg, 2007. Springer-Verlag.

[CL08] D. Caromel and M. Leyton. A transparent non-invasive file data model for algorithmic skeletons. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, pages 1–10, April 2008.

[CM11] C. Campbell and A. Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures.* Microsoft Press, 1st edition, 2011.

[CMV+06] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP31. *Parallel Comput.*, 32(7):539–550, September 2006.

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[Col04] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.*, 30(3):389–406, March 2004.

[Cor01] Intel Corporation. The IA-32 Intel architecture software developer's manual, 2001.

[Cor14] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference, 2014.

[Cou13] L. Courtès. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. Research Report RR-8278, INRIA, April 2013.

[CPK09] P. Ciechanowicz, M.l Poldner, and H. Kuchen. The Munster Skeleton Library Muesli: A comprehensive overview, 2009.

[Dan05] M. Danelutto. QoS in Parallel Programming Through Application Managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '05, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.

[DBM+09] C. Dave, H. Bae, S.J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, December 2009.

[DBQODS07] A. J. Dorta, J. M. Badía, E. S. Quintana-Ortí, and F. De Sande. Parallelizing Dense Linear Algebra Operations with Task Queues in Llc. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, pages 89–96, Berlin, Heidelberg, 2007. Springer-Verlag.

[DG04] J. Dunnweber and S. Gorlatch. HOC-SA: A Grid Service Architecture for Higher-Order Components. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, SCC '04, pages 288–294, Washington, DC, USA, 2004. IEEE Computer Society.

[DGRDS03] A. J. Dorta, J. A. González, C. Rodríguez, and F. De Sande. llc: a Parallel skeletal language. In *Proc. of the Second International Workshop on High Level Parallel Programming and Applications (HLPP 2003)*, page 77–88, Paris, France, Jun 2003.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 19–28, New York, NY, USA, 1995. ACM.

[Dij79] E. Dijkstra. Classics in Software Engineering. chapter Go to Statement Considered Harmful, pages 27–33. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[DRVV00] A. Darte, Y. Robert, F. Vivien, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser Boston, 2000.

[DS00] M. Danelutto and M. Stigliani. SKElib: Parallel Programming with Skeletons in C. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 1175–1184, London, UK, UK, 2000. Springer-Verlag.

[DT95] J. Darlington and H. W. To. Abstract Machine Models for Highly Parallel Computers. chapter Building Parallel Applications Without Programming, pages 140–154. Oxford University Press, Oxford, UK, 1995.

[EGF+12] P. Estérie, M. Gaunard, J. Falcou, J.T. Lapresté, and B. Rozoy. Boost.SIMD: Generic Programming for Portable SIMDization. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 431–432, New York, NY, USA, 2012. ACM.

[EGFL12] P. Estérie, M. Gaunard, J. Falcou, and J.T. Lapresté. Exploiting Multimedia Extensions in C++: A Portable Approach. *Computing in Science and Engineering*, 14(5):72–77, 2012.

[EK10] J. Enmyren and C. W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of 4th Int. Workshop on High-Level Parallel Programming and Applications*, HLPP'2010, New York, NY, USA, September 2010. ACM.

[Eva] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD, http://www.canonware.com/jemalloc/.

[FJ05] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[FSP06] J. F. Ferreira, J. L. Sobral, and A. J. Proenca. JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing*. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 301–304, Washington, DC, USA, 2006. IEEE Computer Society.

[GJP10] C. Grelck, J. Julku, and F. Penczek. S-Net for Multi-memory Multicores. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 25–34, New York, NY, USA, 2010. ACM.

[GM] S. Ghemawat and P. Menage. TCMalloc : Thread-Caching Malloc, http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[GPB+07] J. Giacomoni, G. Price, B. Bushnell, M. Vachharajani, and D. Grunwald. Toward a toolchain for pipeline parallel programming on CMPs. In *In Workshop on Software Tools for Multi-Core Systems*, 2007.

[Gre99] C. Grelck. Shared Memory Multiprocessor Support for SAC. In *Selected Papers from the 10th International Workshop on 10th International Workshop*, IFL '98, pages 38–53, London, UK, UK, 1999. Springer-Verlag.

[Gre05] C. Grelck. Shared Memory Multiprocessor Support for Functional Array Processing in SAC. *J. Funct. Program.*, 15(3):353–401, May 2005.

[GV06] H. González-Vélez. Self-adaptive Skeletal Task Farm for Computational Grids. *Parallel Comput.*, 32(7):479–490, September 2006.

[GVC07] H. González-Véléz and M. Cole. Adaptive Structured Parallelism for Computational Grids. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 140–141, New York, NY, USA, 2007. ACM.

[GVC08] H. Gonzalez-Velez and M. Cole. An adaptive parallel pipeline pattern for grids. In *IPDPS 2008. IEEE International Symposium on Parallel and Distributed Processing, 2008.*, pages 1–11, April 2008.

[GVL10] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[HB] Hoberock and N. Bell. Thrust, http://code.google.com/p/thrust/.

[Her03] J. Herrington. *Code Generation in Action.* Manning Publications Co., Greenwich, CT, USA, 2003.

[Hir12] E.C. Hiram. *Openhmpp.* Placpublishing, 2012.

[HL00] C. A. Herrmann and C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer, 2000.

[HLJH09] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart. Stream Compaction for Deferred Shading. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 173–180, New York, NY, USA, 2009. ACM.

[HTHW14] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging

Application on Modern Multi- and Manycore Chips. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 57–64, New York, NY, USA, 2014. ACM.

[IHS06] M. Ishihara, H. Honda, and M. Sato. Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: IPat/OMP. *IEICE - Trans. Inf. Syst.*, E89-D(2):399–407, February 2006.

[Inta] Intel. Automatic Parallelization with Intel Compilers, https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers.

[Intb] Intel. Intel Core i7-720QM, http://ark.intel.com/products/43122/.

[Intc] Intel. Intel Xeon E5620, http://ark.intel.com/products/47925/.

[JL09] N. Javed and F. Loulergue. OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, APPT '09, pages 436–451, Berlin, Heidelberg, 2009. Springer-Verlag.

[Jos13] V. G. Joshi. A study of possible optimizations for the task scheduler "QUARK" on the shared memory architecture. Master's thesis, University of Tennessee, Knoxville, TN, USA, 2013.

[KAG+09] L.J. Karam, I. AlKamal, Alan Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. Trends in multicore DSP platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, November 2009.

[KB09] Hahn K. and R. Bond. Multicore software technologies. *Signal Processing Magazine, IEEE*, 26(6):80–89, November 2009.

[KLLDS12] N. Khammassi, J.C. Le Lann, J.P. Diguet, and A. Skrzyniarz. MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication*, HPCC '12, pages 71–80, Washington, DC, USA, 2012. IEEE Computer Society.

[KLPR01] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden: Low-Effort Parallel Programming. In *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, IFL '00, pages 71–88, London, UK, UK, 2001. Springer-Verlag.

[KLY+14] J. Kurzak, P. Luszczek, A. Yarkhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. Multithreading in the PLASMA Library. In Sanguthevar Rajasekaran Mohamed Ahmed, Reda A. Ammar, editor, *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*. Chapman and Hall/CRC, March 2014.

[Kob] Petr Kobalicek. Complete x86/x64 JIT and Remote Assembler for C++, https://github.com/kobalicek/asmjit.

[Kos] J. Koskinen. Metaprogramming in C++, www.cs.tut.fi/ kk/web-stuff/MetaprogrammingCpp.pdf.

[KRS+08] H. Kim, E. Rutledge, S. Sacco, S. Mohindra, M. Marzilli, J. Kepner, R. Haney, J. Daly, and N. Bliss. PVTOL: Providing Productivity, Performance and Portability to DoD Signal Processing Applications on Multicore Processors. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 327–333, July 2008.

[Lam74] L. Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2):83–93, February 1974.

[Lee06] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[Lei09] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[LKHR05] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge. Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing. *Proceedings of the IEEE*, 93(2):313–330, Feb 2005.

[LLM08] G. Lashari, O. Lhoták, and M. McCool. Control Flow Emulation on Tiled SIMD Architectures. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 100–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[LOmPnm05] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel Functional Programming in Eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[LP10] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 289–296, Washington, DC, USA, 2010. IEEE Computer Society.

[LSB09] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[Lyo10] R.G. Lyons. *Understanding Digital Signal Processing*. Prentice-Hall accounting series. Pearson Education, 2010.

[Mac08] D. Mackenzie. *An Engine, Not a Camera: How Financial Models Shape Markets*. Inside Technology. MIT Press, 2008.

[McC10] M. D. McCool. Structured Parallel Programming with Deterministic Patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[MCG03] M. Müller, D. Charypar, and M. Gross. Particle-based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Mica] Microsoft. Processor Information, http://msdn.microsoft.com/en-us/library/windows/desktop/ms683194

[Micb] Microsoft. Task Parallel Library, http://msdn.microsoft.com/en-us/library/dd537608.aspx.

[MIEH06] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[MSM04] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[MSS04] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the Pipeline as Object-Oriented States with Transformations. In *HIPS*, pages 12–21, 2004.

[MWHL06] M. D. McCool, K. Wadleigh, B. Henderson, and H.Y. Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[NBF96] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[OMGB10] S. A. Ostadzadeh, R.J. Meeuws, C. Galuzzi, and K. Bertels. QUAD: A Memory Access Pattern Analyser. In *Proceedings of the 6th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'10, pages 269–281, Berlin, Heidelberg, 2010. Springer-Verlag.

[Para] D. Parnham. Gimp Plugin Code, https://github.com/parnham/plugins.

[Parb] D. Parnham. Openillusionist Project, http://sourceforge.net/projects/openillusionist.

[Par07] D. Parnham. *An Infrastructure for Video-Augmented Environments.* PhD thesis, University of New York, New York, NY, USA, Feb. 2007.

[PG92] D. Pnueli and C. Gutfinger. *Fluid Mechanics.* Cambridge University Press, 1992.

[Phe08] C. Pheatt. Intel&Reg; Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[PM07] J.G. Proakis and D.G. Manolakis. *Digital Signal Processing.* Prentice Hall international editions. Pearson Prentice Hall, 2007.

[PRZ05] D.J. Parnham, J.A. Robinson, and Y. Zhao. A compact fiducial for affine augmented reality. In *Proceedings of the 2005 IEEE International Conference on Visual Information Engineering* , VIE'05, pages 347–352, 2005.

[PW96] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[RDAS09] R. Reyes, A. J. Dorta, F. Almeida, and F. Sande. Automatic Hybrid MPI-OpenMP Code Generation with Llc. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 185–195, Berlin, Heidelberg, 2009. Springer-Verlag.

[Rob13] A. D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, March 2013.

[RPK00] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, July 2000.

[Sch11] B. Schling. *The Boost C++ Libraries.* XML Press, 2011.

[SG02] J. Sérot and D. Ginhac. Skeletons for Parallel Image Processing: An Overview of the SKIPPER Project. *Parallel Comput.*, 28(12):1685–1708, December 2002.

[Sin04] H. Singh. *Introspective C++.* PhD thesis, Virginia Polytechnic Institute, Virginia, VA, USA, 2004.

[SL93] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, February 1993.

[SL94] A. Stepanov and M. Lee. The Standard Template Library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[SL02] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2002.

[Sol] Northrop Grumman Corporation IT Solutions. CLOC - Count Lines Of Code v1.53, http://cloc.sourceforge.net.

[SOW+95] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[ST98] D.B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[Sta03] J. Stam. Real-Time Fluid Dynamics for Games, 2003.

[SZ08] J. E. Savage and M. Zubair. A Unified Model for Multicore Architectures. In *Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies*, IFMT '08, pages 9:1–9:12, New York, NY, USA, 2008. ACM.

[TAS07] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, March 2007.

[TSS+03] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. *SIGPLAN Not.*, 38(10):203–215, June 2003.

[TTTn+09] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC Programmability Using Classroom Studies. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 10:1–10:7, New York, NY, USA, 2009. ACM.

[Van02] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.

[VCJC+13] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[VSG+12] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell. SESAM/Par4All: A Tool for Joint Exploration of MPSoC Architectures and Dynamic Dataflow Code Generation. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 9–16, New York, NY, USA, 2012. ACM.

[WJNB95] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[Wol09] W. Wolf. Multiprocessor System-on-Chip Technology. *Signal Processing Magazine, IEEE*, 26(6), November 2009.

[Yar12] A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures.* PhD thesis, The Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, TN, USA, 2012.

[YLY10] T.F. Yang, C.H. Lin, and C.L. Yang. Cache-aware task scheduling on multi-core architecture. In *International Symposium on VLSI Design Automation and Test (VLSI-DAT), 2010* , pages 139–142, April 2010.

List of Figures

2.1 Sequence . . . 25
2.2 Selection . . . 25
2.3 Iteration . . . 25
2.4 The Map Pattern . . . 28
2.5 The 2D Stensil Pattern . . . 28
2.6 Reduction Pattern . . . 29
2.7 Serial Scan Pattern . . . 30
2.8 Pack Pattern . . . 31
2.9 Fork-Join Pattern . . . 32
2.10 Super Scalar Task Graph Pattern . . . 33

3.1 Program can be decomposed into a set of tasks at different granularity levels . . . 47
3.2 In ideal parallel program, finest grain tasks can be executed simultaneously. 48
3.3 Realistic parallel program contains both parallel and sequential tasks at different granularity level due to the potential dependencies between some tasks. . . . 49
3.4 Sequential Execution . . . 50
3.5 Parallel Execution . . . 50
3.6 Data Parallel Execution . . . 50
3.7 Pipeline Execution . . . 50
3.8 Parallel program combining several types of parallelism at different granularity levels. . . . 50
3.9 Overview of the XPU Architecture . . . 51
3.10 Different implementations of the Hierarchical Task Group Graph . . . 53
3.11 Sequential task execution details. . . . 54
3.12 Parallel task execution details. . . . 55
3.13 Parallel loop execution details. . . . 56
3.14 Pipeline execution details. . . . 57

4.1 Task design allows reusing different pieces of code through an extendable set of implementations . . . 65
4.2 Task data dependencies detection . . . 68
4.3 Service-Based Infrastructure For Executing Remote Tasks on Dynamic Distributed Systems . . . 72
4.4 Implementation details of the XPU Computing Task . . . 74

5.1 Unlike the superscalar task graph representation (at the right), the hierarchical task graph representation (at the left) introduces join points (barriers) where parallel tasks are synchronized. This can introduce unnecessary idle times at the execution time. . . . 78
5.2 "sequential" specifies the sequential execution of a set of tasks. . . . 80

5.3 The "parallel" construct specifies the simultaneous execution of a set of tasks. . . . 81
5.4 Task graph with both parallel and sequential regions can be specified by combining "parallel" and "sequential". . . . 82
5.5 Race condition does not occur in the first configuration since the shared data (data 3) is accessed concurrently in read only mode. However, race condition can happen in the second configuration where concurrent writes are operated on the shared data. . . . 83
5.6 The shared data detection algorithm can be applied to all hierarchy levels of and hierarchical task graph. . . . 84
5.7 Design of the Lockable interface and its implementations. The Lockable Factory guarantee that a unique Lockable is associated to each data identified by its memory address. . . . 85
5.8 Task accessing concurrently to shared data are executed inside critical sections: the shared resource is locked before the task execution then unlocked when execution terminate. . . . 86
5.9 Example of a task graph that specifies parallelism and data dependencies. 87
5.10 Comparison between XPU and TBB in term of required programming effort: required extra-code and reuse of sequential code. . . . 88
5.11 Execution details of the "*sequential*" skeleton. . . . 89
5.12 Execution details of the "*parallel*" skeleton. . . . 90

6.1 Example of data parallelism implementation in an image processing application . . . 94
6.2 Basic partitioning of a data array. . . . 95
6.3 Simplified Architecture Overview of the Intel Dunnington Processor . . . 98
6.4 Simplified Architecture Overview of the Intel Hapertown Processor . . . 98
6.5 Simplified Architecture Overview of the Intel Nehalem Processor . . . 98
6.6 Worker Pool-Based Run-Time System With Private Work-Queue . . . 100
6.7 CHATS Data Partitioning on an 8 Hardware Threads Intel Nehalem Processor . . . 101
6.8 CHATS Data Partitioning on an on hybrid SMT-CMP-SMP platform with 16 Hardware threads (2 x Intel Xeon E5620 at 2.4 GHz) . . . 102
6.9 Static Scheduling In Theory . . . 103
6.10 Static Scheduling In Practice . . . 103
6.11 Dynamic Scheduling In Theory . . . 104
6.12 Dynamic Scheduling In Practice . . . 104
6.13 Task Stealing Scheduling In Theory . . . 105
6.14 Task Stealing Scheduling In Practice . . . 105
6.15 Average processing time of different workload sizes. . . . 107
6.16 Cache-miss rate for different problem sizes. . . . 107
6.17 Traditional scalar operation on single data versus simultaneous SIMD operation on multiple data . . . 108

6.18 SIMD Operations using 128 bits-wide XMM register in single precision and double precision float configurations 109
6.19 Vectorization can improve substantially floating operation throughput in comparison with traditional operations 110
6.20 Functions vectorization offers significant floating operation throughput improvement in comparison with standard C math library functions implementations 110
6.21 Example of incompressible fluid simulation based on the Navier-Stokes equation. We note that this simulation has been implemented using XPU and is based on an algorithm for real-time fluid animation which is described in [Sta03]. 112
6.22 The Navier-Stokes equation for a Newtonian incompressible fluid which formulates the conservation of momentum. 113
6.23 Smoothed particle fade with distance. 113
6.24 The SPH algorithm derives a scalar quantity A at location r by computing a weighted sum of all particles. 114
6.25 The SPH algorithm derives a scalar quantity A at location r by a weighted sum of all particles. 114
6.26 Every simulation step, the algorithm executes five kernels. 114
6.27 The three dimensions scene is represented as a box which is divided in small cells. These cells can be regrouped into blocks on which concurrent threads can operate. 116
6.28 In the parallel implementation of the "*fluidanimate*" application, each kernel is executed simultaneously on multiple data partitions. 117
6.29 Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with two AMD Opteron 252 K8 (L1 64 KB, L2 1M, 2.8 GHz). 121
6.30 Execution time of the "Fluid Animation" application for different problem sizes on hybrid SMT-CMP Nehalem Processor (Intel Core i7 Q720) with 8 Hardware threads. 122
6.31 Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with 2 x Core 2 X5472 Processor (4 Cores L1 32KB, L2 6MB, 3 GHz 122
6.32 Execution time of the "Fluid Animation" application for different problem sizes on a multiprocessor platform with 2 x Intel Xeon E5620 processors.123
6.33 Programmability Comparison of the "Fluid Animation" application: Line count of the sequential version and the parallel versions using XPU, PThreads and TBB. 123
6.34 "*Blackscholes*" execution time on Intel Core 2 Duo E8600 Processor (2 Cores) for different problem size. 129
6.35 "*Blackscholes*" execution time on 8 Threads Intel Core i7 Q720 Processor 129
6.36 "*Blackscholes*" execution time on a 16 Threads SMP platform with two Intel Xeon E5620 Processor at 2.4 GHz 130
6.37 Black-Scholes Scalability (2 to 16 Hardware Threads). 130

6.38 Black-Scholes Programmability Comparison: Line count of the sequential version and the parallel versions using XPU (vectorized), OpenMP, PThreads and TBB. . . . 131
6.39 Black-Scholes Programmability Comparison: Line count of the sequential version and the vectorized parallel versions using XPU (Vectorized), OpenMP+SSE, PThreads+SSE. . . . 131

7.1 Pipeline Execution Pattern: Consumer/Producer relationship between pipeline stages. . . . 134
7.2 Serial Execution: Stage 1 and 2 are executed sequentially on the same processor. . . . 134
7.3 In Pipeline Execution Pattern, only dependent tasks are serialized. . . . 135
7.4 Execution of three stage pipeline which process two item of interest. . . . 140
7.5 Three types of unit work: first stage execute event notifier work, intermediate stages execute event relay work and last stage execute event listener work . . . 141
7.6 Thread-Based Run-Time System : A thread is created for each pipeline stage, the main thread wait for these threads to process all available data. 142
7.7 In the load-balanced scheduler implementation, task are submitted to a pool of *workers* through shared *work queue*, idle *workers* pickup the tasks from the work queue. Therefore, the load is constantly and dynamically balanced. . . . 143
7.8 Worker Pool-based run-time system with private Work Queue. . . . 144
7.9 The Adaptive Threshold Edge detection Algorithm. . . . 145
7.10 Lines count of the original sequential version and the required parallelism-related extra-code in the XPU and TBB version. . . . 146
7.11 Levenshtein distance (arbitrary unit) between original sequential version and respectively XPU and TBB parallelized versions. . . . 147
7.12 Performed frame rate (fps) by sequential and pipeline processing with different scheduling techniques on an 8 threads processor (highest best). 147
7.13 Average achieved frames rate (frame/sec) on an 8 threads processor (Intel Core i7 Q720). . . . 148
7.14 Average achieved frame rate (frames/sec) on a 16 hardware threads platform (SMP with 2 x Intel Xeon E5620 at 2.4 GHz). . . . 148

8.1 The task dependency graph of the tiled Cholesky factorization algorithm for a 5x5 tiles decomposition. . . . 154
8.2 In many tiled linear algebra algorithm, the number of tasks grows as $O(n^3)$ with the number of tiles. While generating 35 tasks for 15 tiles configuration, the Cholesky factorization algorithm generates 4960 tasks for 900 tiles decomposition. . . . 154
8.3 The fork-join execution model used by XPU, TBB, Cilk Plus, OpenMP... generates unnecessary idles times when executing certain Task Graphs (DAG). . . . 155

8.4 The super-scalar execution model used by FATMA, SMPSS or Quark executes asynchronously the tasks and use event-based peer-to-peer synchronization model between dependent task. This allows FATMA to eliminate unnecessary idles times when executing Task Graphs (DAG). . 155
8.5 The FATMA code introduce minor parallelism-related extra-code (highlighted in red) while preserving the legacy code from any significant alteration. 157
8.6 The automatic parallelization process start from a specification of a sequence of task to generate transparently the task dependency graph then use it to schedule asynchronously these tasks. 158
8.7 Tiled representation of N x N real matrix, in this example the matrix is decomposed into 3 x 3 tiles, each tile is a NB x NB submatrix. 159
8.8 Result of the Application of the Virtual Task Execution Tracing process to the sequence of tasks. 170
8.9 Task Execution Back-Tracing exploits the simulated execution traces to extract dependencies between tasks represented by the red direct edges. 171
8.10 The Directed Acyclic Graph (DAG) corresponding to the example program. 172
8.11 Comparison between the parallel implementation of the tiled linear system solver (dgesv) using the Thread-based infrastructure and the Intel Core i7 173
8.12 Event-based implementation of the XPU Pipeline skeleton. 174
8.13 Event-based implementation of the FATMA super scalar task graph. . . 174
8.14 Comparison between FATMA, QUARK and SMPSs implementations of the tiled Cholesky factorization on and 8 Threads Intel Core i7 Q720 processor. 176
8.15 LU Factorization 177
8.16 Linear Solver 177
8.17 Merged Algorithms 177
8.18 Comparison between FATMA and PLASMA (Static Scheduling) implementations of the tiled *dgesv* on an SMP platform with 2 x Intel Xeon E5620 at 2.4 GHz (16 Hardware Threads). 178

9.1 Overview of the Radar signal processing algorithm. 183
9.2 Computing load of each processing block for a 64 channels-burst (Floating-point Operations). 185
9.3 Input/Output Data volume per burst for each processing block in a 64 channels configuration. 186
9.4 Execution time of the initial serial implementation for 16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz (16 Threads / 8 Cores). 187
9.5 Application Parallelization Methodology: Application is progressively parallelized at different levels of granularity by expressing different types of parallelism. 188
9.6 Execution time of both the sequential version and the task-parallel version for 8,16,32 and 64 Channels-Burst on 2 x Intel Xeon E5620 2.4 GHz. 190

9.7 Parallelization of the MTI block at instruction and thread level using the XPU vectorization and the parallel loop. 191
9.8 Parallelization of the COR block at instruction and thread level using the XPU vectorization and the parallel loop. 192
9.9 Execution time achieved by the parallelized version for 8,16,32 and 64 channels-burst on 2 x Intel Xeon E5620 2.4 GHz. 193
9.10 Execution time achieved by the parallelized version for 8,16,32 and 64 Channels-burst on an Intel Core i7 Q720 1.6 GHz 194
9.11 The radar processing algorithm is split into three balanced stages which executes concurrently in a pipeline execution pattern. Concurrent pipeline stages works on different bursts simultaneously. 194
9.12 The final parallel application is parallelized using different types of parallelism at different levels of granularity. 195
9.13 Worst and best execution time of the parallel version with pipeline parallelism version execution for 8,16,32 and 64 Channels-Burst on the Intel Core i7 Q720 1.6 GHz (8 Threads) 196

10.1 Non-Windowed FFT exposes poor out-of-band rejection due to "leakage" phenomenon which affect the nearby of the bin of interest. 199
10.2 Applying PFB processing before the FFT provides a flat response on the bin of interest with an excellent out-of-band rejection and alleviates both the "leakage" and "scalloping loss" phenomenons. 200
10.3 The frequency response of a Polyphase Filter Bank composed of a set of overlapping filters Polyphase Filter Bank Processing. 201
10.4 Overview of the polyphase filter bank application in its single channel configuration. 202
10.5 Computing Loads (GFLOP/S) respectively in the 1-Channel and 4-Channels Configurations 204
10.6 Both the filter bank stages and the detection stages are integrated as processing stages within a pipeline execution pattern. 206
10.7 The first detection block (LTI+STI) is executed outside the pipeline in an asynchronous fashion to avoid false dependency between (LTI+STI) and PFB2. 206
10.8 The initial pipeline-only configuration perform about three times faster that the original sequential configuration on an Intel Xeon W3680 with 12 Threads at 3.3 GHz. 207
10.9 Effect of vectorization in comparison on the original sequential version on an Intel Xeon W3680 at 3.3 GHz 208
10.10 Data parallelism in PFB0 is implemented through an XPU parallel for loop. 209
10.11 Parallelization of loops in PFB0 stages and LTI reduced execution time by more than 30% on an Intel Xeon W3680 with 12 Threads at 3.3 GHz. 209

10.12Both Task and Data parallelism are implemented in PFB1 by specifying parallelism between five filters and implementing each of these filters as a parallel for loop. 210
10.13LTI and STI can be executed as parallel independent tasks. Additionally, data parallelism inside LTI can be specified through an XPU parallel for loop. 210
10.14Implementation of Task Parallelism through parallelization of filters execution at each PFB stage allowed us to achieve a 31% execution speedup on the same platform (Intel Xeon W3680 with 12 Threads at 3.3 GHz). 212

List of Tables

2.1 Comparative table of the algorithmic skeleton frameworks 40

6.1 Description of Cache and Memory Hierarchy of some CMP Architectures in UMAM. 99