

Instituto Tecnológico de Costa Rica  
Escuela de Ingeniería Electrónica

Programa de Maestría en Ingeniería Electrónica



## **Sequential Code Parallelization for Multi-core Embedded Systems: A Survey of Models, Algorithms and Tools**

para optar por el título de  
*Magister Scientiae* en Ingeniería Electrónica  
énfasis en Sistemas Empotrados

con el grado académico de  
Maestría

Jorge Alberto Castro Godínez

Cartago, 15 de Diciembre del 2014



I declare that this thesis document has been made entirely by my person, using and applying literature on the subject, and introducing my own knowledge and experimental results.

In the cases I have used literature, I proceeded to indicate the sources by the respective references. Accordingly, I assume full responsibility for this thesis work and the content of this document.

Jorge Alberto Castro Godínez

Bühl, Germany. December 15, 2014

Céd.: 1 1236 0930

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.

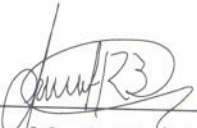


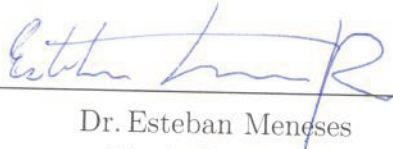



Instituto Tecnológico de Costa Rica  
Electronics Engineering School  
Master's Thesis  
Evaluation Committee

Master's Thesis presented to the Evaluation Committee as a requirement to obtain the Master of Science degree from the Instituto Tecnológico de Costa Rica.

Evaluation Committee Members

  
\_\_\_\_\_  
MSc. María Rodríguez  
Thesis Evaluator

  
\_\_\_\_\_  
Dr. Esteban Meneses  
Thesis Evaluator

  
\_\_\_\_\_  
MSc. Miguel Ángel Aguilar Ulloa  
Thesis Director

The members of the Evaluation Committee certify that this Master's Thesis has been approved and that fulfills the requirements set by the Electronics Engineering School.

Bühl, Germany. December 15, 2014



# Abstract

In recent years the industry experienced a shift in the design and manufacture of processors. Multiple-core processors in one single chip started replacing the common used single-core processors. This design trend reached the develop of System-on-Chip, widely used in embedded systems, and turned them into powerful Multiprocessor System-on-Chip. These multi-core systems have presented not only an improvement in performance but also in energy efficiency.

Millions of lines of code have been developed over the years, most of them using sequential programming languages such as C. Possible performance gains of legacy sequential code executed in multi-core systems is limited by the amount of parallelism that can be extracted and exploit from that code. For this reason, several tools have been developed to extract parallelism from sequential program and produce a parallel version of the original code. Nevertheless, most of these tools have been designed for high-performance computing systems rather than for embedded systems where multiple constraints must be considered, and a reduction in the execution time is not the only desirable objective.

Due there is no definitive solution for parallelizing code, especially for multi-core embedded systems, this work aims to present a survey on some different aspects involved in parallelizing code such as models of code representation, code analysis, parallelism extraction algorithms, parallel programming. Also existing parallelizing tools are presented and compared.

This work ends with a recommended list of important key aspects that should be consider when designing and developing a parallelizing compiler, automatic or semiautomatic, for multi-core embedded systems; and when using existing tools to use them.

**Keywords:** Parallelism extraction, multi-core embedded systems, representation models, code analysis, parallelizing algorithms, parallelization tools.





# Resumen

En los últimos años, la industria ha experimentado un cambio en el diseño y manufactura de procesadores. Procesadores con múltiples núcleos en un solo chip han reemplazado aquellos procesadores, comúnmente usados, con un solo núcleo. Esta tendencia de diseño ha alcanzado el desarrollo de sistemas en chip, ampliamente usados en sistemas embebidos, y los ha convertido en potentes sistemas en chip con múltiples procesadores. Estos sistemas multinúcleo no solo han presentado mejoras en el rendimiento sino también en la eficiencia energética.

Millones de líneas de código han sido desarrolladas a lo largo de los años, principalmente usando lenguajes de programación secuencial como C. Los potenciales beneficios de ejecutar código secuencial existente en sistemas multinúcleo están limitados por la cantidad de paralelismo que pueda ser extraído y explotado de estos códigos. Por esta razón, varias herramientas han sido desarrolladas para extraer paralelismo de programas secuenciales y producir una versión paralela del código original. Sin embargo, la mayoría de estas herramientas han sido diseñadas para sistemas computacionales de alto rendimiento en lugar de sistemas embebidos, donde múltiples restricciones deben ser consideradas, y una reducción del tiempo no es el único objetivo deseable.

Debido a que no existe una solución definitiva para la paralelización de código, particularmente para sistemas embebidos multinúcleo, este trabajo tiene como objetivo presentar un estudio sobre diferentes aspectos involucrados en la paralelización de código tal como los modelos de representación de código, análisis de código, algoritmos de extracción de paralelismo, programación paralela. Además, herramientas existentes son presentadas y comparadas.

Este trabajo concluye con una lista de recomendaciones al respecto de aspectos clave importantes que deben ser considerados al diseñar y desarrollar un compilador paralelizable, automático o semiautomático, para sistemas embebidos multinúcleo, así como cuando utilizar herramientas existentes.

**Palabras clave:** Extracción de paralelismo, sistemas embebidos multinúcleo, modelos de representación, análisis de código, algoritmos de paralelización, herramientas para paralelización.



*to Silvia & Fabiana*



# Acknowledgments

I am deeply grateful for the opportunity to continue with my formal higher education. It has been always my goal to obtain all the education possible. I truly believe that education is the gate to bigger opportunities in life.

I would like to thank to M.Sc. Miguel Aguilar, my thesis's director, that despite his personal challenges, continued willing to supervise me in this work. Also I thank to M.Sc. María Rodríguez and Dr. Esteban Meneses for being part of the evaluation committee.

I would like to thank the master's program coordination, in the persons of Dr.-Ing Paola Vega and M.Sc. Anibal Coto, for their support, help, and collaboration. Also, I thank to my colleagues at the Costa Rica Institute of Technology for their support and encouragement, especially to Dr.-Ing. Pablo Alvarado for his counsel and advice along the time of my master's studies.

I have to recognize and deeply thank to the MICITT (Ministerio de Ciencia, Tecnología y Telecomunicaciones), the CONICIT (Consejo Nacional para Investigaciones Científicas y Tecnológicas) and the Fondo de Incentivos, that made possible the funding for my master studies, by means of a scholarship.

Finally, and most important, I thank to my family, especially to Silvia, my wife, who always impulses and motivates me in our life adventures; and to Fabiana, my daughter, who inspires me to do my best everyday.

Jorge Alberto Castro Godínez

Bühl, Germany. December 15, 2015



# Contents

List of Figures	v
List of Tables	vii
List of Abbreviations	ix
<b>1 Introduction</b>	<b>1</b>
1.1 The sequential code problem . . . . .	5
1.2 Parallelism extraction . . . . .	6
1.3 Scope of this work . . . . .	7
1.4 Outline . . . . .	8
<b>2 Background</b>	<b>9</b>
2.1 Amdahl's law . . . . .	9
2.2 Dependencies in code . . . . .	10
2.2.1 Data dependencies . . . . .	11
2.2.2 Control dependencies . . . . .	12
2.3 Parallelism: types and relationships . . . . .	13
2.3.1 Granularity and parallel performance . . . . .	13
2.3.2 Data Level Parallelism . . . . .	14
2.3.3 Instruction Level Parallelism . . . . .	15
2.3.4 Thread Level Parallelism . . . . .	16
2.3.4.1 Task Level Parallelism . . . . .	17
2.3.4.2 Pipeline Level Parallelism . . . . .	18
2.4 Parallelizing compilers . . . . .	19
2.5 Embedded Systems and High-Performance Computing . . . . .	22
2.5.1 Parallelizing for ES and HPC . . . . .	24
<b>3 Representation Models</b>	<b>27</b>
3.1 Data Flow Graph . . . . .	27
3.2 Data Dependence Graph . . . . .	29
3.3 Control Flow Graph . . . . .	29
3.4 Dependence Flow Graph . . . . .	32
3.5 Control Data Flow Graph . . . . .	35
3.6 Program Dependence Graph . . . . .	36

3.6.1	Augmented Program Dependence Graph . . . . .	38
3.6.2	Parallel Program Graph . . . . .	38
3.7	Hierarchical Task Graph . . . . .	40
3.7.1	Augmented Hierarchical Task Graph . . . . .	41
3.8	System Dependence Graph . . . . .	42
<b>4</b>	<b>Code Analysis and Parallelizing Algorithms</b>	<b>47</b>
4.1	Code analysis . . . . .	47
4.1.1	Static . . . . .	47
4.1.2	Dynamic . . . . .	50
4.1.3	Hybrid . . . . .	52
4.2	Algorithms . . . . .	53
4.2.1	Machine learning . . . . .	53
4.2.2	Integer Linear Programming . . . . .	54
4.2.3	Thread extraction algorithm . . . . .	56
4.2.4	Genetic Algorithms . . . . .	57
<b>5</b>	<b>Parallel Programming Models and Parallelism Extraction Tools</b>	<b>61</b>
5.1	Parallel programming models . . . . .	62
5.1.1	OpenMP . . . . .	63
5.1.2	POSIX Threads . . . . .	67
5.1.3	MPI . . . . .	68
5.1.4	CUDA . . . . .	69
5.1.5	OpenCL . . . . .	70
5.1.6	Cilk . . . . .	71
5.1.7	Taxonomy . . . . .	72
5.2	Tools . . . . .	72
5.2.1	ParallWare . . . . .	73
5.2.2	PaxES . . . . .	73
5.2.3	PLuTo . . . . .	75
5.2.4	Par4All . . . . .	76
5.2.5	AESOP . . . . .	77
5.2.6	MAPS . . . . .	79
5.2.7	Paralax . . . . .	80
5.2.8	Tournavitis . . . . .	81
5.2.9	Thies . . . . .	82
5.3	Taxonomy . . . . .	83
<b>6</b>	<b>Conclusions and future work</b>	<b>87</b>
6.1	Conclusions . . . . .	87
6.2	Future work . . . . .	89
	<b>Bibliography</b>	<b>91</b>



---

A  $\Pi$  parallel codes

105



# List of Figures

1.1	Processors performance . . . . .	2
1.2	OMAP5432 MPSoC . . . . .	4
2.1	Data dependence graph . . . . .	11
2.2	Parallelism levels . . . . .	14
2.3	Relationship of parallelism types . . . . .	17
2.4	Relevant parallelism forms to embedded systems . . . . .	19
2.5	Porting strategies for parallel applications . . . . .	20
3.1	Data Flow Graph for a quadratic equation solution . . . . .	28
3.2	Data Dependence Graph . . . . .	30
3.3	Control Flow Graph for a acyclic and cyclic code . . . . .	31
3.4	Control Flow Graph . . . . .	32
3.5	Control Flow Graph and Data Dependence Graph for example code . . . . .	33
3.6	Dependence Flow Graph . . . . .	33
3.7	Control Data Flow Graph . . . . .	35
3.8	Program Dependence Graph . . . . .	37
3.9	Augmented Program Dependence Graph . . . . .	39
3.10	Hierarchical Task Graph . . . . .	41
3.11	Augmented Hierarchical Task Graph . . . . .	42
3.12	System Dependence Graph . . . . .	44
4.1	Call tree for watermark detection algorithm . . . . .	51
4.2	Gene configuration in genetic algorithm . . . . .	58
5.1	Speedup on TI DSP multi-core platform using OpenMP . . . . .	66
5.2	PaxES parallelization tool . . . . .	75
5.3	PLuTo source-to-source transformation system . . . . .	76
5.4	Par4All flow . . . . .	77
5.5	AESOP Parallelizer . . . . .	78
5.6	MAPS tool flow . . . . .	79
5.7	Paralax tool flow . . . . .	80
5.8	Tournavitis parallelization work-flow . . . . .	81



# List of Tables

3.1	Auxiliar nodes. . . . .	45
5.1	Comparison chart for parallel programming models. . . . .	73
5.2	Taxonomy of tools for parallelism extraction. . . . .	84
5.3	Taxonomy of tools for parallelism extraction. <i>Continuation</i> . . . . .	85



# List of Abbreviations

AHTG	Augmented Hierarchical Task Graph
BCE	Base Core Equivalent
BDF	Boolean Dataflow
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CMP	Chip Multi-processor
CSDF	Cyclo-static Dataflow
DepFG	Dependence Flow Graph
DFG	Data Flow Graph
DLP	Data Level Parallelism
DSP	Digital Signal Processor
DSWP	Decoupled Software Pipelining
GA	Genetic Algorithm
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HTG	Hierarchical Task Graph
IC	Integrated Circuit
ILP	Instruction Level Parallelism
IR	Intermediate Representation
ISA	Instruction Set Architecture
LLP	Loop Level Parallelism
MoC	Model of Computation
MPSoC	Multiprocessor System-on-Chip
PDG	Program Dependence Graph
PE	Processing Element
PLP	Pipeline Level Parallelism
POSIX	Portable Operating Systems Interface
SDF	Synchronous Dataflow
SDG	System Dependence Graph
SMP	Symmetric Multiprocessor
SoC	System-on-Chip
TI	Texas Instruments
TkLP	Task Level Parallelism
TLP	Thread Level Parallelism





# Chapter 1

## Introduction

Nowadays computer-based systems are ubiquitous. Every person interacts with many of these systems everyday: from conventional personal computers to tablets and smart-phones, and from computers in cars, planes and trains to home appliances powered by microcontrollers. Embedded and cyber-physical systems have reached unthinkable areas and its computational power is far beyond the desktop computers of a couple decades ago.

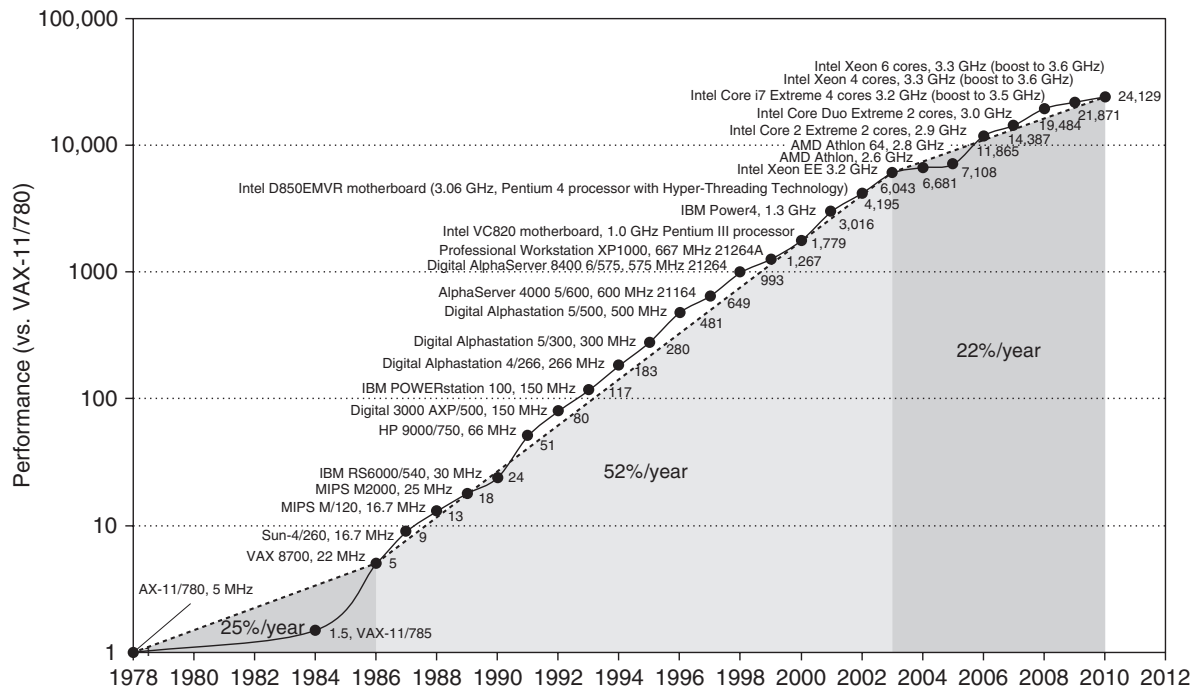
The end of the Dennard's scaling theory [36] in the sub-micron era, and the continuous increment in transistor count per integrated circuit (IC) due the Moore's law [89], forced a new paradigm in the design of microprocessor to harness more available hardware and increase performance. Around 2005 the industry shifted from single-threaded processors to multi-core processors [41]. As seen in Figure 1.1, the increase in performance in the last decade is thanks to the entrance on the scene of processor of multi-core systems, such the Intel Core 2 family.

A *multi-core* processor is a single computing system that presents two or more independent processors units (*cores* or, in a general perspective, *Processing Elements*, PE,) in an IC, with the goal to enhance performance, reduce power consumption, and be capable to efficiently process simultaneous tasks [106]. According to [130], a multi-core architecture implies three fundamental aspects:

- There are multiple computational cores.
- There is a way by which the cores can communicate.
- The processor cores communicate with the outside world.

With a multi-core system, a software program can be splitted and distributed to different PEs in order to be executed simultaneously and, in consequence, increase the task performance. Also, multiple independent software programs can be executed in different PEs.

The multi-core concept may appear to be trivial. Nonetheless, it suffers of scalability issues and there are numerous trade-offs to consider in its design. For instance, one



**Figure 1.1:** Processors performance from 1970 (Taken from [61]).

crucial consideration is whether the processor should be homogeneous or expose some heterogeneity; most current general-purpose multi-core processors are homogeneous both in Instruction Set Architecture (ISA) and performance, meaning that the cores can execute the same binaries and that it does not really matter, from functional point of view, on which core a program runs. However, recent multi-core systems allow, through system software, to control the clock frequency and supply voltage for each core individually in order to either save power or to temporarily boost single-thread performance [130]. In the other hand, heterogeneous multi-core architecture, using at least two different kinds of cores that may differ from performance and functionality up to different ISA, have been used in areas like gaming devices and high-performance computing.

Increasing the amount of cores or PEs in a multi-core system, reaching the dozens and even hundreds, is considered as *many-core* processors [58]. Currently exists some of commercial many-core systems. For example: the Intel Laberre [114], a many-core visual computer architecture made of multiple in-order x86 CPUs; the Intel Xeon Phi, that integrates up to 61 x86 64-bit cores [69]; and the Adapteva Ephiphany-IV, that presents 64 high performance RISC cores capable all to run for just 2 W of maximum chip power consumption [4]. Several challenges come when considering plenty of PEs. For instance, the capacity of applications to be mapped and to exploit the computational power of this platforms, and the current fact that for these system it will be impossible to switch on all PEs at the same time due the power and thermal constraints. This phenomenon is called the dark silicon problem and it will have a significant impact on how future processors will be designed [130]; recent works even predict that for 8 nm technology node, more than 50% of a multi- and many-core chip will remain dark [41].

It is important to set a difference between multi-core and *multi-thread* processors. The latter is considered as the ability of a program or an operating system process to manage its use by more than one user at the same time, which could be done by a single-core processor. Multi-threading capabilities are desired in multi-core systems because it aims to increase the utilization of each core employing Thread Level and Instruction Level Parallelism<sup>1</sup> [68]. Also, processor engines that can run numerous simple threads concurrently, traditionally used for graphics and media applications, are now evolving to allow general-purpose usage. Some authors refer to this type of systems as Many-Thread (MT) machines. Examples of these are the current General-purpose computing on Graphics Processing Units (GPGPU) of Nvidia and AMD [58].

Embedded systems scene is not exempt of the multi-core advent. *System-on-Chip* (SoC), broadly used in embedded systems, are defined as IC that contains not just a General Purpose Processors (GPP), mainly RISC-architecture based, but also includes, in the same silicon substrate, digital, analog, mixed-signal and even radio-frequency functions. A SoC commonly integrates communication and data transfer interfaces such as I2C, SPI, I2S, USB, Ethernet, WiFi, display systems controller, like VGA or HDMI, memory management, to name some. Which components are integrated in a SoC depends of the application for which is designed [70].

A *Multiprocessor System-on-Chip* (MPSoC) is a SoC with multiple PEs and special hardware blocks used to accelerate certain functions [142]. Most MPSoCs are heterogeneous, because of the diversity of PEs, which makes them harder to program than traditional multi-core systems. The combination of high reliability, real-time performance, small memory footprint, and low-energy, represents a considerable challenge in MPSoC software design [70].

The Figure 1.2 presents a good example of a state-of-the-art MPSoC. It contains a dual core of ARM Cortex-A15 Microprocessor Subsystem, capable to run up to 2 GHz, an ARM Dual Cortex-M4 image processing unit, HD hardware accelerator subsystem, a dual-core PowerVR 3D GPU, 2D-Graphics accelerator, imaging subsystem consisting of image signal processor and still image co-processor, also plenty of communication interfaces, memory controllers and diverse input/outputs ports. This example shows the diversity of PEs that can be found in a modern MPSoC.

Technology trends in MPSoCs design indicate an increase in the amount of PEs and the specialization of them. A multi-core system with specialized cores have demonstrated not only an increase in performance, but also a better power consumption compromise. In chips like the Apple A5, half of the chip area is dedicated to accelerators that are active only for some time and for specific tasks [103]. For instance, consider a RISC-based big core that operates at 620 MHz, consumes 365 mW, and presents a throughput of 900 MIPS. In the other hand a small core, also RISC-based, that operates at 225 MHz, consumes 32 mW, and presents a throughput of 330 MIPS. If three small core work together, this set will present a slightly increase performance, 990 MIPS, but a reduced

---

<sup>1</sup>Chapter 2 presents these types of parallelism.

## TI OMAP5432 SoC

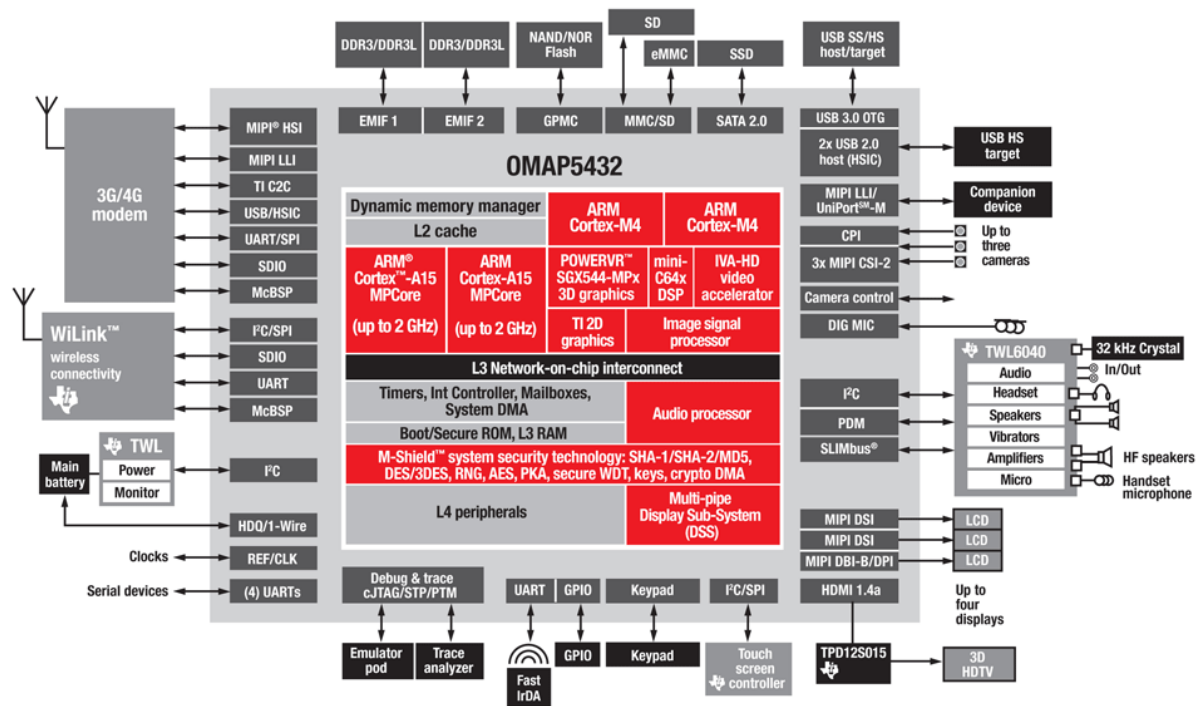


Figure 1.2: OMAP5432 MPSoC Multimedia Device (Taken from [www.ti.com](http://www.ti.com)).

required power budget, 96 mW, which means 4 times energy improvement regarding the big core.

Embedded and cyber-physical systems present different constraints than PC-processors, e.g.:

- **Energy:** most of embedded systems are battery-powered, so this resource must be well administered. For a handheld device results impractical to carry huge battery packs or keep the device plugged to energy sources.
- **Heat dissipation:** most of the embedded system use passive cooling to dissipate heat due the form factor of the final devices.
- **Response time:** real-time applications may have hard requirements. Certain mission-critical systems can not admit failures, and a correct execution out of a time constraint can lead to a tragic outcome.

The complexity of the applications nowadays require higher performance capabilities within tight power envelops. This is a reason why many heterogeneous MPSoC use Graphical Processor Units (GPU) or Digital Signal Processors (DSP) to increase performance of specific software applications such as multimedia, gaming, audio and image processing, machine vision. The following are just two examples of current high performance embedded platforms:

- big.LITTLE is an heterogeneous multi-core architecture from ARM that presents a compromise of high-performance cores combined with power-efficient cores, with the goal to provide peak-performance processing capacity, a higher sustained performance, and increase the parallel processing performance, at a lower average power consumption [12]. Also it includes GPUs to assist in graphics related tasks. This platform allows the operative system to assign task and threads to the appropriate core, either a low-power or a high-power one, based on dynamic run-time behavior.
- Texas Instruments (TI) have released in the past years the Keystone Multi-core Processors family following in part the multi-core DSP trends described in [73]. The Keystone I presents configurations from 1, 2, 4 or 8 cores of the powerful fixed and floating-point C66x DSPs. The Keystone II integrates to the C66x cores a cluster of up to 4 ARM Cortex A-15 cores for general purpose computing, providing a unified platform of RISC and DSP cores. These platforms are suitable for high-performance signal processing applications such as mission critical systems, medical imaging, communications and networking, to name a few.

## 1.1 The sequential code problem

Take advantage of the computational power of a MPSoC, or a general multi-core system, does not come for free. Millions of lines of sequential code have been developed over the years considering its execution on a single core. A sequential program is one whose instructions are executed in the order they were written. Possible performance gains of legacy sequential code executed in multi-core systems is limited by the amount of parallelism that can be extracted and exploit from that code. Considering the improvement in core performance and energy efficiency, one could consider to keep mapping sequential applications to a single PE, but doing so no profit is achieved from the modern MPSoC parallel computing power.

The majority of mainstream programming languages are designed to express sequential programs [119]. Most of the code running in embedded systems has been written in languages like C<sup>2</sup> and C++. These programming languages are inherently incapable to express parallelism. If a parallel program is intended to be written, add-on libraries and language extensions are used, which encapsulate the expression of concurrency in a form that, to the compiler or language, it remains sequential [119].

Availability of parallel hardware platforms has not change automatically the form in which software is develop. We still continue thinking and developing code in a sequential fashion, even this is the way programming is still taught in universities and technical training institutions, which represent a challenge to exploit hardware parallelism. Parallel

---

<sup>2</sup>This language has been widely used in embedded systems due it brings an abstraction layer with easy access to hardware components. It has have been adopted for many chips manufacturing providing compilers for a extensive assortment of architectures, even DSPs and soft-cores built on FPGAs.

programming paradigms should be taught in order to take advantage of the hardware technological advances.

A proposal to overcome the codes shortcomings, is to create specifically languages for concurrent programming in order to exploit parallel processing capabilities in a general sense and to facilitate automatic compilation and analysis [119]. Nevertheless, the sequential problem, as proposed by [23], should be addressed as a parallelism extraction problem from sequential programs, if legacy sequential code is trying to be reused.

## 1.2 Parallelism extraction

The improvement in performance that can be achieved using multi-core processors depends very much on the software algorithms used and their implementation. New approaches propose to write the code using explicit parallelism instructions and indications, by using libraries and language extensions to current programming languages.<sup>3</sup>

Manual parallelization is a time-consuming and error-prone task. It is desired a tool capable to take an unaltered and unannotated (or with small annotations) sequential program, and produce a parallel version of the original program, able to be compile and execute in a multi-core platform [88]. However, this is an ambitious goal. Most of existing applications require some effort from the programmer in re-factoring the code to help the tool to discover and exploit parallelism, which corresponds to a semiautomatic parallelization. Those tools that can handle unmodified code are consider as automatic parallelization tools.

Extracting parallelism from a sequential code is not a new task. Since the 80s and 90s, the problem has been addressed with different success rates. According to [121], the topic of auto-parallelizing compilers has been in the table of researchers from a long time ago, mainly attacked for high performance computing (HPC) and desktop systems. Many of the techniques used and developed over the time to attack parallelization have moved into commercial compilers and are supporting both parallel and serial optimization.

Automatic and semiautomatic parallelization success have achieved fairly success in FORTRAN language, and partially in languages such as C/C++, due the language inherently inability to represent parallelism. Application program interfaces such as OpenMP has become popular. OpenMP is a compiler extension for C, C++ and FORTRAN that allows to add parallelism into existing source code without significantly having to rewrite it. The programmer's knowledge of the source code is required to determine which region of the program can run in parallel and then annotate the code. The OpenMP success had pressured multi-core vendors and designer to support it by their platforms and tools.

Techniques such as OpenMP provide more information to the compiler in the form of

---

<sup>3</sup>In Section 5.1, parallel libraries and language extensions, frequently used in industry and academia, are presented.

code annotations, but they still suffer from the fact that they are inherently bound to the sequential languages that they are used with.

Some challenges faced by the automatic and semiautomatic parallelization problem are:

- The diversity and nature of the different programming languages makes it difficult to have a universal tool for such task. Each programming language is able to intrinsically represent parallelism at different levels, with certain degree of parallelism (DoP).
- Using pointers, like in C or C++, create a big problem for compilers because they obscure what data is actually being accessed, making the problem of understanding data access hard task [121].
- Compiler has little information about the program. Runtime information obtained from the program through profiling is useful.
- The cores architecture is another issue. Different ISA based processors would require a compiler capable to map parallel code to different architectures, and probably different backends. For instance, a heterogeneous multi-core system with different types of PEs, such as General Purpose Processors (GPP) and DSPs, in the same chip. Communication and synchronization among cores is another challenge that can have a big impact in performance.

Recently works have addressed new and important approaches, even considering constrained MPSoCs for embedded systems. As a glimpse, in [125] static information at compiling time is complemented considering profiling and runtime information to the compiler. In [33] parallelizing techniques are developed considering not only improvement in execution time as an objective but considering energy, an important constraint in nowadays embedded systems.

### 1.3 Scope of this work

It is clear that automatic and semiautomatic parallelizations is an open-end problem and there is not a general solution. Embedded MPSoC platforms present new challenges and opportunities. Even that most compiler technology have been thought for one-core systems, and of course it does not scale properly to multi-core ones, its fundamentals can be used to develop parallelizing compilers at a greater level of granularity. Many of the approaches treated over the time to represent and extract parallelism have been focused to supercomputers and desktop computers. These different approaches, used to tackle this challenge, are worthy to study and analyze in order to find applicability in recent multi-core embedded systems.

This work aims to present a survey related with the models, algorithms and tools used for parallelism extraction and its possible usage for multi-core embedded systems. This work aim to assess:

- Different representation models used to abstract parallelism.

- Algorithms used to detect and extract parallelism.
- Existing tools for parallelism extraction and conversion of sequential code into a parallel version and its usability in real MPSoC platforms such as the Texas Instruments Keystone I TMS320C6678 with 8 DSP cores.

## 1.4 Outline

The remainder of this thesis is structured as follows:

**Chapter 2:** Background concepts related automatic and semiautomatic parallelization are presented, such as dependencies, types of parallelism, granularity, and characteristics of multi-core systems, in embedded and high-performance scopes.

**Chapter 3:** Models employed to represent code and its relationship are presented in this chapter. Its characteristics are presented and discussed.

**Chapter 4:** Many algorithms have been used to detect and extract parallelism from sequential code. Some of this algorithms are presented in this chapter. This chapter presents characteristics of static, dynamic, and hybrid code analysis.

**Chapter 5:** Commercial and academic tools for parallelization of sequential code have been developed in the past years. As well, there are parallelism extraction approaches developed and proposed. Some tools and approaches are considered and analyzed in this chapter. Code libraries and language extensions have been widely used for tools to transform the sequential codes into parallel, some of which are presented in this chapter.

**Chapter 6:** Conclusions from this work and possible future lines of analysis and discussion are presented.



# Chapter 2

## Background

To better comprehend how different models represent parallelism, and how algorithms and tools work to find and extract parallelism from sequential code, it is necessary to introduce some background concepts regarding parallelism, e.g., Amdahl's law, dependencies, types of parallelism and its relationships, characteristics of parallelizing compilers, and characteristics of embedded and high-performance systems.

### 2.1 Amdahl's law

Gene Amdahl's work brought an important conclusion to the computational landscape, the Amdahl's law [9]. This law establishes that *the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program*. Considering this premise, any application can present a portion of code that executes strictly in a serial manner, and a complementary portion which can be executed in parallel. Hence, for a given sequential program, its execution time can be improved if portions of code are run in parallel. Determine these parallel sections of code is the challenge.

Mathematically, Amdahl's law states that if  $P$  is the portion of a program in parallel and  $(1 - P)$  the serial portion, the maximum speedup that can be achieved by  $N$  processors is given by

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \quad (2.1)$$

It is important to emphasize that Amdahl's considerations were conceived in a time when single-core computers were available. Execute programs in parallel was possible by interconnecting several computers and distributing parallel portions of the application among them.

Today multi-core capabilities in a single chip are higher than ever though, but the vast

amount of sequential code developed along the years, limits the exploitation of this platform using such programs as currently. It is needed to extract the inherent parallelism in applications in order to obtain the parallel fraction of (2.1).

Some studies have questioned the validity of this law in the multi-core era, specially considering embedded systems. Nevertheless, authors have presented important consequences of the Amdahl's law in the multi-core era [62].

Considering a baseline core equivalent, BCE, and assuming a system of  $n$  BCEs, the authors of [62] developed 3 models for multi-core chips: *symmetric*, single BCEs or groups of BCEs, e.g, tiles of 4 BCEs; *asymmetric* or heterogeneous, where cores present different computational power capabilities; and *dynamic*, where some cores in a chip are combined dynamically to boost the performance in the sequential component. For each model speedup formulations are defined based in (2.1). Simulations, considering high fractions of parallel code, showed an increase in the speedup as predicted by Amdahl's law.

Even that the space explored by the authors can not be build by current design techniques, this work set forth insightful ideas for multi-core hardware and software developers to harness the continuous increments in the number of transistors per IC, such as dealing with the overhead to develop parallel code, and its associated cost, and the challenge to schedule software to asymmetric or dynamic multi-core chips. The work even show that asymmetric multi-cores can offer potential speedups that are much greater then symmetric multi-core chip.

## 2.2 Dependencies in code

Automatic and semiautomatic parallelism detection requires data dependence testing [96]. Data dependence relations are used to determine when two instructions, operations, statements, or iterations of a loop, can be executed in parallel. How one instruction depends on another is important to determining how much parallelism exists in a program and how that parallelism can be exploited. The dependencies tell where data is produced and consumed.

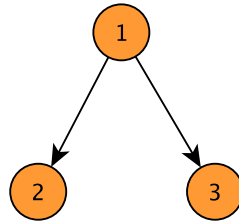
For instance, if two instructions are data dependent, they must execute in order and cannot be executed simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between two instructions. The key is to determine whether an instruction is dependent on another one [61], and a precise analysis of these dependencies is critical do detect and extract parallelism while not violating the functional correctness of a program. In this section, data<sup>1</sup> and control dependencies in basic code structures are presented.

---

<sup>1</sup>These dependencies are classified as load-store, due they are expressed in terms of load-store order [74].

```
1 X = Y + Z;  
2 W = X + 3;  
3 V = X * 2;
```

**Listing 2.1:** True dependencies.



**Figure 2.1:** Data dependence graph.

### 2.2.1 Data dependencies

Consider the code in Listing 2.1. The instruction 2 can not be executed at the same time than instruction 1, due the first modify the value of  $X$  needed by the second instruction. If both instructions are execute at the same time,  $W$  would have an incorrect value. The same situation happens with instruction 3, it can not be executed at the same time than 1. Thus instruction 1 must be executed before 2 and 3, and thus ensure that instruction 2 and 3 receives the correct value from the first instruction [96]. This is called *true dependence*, and it is depicted in Figure 2.1.

In the context of hardware design, dependencies are commonly called *hazards*. Data hazard is consider whenever there is a data dependence between instructions and, just as previously presented, the overlap of instructions during execution would change the order of access to the operand involved in the dependence. *Read After Write* (RAW) is a common data hazard that corresponds to *true dependences* [74]. From the Figure 2.1 can be notice that instruction 2 and 3 are not connected and they may be executed in parallel if PEs are available.

A parallelizing compiler should implement techniques to exploit parallelism by preserving the program order only where it affects the outcome of the program. Detecting and avoiding hazards ensures that necessary program order is preserved [61].

Consider the code in Listing 2.2. The instruction 2 assigns a new value to  $Y$ . Since instruction 1 use an old value of  $Y$ , it must be executed before 2. This is called *antidependence* and it is data harzard classified as *Write After Read* (WAR). At a microarchitectural level, this hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline.

A third type of dependence is exemplified in Listing 2.3. In this case, the instruction 3 assigns a new value to  $X$ , after instruction 1 did it. In a case where instruction 1 is

```

1 X = Y + Z;
2 Y = W / 2;

```

**Listing 2.2:** Antidependence.

```

1 X = Y + Z;
2 Z = X + 3;
3 X = V + W;

```

**Listing 2.3:** Output dependence.

executed after the 3, the value of  $X$  would be incorrect. For this case, instruction 1 must precede the execution of 3. This is known as *output dependence* and correspond to a *Write After Write* (WAW) data hazard, where writes end up being performed in the wrong order.

## 2.2.2 Control dependencies

The flow of control must also be taken into account when discovering data dependence relations. A control dependence determines the ordering of an instruction with respect to a branch instruction, so that instruction is executed in correct program order and only when it should be executed [61]. Control dependencies must be preserved to keep the program order. One simple example is the dependence of the `then` part of an `if` statement on a branch. For instance, consider the code segment in Listing 2.4. The output dependence between instruction 1 and 3, and the true dependence between 1 and 5 are present. However, between instruction 3 and 5 there is not a true dependence. Due instruction 3 and 5 are in different branches of the same `if` statement, the value of  $A$  assigned to  $D$  will never proceed from instruction 3.

A slightly modified code in show in Listing 2.5. In this code true dependences appears between instruction 1 and 5, and instructions 3 and 5. Those dependences could be computed by a compiler, even when the value assigned to  $D$  would come only from one of the instructions (1 or 3) depending of the value of  $X$ .

Due the actual execution flow of a program is not known until run time, data dependence relation does not always imply data communication or memory conflicts [96]. This make a static analysis, at compile time, a needed but incomplete tool to accurate determine and extract parallelism.

According to [61], there are two constraints imposed by control dependences:

- An instruction that is control dependent on a branch cannot be moved before the branch, so that its execution is no longer controlled by the branch. One instruction from inside the `then` section in an `if` statement, cannot be moved before the `if`.
- An instruction that is not control dependent on a branch cannot be moved after the

```
1 A = B + C;  
2 if (X >= 0) then  
3   A = 0;  
4 else  
5   D = A;  
6 end if
```

**Listing 2.4:** Control dependence 1

```
1 A = B + C;  
2 if (X >= 0) then  
3   A = A + 2;  
4 end if  
5 D = A * 3;
```

**Listing 2.5:** Control dependence 2.

branch, so that its execution is controlled by the branch. One instruction before a `if` statement cannot be move it into the `then` segment.

## 2.3 Parallelism: types and relationships

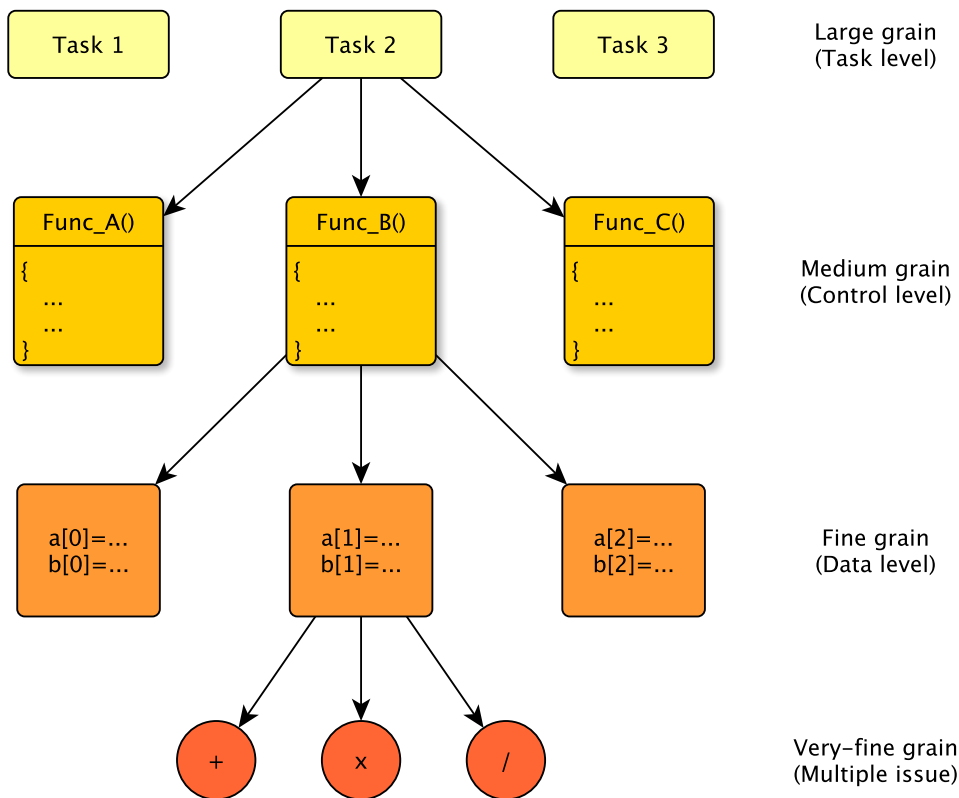
### 2.3.1 Granularity and parallel performance

In modern computers systems, parallelism appears at different abstraction levels, both in hardware and software. In [90], parallelism is consider at signal, circuit, component, and system levels. Signals and circuits levels are consider as hardware parallelism, and the execution in those levels occurs concurrently due the nature of hardware. Component and system levels are consider as software parallelism and they are mostly expressed implicitly or explicitly using various software techniques.

The levels of parallelism can be based on pieces of code, also called *grain size*, that can be potential candidates for parallelism. Figure 2.2 shows a classification of levels of parallelism according to granularity.

The granularity can be defined as the size of work in a single task of a multi-thread application [37], the amount of work in a parallel task. It is important to mention that different levels of granularity can affect the performance in different aspects. As presented in [37], a fine-grained parallelism means individual tasks are relatively small in terms of code size and execution time, but it introduces communication overhead; in coarse-grained parallelism data communicate infrequently but may introduce load imbalance among parallel tasks and cause synchronization issues [26].

As presented in Figure 2.2, different levels of parallelism can be found in an application,



**Figure 2.2:** Parallelism levels (Based on Figure from [90]).

from execution of specific operations, such as adds or multiplications, up to tasks or threads of a task execution at the same time in different PEs. Parallelism based on code granularity have a common goal to boost processor efficiency facing the compromise of latency of lengthy operations. The very-fine grain size, found in code as instructions, is parallelized by the processor. The fine grain size, loops or instructions blocks, is primarily parallelized by the compiler. Parallelize medium and large grain size code segments, found in program functions and separate programs, has been a task for the programmer. Automatic or semiautomatic parallelizing tools look for the exploitation of coarser-grained size sections.

From the granularity concepts, and how it is related with different levels of code, it is important to conceptualize the different levels of parallelism that is possible to extract and exploit from an application, and also how these types are related one to each other. In [61] the basic, and classic, types are covered from the micro-architectural perspective and considering software impact. [139] and [115] emphasize in their relationships.

### 2.3.2 Data Level Parallelism

*Data Level Parallelism* (DLP) is a class of parallelism that can be exploited by broadcasting, at each step in the execution, one operation to a set of  $n$  PEs, this is several

cores executing the same instruction (or instructions) simultaneously, but using different data [139]. As a simple example, consider a thresholding operation applied to a gray-scale image. If  $n$  PEs are available, each one can perform the threshold operation on one single pixel at a time. This type of parallelism is exploited using Single-Instruction Multiple-Data (SIMD) processors, according to Flynn taxonomy [45]. DLP tries to duplicate the statements of a loop's body to execute the same task on several processing units.

There are three processors variants that take advantage of the SIMD concept. The *vector* variation implements instructions that operates on one dimension arrays of data, a pipelined execution of many data operations. The *SIMD* variation perform simultaneous parallel data operations and nowadays is found in most Instruction Set Architectures (ISA) today that support primarily multimedia applications [61], such as the Multimedia Extensions (MMX) and the Streaming SIMD Extensions (SSE) in the x86 CPU architecture. A GPU, for instance, offers high potential performance used in current multi-core computer to accelerate task as video decoding.

### 2.3.3 Instruction Level Parallelism

*Instruction Level Parallelism* (ILP) is a type of parallelism in which one thread of control is capable to schedule multiple, and usually different, operations for the execution on different PE [139]. The processor must present multiple functional units capable to operate in parallel, in one same clock cycle. For example overlap CPU and I/O activities, memory interleaving techniques [90].

This parallelism has been exploit by superscalar and Very Long Instruction Word (VLIW) processors, where those might have, e.g., two integer ALU functional units, a floating multiply unit and floating point divide unit. In a processor like this one, the code might be scheduled in order to have an integer add, a floating point multiply and a floating point division instructions executed at the same time if the instructions issued are independent and have no true dependencies between them.

According to [61], there are two major approaches to exploiting ILP. One relies on hardware to help in the process of discover and exploit the parallelism dynamically; the other one relies on compilers to find parallelism statically at compilation time.

A basic block of code, one with no branches (just at the beginning and at the end), present a small amount of parallelism. To increase the parallelism at this level, ILP should be done across multiple basic blocks. The most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *Loop-Level Parallelism* (LLP).

There are several techniques that allow the conversion of LLP into ILP:

- Compilers techniques such as loop unrolling and basic pipeline scheduling.
- Reduce branch costs with advanced branch prediction.
- Overcome data hazards with dynamic scheduling, e.g. using Tomasulo's approach.

- Hardware based speculation.
- Multiple issues and static scheduling.
- Increase instruction fetch bandwidth.

### 2.3.4 Thread Level Parallelism

*Thread Level Parallelism* (TLP) is a type of parallelism exploited in architectures where multiple threads of control can be executed, each of which involves multiple operations that may be executed on multiple functional units. Chip multi-processors (CMP) and MPSoC can exploit this type of parallelism [139].

Multi-core processors, typically controlled by a single operating system and that share memory through a shared address space, are used to exploit this type of parallelism. Such systems exploit TLP through two different software models, by the execution of [61]:

- A tightly coupled set of threads collaborating on a single task, which is typically called parallel processing.
- Multiple and relatively independent processes whose origin can be from one or more users.

Many multi-core systems support multiple threads executing in an interleaved manner, just like in single multiple-issue processors. According to [45], processors capable to exploit TLP are classified as Multiple-Instruction Multiple-Data (MIMD) processors. For example, if  $n$  PEs are available, it is capable to handle up to  $n$  simultaneous threads mapped to its cores. The independent threads within a single process are typically identified by the programmer or created by the operating system. Also, it is important to denote that a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop [61].

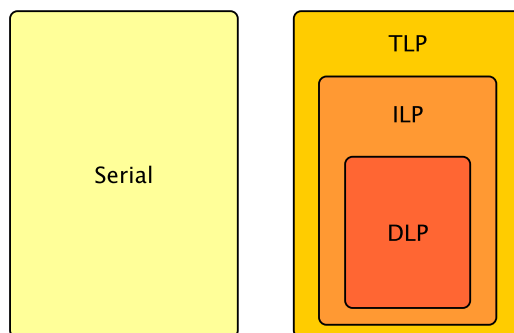
This type of parallelism presents challenges to be exploited: one is the limited parallelism available in the source code of applications, which limits the potential speed-up using the multi-core systems; the other is related with the potential high communication cost between threads running on different PEs, and the effect of long communication delays is clear.

At this point, three general parallelism levels have been presented. They exhibit a hierarchy. As a summary [115]:

- Data Level Parallelism is the lowest level. It is class of parallelism where one operation is performed by a group of similar processing elements at the same time.
- Instruction Level Parallelism is the next level. It refers to several different instructions being executed by different processing elements at the same time.
- Thread Level Parallelism is at the top level in the hierarchy. It refers to multiple threads of control being executed in parallel, usually in multiple PEs.

There is an interesting and important relationships between these parallelism levels, ac-





**Figure 2.3:** Relationship of parallelism types (Based on Figure from [139]).

According to [139] and [115]. As defined by Amdahl's law, there is a portion of the code that can only be executed in serial and a portion that can be executed in parallel, as shown in the Figure 2.3. This model breaks the parallel portion into fractions of parallelism that can be only exploited by TLP; a fraction that can exploit TLP and ILP, which is the ILP fraction; and a fraction that can be exploited by any of the three types of parallelism, this is the DLP fraction. This way to describe the parallelism present in a given application, is considered by the authors as a guide for the parallelism extraction process and for targeting to the right PEs.

For instance, DLP is a subset of ILP. If a ILP-capable processor can exploit data parallelism if it has multiple versions of each type of functional units in a PE. It schedules multiple instructions of the same type, on multiple units of the same type. On the other hand, ILP cannot be exploited by DLP-capable processors, due ILP is exploited by concurrently executing different instructions on different functional units. In DLP architectures, a common instruction is broadcast to all PE, so it is not able to execute independent instructions in parallel because it can only broadcast one type of instruction simultaneously.

ILP is a subset of TLP. An ILP program can be executed on a PE design to exploit TLP, creating multiple instances of the ILP's thread control. Conversely it is not possible due TLP present in a program that have multiple independent threads of control cannot be exploited by an ILP PE since it can only execute one thread control at a time.

Embedded multi-core systems, due its variety of PEs, present a better performance boost when coarser levels of parallelism are extracted from sequential code and exploit in its architectures. There are two types of TLP that can be exploit particularly for multi-core embedded systems, Task Level Parallelism and Pipeline Level Parallelism [33].

#### 2.3.4.1 Task Level Parallelism

*Task Level Parallelism* (TkLP) is a coarser-grained type of TLP. This type of parallelism focus on the distribution of execution process, or threads, to the PEs of a multi-core or

multi-processors system. As presented in [33], big and independent code blocks of an application can be processed by concurrently executed tasks. These blocks can be code functions or even single statements, depending on the desired level of granularity. This parallelism divides statements of an application into coarse-grained, disjunctive tasks to operate, in preference, on independent data.

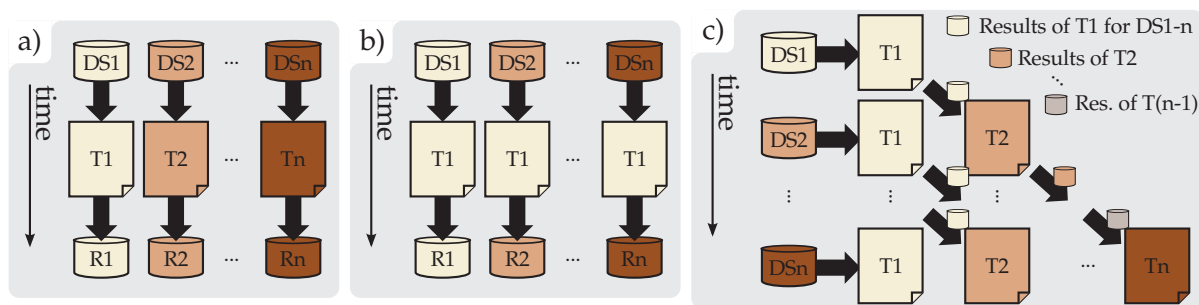
In embedded systems, this type of parallelism is efficient because in many cases only few data has to be communicated between the different executed tasks. [33] propose techniques for extracting this kind of parallelism applied to homogeneous and heterogeneous embedded MPSoCs for one and also for multiple objectives. Task level parallelism often benefits from concurrently executed function calls or the parallel execution of several independent loops. In [50], one of the first works on automatic detection of task-level parallelism is presented. The authors approach the problem of detecting, expressing, and optimizing task-level parallelism, considering a task as program statements of arbitrary granularity.

#### 2.3.4.2 Pipeline Level Parallelism

*Pipeline Level Parallelism* (PLP) is another type of TLP. It can be used to extract efficient parallelism from many embedded applications, especially those which are written with a streaming-oriented structure. Pipeline parallelism splits a loop code structure into different tasks, which are executing disjointed parts. In other words, with Pipeline Parallelism each loop iteration is splitted into stages and threads operate on different stages from different iterations concurrently. Each stage is assigned one or more worker threads and an in-queue which stores the work to be processed by that stage.

PLP is powerful because it can expose parallelism in ordered loops where iterations are non-independent, loop-carried are present, and cannot run concurrently. By splitting each loop iteration into segments, intra-iteration parallelism can be exposed. This parallelism is a powerful method to extract parallelism from loops which are difficult to parallelize otherwise. The advantage of this parallelization is that each task can start the next iteration of the loop, executing its assigned statements as soon as it has communicated its result to the tasks waiting for its output. In this way, a pipeline of calculations is created [29].

The advent of general-purpose multi-core system as well as the proliferation of multimedia consumer electronics has underlined the importance to focus on higher-level parallelism [125]. Multimedia applications typically comprise pipelined computations and operate on a stream of data of different granularity. The reason is that many embedded applications, especially those like networking, voice and image processing, and multimedia tasks like video decoding, are structured in a pipelined manner. All these applications have in common that most of their parallelism is hidden in loops containing different pipelining-based jobs. For instance, in image processing several filters are applied to a block of a source image and each filter depends on the previous one [127].



**Figure 2.4:** Relevant parallelism forms to embedded systems: a) Task, b) Data, and c) Pipeline Level Parallelism (Figure taken from [23]).

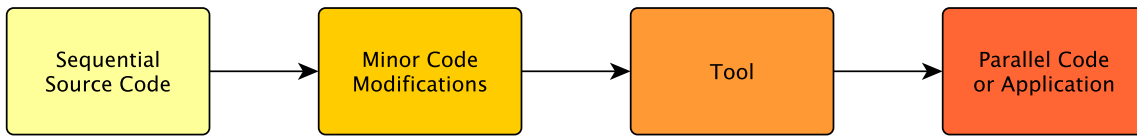
PLP may be also conceptualized as a chains of producers and consumers directly connected in the stream graph [51]. Compared to DLP, this type of parallelism approach offers reduced latency, reduced buffering and good locality. It does not introduce any odd communication, and it provides the ability to execute any pair of statements in parallel. Nevertheless, this type of pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution. In addition, effective load balancing is critical, as the throughput of the stream graph is equal to the minimum throughput across all of the PEs.

From the different kinds or level of parallelism already exposed, some can be present, but hidden, in an application. Traditional compiler has focused in detecting and exploiting ILP. Nevertheless, the required parallelism for MPSoC is in a coarser-grained level. So, the most important types of parallelism should be consider by tools looking for an automatic or semiautomatic parallelism extraction should focus in Data, Task, and Pipeline Level Parallelism. Figure 2.4, from [23], depicts these parallelism types for different tasks ( $T_1, T_2, \dots, T_n$ ), their corresponding input data set ( $DS$ ) and the output results ( $R$ ).

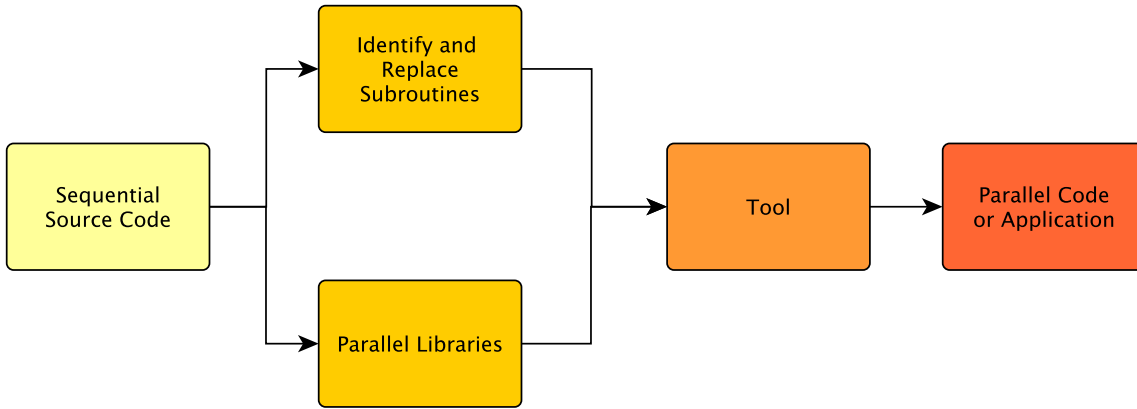
## 2.4 Parallelizing compilers

Automatic and semiautomatic parallelization of sequential source code has been an ambitious research focus in order to exploit and take advantage of the capabilities of parallel architectures and commercial multi-core processors, such as modern MPSoC. The ultimate goal for a parallelizing compiler, one automatic, is to take an unaltered and unannotated sequential program and perform a compilation process that produces a efficient parallel object code without any, or very little, additional work [88], relieving the programmers from the exhausting and error-prone task of manual parallelization [47]. This issue has been deeply covered for dense array-based numerical programs and shared memory multi-core processors, due the wide availability of those types of processors and the great amount of applications that using this data structure.

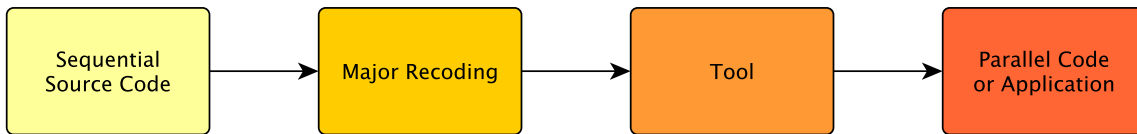
There are essentially two approaches for parallel programming according to [90]. One is based on *implicit* parallelism, where the parallelizing compiler is responsible for obtaining



(a) Automatic parallelization



(b) Parallel libraries



(c) Recoding

**Figure 2.5:** Porting strategies for parallel applications (Based on a Figure from [90]).

the parallelism and scheduling the tasks. This approach is followed by parallel languages and parallelizing compilers developed, and is important to mention that there is no control by the user in, e.g., how the scheduling is done. The other approach is based on *explicit* parallelism, in which the programmer is responsible for most of the parallelization effort, such as task decomposition, mapping task to processors, communication structure. Even when this approach represents a big load of work, the user is often the best judge of how parallelism can be exploited for a particular application.

A parallelizing tool could include both implicit and explicit parallelism approaches, due to the fact that programmer's knowledge about which parts of the application to parallelize is extremely useful. This information can be accompanied by dynamic information, application's run-time behavior, that can bring information, e.g, about which code's sections in a program are executed the most (see Section 4.1).

In [90], three approaches are presented on how obtain a parallel version of a sequential program. Figure 2.5 illustrates these approaches. This classification can apply to automatic and semiautomatic parallelizing compilers, with some minor differences. The

Figure 2.5(a) truly represent the case of an automatic tool. Minor modifications, or no modifications, are expected for an automatic process, however some existing automatic tools can not support certain code structures and require slightly modifications in order to automatically detect and extract parallelism<sup>2</sup>. Other important point is that automatic tools may use libraries and language extensions to generate parallel version of the input sequential code (see Chapter 5). The Figure 2.5(b) represents a flow that applies for a semiautomatic parallelizing compiler. In this case, the programmer annotate or instrument the code to help the tool discover parallelism, or to indicate important code sections that have parallelism potential; even mayor code modifications by the programmer are done to help the tool to better find and exploit the parallelism. Figure 2.5(c) present the manual process of converting sequential source code to a parallel version.

A parallelizing compiler may present the same stages of a typical compiler: a front-end, a middle-end, and a back-end; but for the purposes of extracting parallelism, [23] proposes that these compiler phases perform their work at a coarser level than regular compilers.

In a roughly way, the general procedure of parallelization should consider (based on [137]):

1. Identification of parallelizable code sections, which in a semiautomatic fashion can be determined by the programmer. The challenge arises when the program was written by a person different than the one trying to parallelize it. These code “hot spots” or parallelizable candidates could be functions or loops.
2. Dependencies analysis of the selected code sections to determine a model of the program. This step difficult because it involves lot of analysis. Generally for codes that use pointers are difficult to analyze. Many special techniques such as pointer alias analysis, functions side effects analysis are required to conclude whether a section of code is dependent on any other code.
3. Reduce the amount of iterations among different parts of the program that may prevent parallelization, usually done by code transformations. Code transformation are used to remove dependencies [88]. Code is transformed such that the functionality, and hence the output, is not changed but the dependency, if any, on other code section or other instruction is removed. Also, additional optimization of sequential threads so it can be execute efficiently.
4. Once dependencies are identified and reduced, the parallelism should be detected and extracted from the input source code.
5. The last step is to generate the parallel code, manually or automatically, or in some cases produce the parallel executable code. In case that the output is a parallel version, this must have the same functionality than the original sequential code, but with additional constructs or code structures, that once compiled, create multiple threads or processes.

---

<sup>2</sup>An example is the tool presented in Section 5.2.1

One key to parallelization is find a way to split the work of the code such that a write to a memory location by one thread is always separated from another thread's access to the memory location by a synchronization. So, uncertainty about which data is being accessed by which threads really hampers parallelization [121]. In fact, one of the main difference between a shared and distributed memory is how communicating thread, or processes, communicate. Barriers are used to synchronize parallel execution of tasks, and this barriers ensures that all the threads executing in parallel finish before any part of the program after the parallel construct is executed, for example a serial, not parallelizable, part of the program [88].

When a program executes in parallel, different parts of the program are spread across multiple PE and execute simultaneously. However, develop parallel applications exhibits challenges not presented in sequential programming, e.g., non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, fault-tolerance, heterogeneity, shared or distributed memory, dead locks and race conditions; this challenges should be considered in the process of developing a tool that identifies and extract parallelism [90]. The challenge is even bigger when considering state-of-the-art MPSoC systems where not only heterogeneous CPU cores are found (with different power/performance profiles but sharing the same ISA) but also PEs such as DSPs and GPUs.

Considering MPSoCs, a parallelizing compiler should not just take in account increase performance by exploiting all the available parallelism in a program. It should consider outstanding aspects regarding MPSoCs, such as energy consumption associated with the number of PEs active and running, and communication overhead cause by the different threads and tasks running and it impact in the overall performance. In this context not always exploit the highest amount of parallelism available is the best choice for a MPSoC.

The present work intend to present and discuss different aspects to be consider when developing a tool for automatic or semiautomatic parallelization tool from sequential code, such as representation models, algorithms for detection and extraction parallelism, and tools and approaches already developed, and how those can be used considering multi-core embedded systems as a target.

## 2.5 Embedded Systems and High-Performance Computing

Embedded System (ES) and High-Performance Computing (HPC) are two distinct areas in the computational world. General-purpose systems, such as desktop and servers computers, have followed an aggressive development to provide high performance for a range of applications [72]. Since the 1960s, HPC systems have been designed as front-line machines considering its processing capacity. At the beginning these systems were designed with only few processors, by the end of the last decade, massively parallel supercomputers with tens of thousands of the manufactured state-of-the-art processors became the norm.

HPC systems have been extensively used for a wide spectrum of computationally intensive tasks in various fields, e.g., numerical simulation, systems modeling and synthesis, quantum mechanics, weather forecasting, molecular modeling and machine learning [86].

In the other hand, the embedded systems are computational systems that have been designed, generally, with one or a few specific functions [141], either as custom standalone device dedicated to the execution of specific tasks, e.g. aeronautical guidance systems in aerospace industry, or as custom integrated devices with a dedicated function within a larger mechanical or electrical system, e.g. ABS car breaks, telecommunications systems (radar, satellite), bio-medical instrumentation [86]. Despite the broad field of embedded systems applications, there are some common characteristics:

- Specialized to an application domain, single application or task, which makes them less flexible but probably more efficiently to design.
- Underlines many and tight constraints, such as power/energy, heat dissipation, timing, form factor, price.
- Due the constraints, design embedded systems becomes more challenging than design a general-purpose computers.
- There is a high volume compare to regular CPUs. For instance, a luxury car has already more than 100 embedded systems.
- Embedded systems are a whole system in a single IC, with CPU, memory and I/Os.

Many modern embedded systems present high computing capabilities that overcome the desktop computers of 15 or 10 years ago. Current embedded system encompass HD, 3D video and multimedia systems, and smart-phone and tablet computers never thought before. The latter two have been classified as *Personal Mobile Device* (PMD), a term used for wireless devices with multimedia user interfaces [61].

Much of the research effort in parallelization detection and exploitation from sequential code have been done for HPC, a probe of this is the automatic and semiautomatic parallelization tools developed (see Chapter 5). Due the difference in the characteristics of both worlds, HPC parallelization tools are not able to work, one to one, to a wide diversity of embedded systems. Of course, some tools may applied due some embedded systems capabilities. For instance, Texas Instruments offers support for the OpenMP API in its KeyStone multi-core architecture [66], allowing the usage of OpenMP parallel versions of sequential C codes, parallelized with HPC tools.

Even the differences between these two worlds, nowadays both face similar design challenges and issues, whose solutions converge to similar approaches. For example, increasing computational demands and large degrees of parallelism take an important role, especially when considering in real-time applications. These factors led to distributed many-core platforms in embedded systems design, and to massively parallel systems in HPC domain [87]. The introduction of heterogeneous processors, specialized for different and specialized workloads, like GPUs, which often lead to the integration of embedded systems in complex HPC systems. The movement of graphics processing units into mainstream general-purpose platforms in now a reality in desktop IC CPUs [72]. Modern computing

systems contain a heterogeneous mixture of processing cores (e.g., FPGAs, GPUs, DSPs) combined with a single- or multi-core CPUs.

Power constraints have become one of the most relevant design considerations for both classes of systems, even is now consider a dominant design consideration, due the increase in power density with every next technology node. Communication minimization, which is fundamental for reducing power consumption on embedded processors, is now a stringent requirement for next supercomputer design at the chip, node, and machine level [86]. The technology scaling, which resulted in process variation as well as component degradation, introduced the need of designing more flexible systems, able to deal with unpredictable behaviors (adaptivity). In the nano scale era reliability and predictability have become important metrics [116].

The design process of such increasingly new complex architectures requires great skills and significant programming effort, which is a time-consuming and error-prone task. Improvements in the design automation process is a challenge for both embedded systems and HPC scenarios. Most of the approaches used for the design of embedded systems have been also applied, in a way or another and at a different scale, to HPC systems [72]. As proposed in [48], a holistic approach that relies on tightly coupled hardware-software co-design methodologies is required for the design of modern embedded systems and supercomputers requires.

### 2.5.1 Parallelizing for ES and HPC

The development of parallelizing compilers have been focus for HPC, and they have followed, in one way or another, the steps aforementioned. Parallelizing for ES should consider some aspect usually not contemplated for HPC. Some differences are:

- Many of the HPC systems are distribute-memory systems, while mainly ES are shared-memory. This affect the way that information is shared and communicate among process or task executed in different PEs.
- ES exhibit a wide variety of PEs, so harnessing all the computational power available requires the capacity to map executable code to heterogeneous cores, even architectural different. HPC usually present a large number of cores of the same type. This affect the number of parallel task concurrently executed.
- Objectives in parallelization for ES go beyond than speed-up. Due the constrained nature of ES, other objectives should be take in count such as time and energy. For instance, exploiting a high degree of parallelism (DoP) to enhance speed-up could cause an increase in energy consumption and task could end pretty soon considering the time requirement. Hence a trade decreasing the degree of parallelism and consuming less energy while being within time constraints could be a better bet.
- Due the diversity of ES, the target hardware characteristics should be consider as an input to the parallelizing compiler, which also applies to HPC systems. For example,



---

operation voltage, frequency and TDP (*Thermal Design Power*) parameters should be consider, so that parallelism is not exploit for long periods at maximum running capacity, exceeding thermal constraints and causing the activation of mechanism like DTM (*Dynamic Thermal Management*) to mitigate the emergence but throttling down the execution and compromising the total reachable speed-up [104, 97].



# Chapter 3

## Representation Models

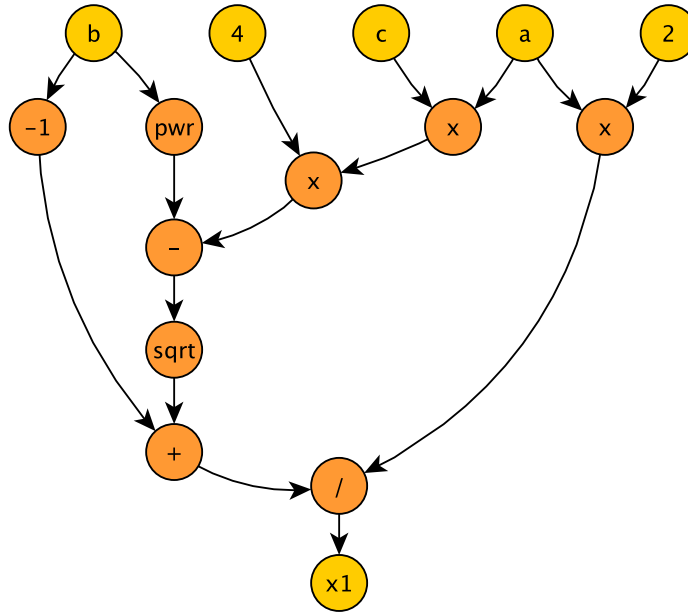
Computational or representation models are abstractions of conceptual notions, used to express the function of a system. For example, a finite-state machine (FSM) can represent a sequential logic circuit. Most of the graphs used in code analysis are based on *direct graphs*. According to [6], a direct graph  $G$ , where  $G = (B, E)$ , is composed by a set of *blocks* or *nodes* (also called *vertices*),  $B = b_1, b_2, b_3, \dots, b_n$ , and a set of *edges* (arcs or arrows),  $E = (b_i, b_j), (b_k, b_l), \dots$ . The nodes, not necessarily distinct, are connected by an edge, e.g., the edge  $(b_i, b_j)$  indicates that there is a link from node  $b_i$  to  $b_j$ , where  $b_i$  is the predecessor of  $b_j$  and  $b_j$  is the successor of  $b_i$ . Several types of graphs are used to perform code analysis to detect and extract parallelism, and perform code transformation and optimization. As mentioned by [144], data flow models naturally expose the parallelism contained in an application.

*Intermediate representation* (IR) is a data structure that is constructed from input data to a program. IR is used to represent the original program in an equivalent form, more suitable for exposing application features, such as parallelism level [86]. IR allows to match with the inherent parallel nature of hardware cores. Most of the direct graphs presented in this Chapter are created from IR in parallelizing compilers and approaches.

Computational structures consists basically of two parts: data and control. Each of them can be represented by a direct graph, a *Data Flow Graph* (DFG) and a *Control Flow Graph* (CFG), also called precedence graph. These two models are presented first in this Chapter.

### 3.1 Data Flow Graph

A *Data Flow Graph* (DFG) comprises a set of nodes, containers or storage cells ( $S$ ), and operators ( $P$ ), and a set of edges that indicates the input and output storage cells related to operators [135]. The nodes of a DFG perform operations when the data values are available, following the *firing rule* [133].



**Figure 3.1:** DFG for a quadratic equation solution.

The DFG represents global data dependence at the operator level (called as the atomic level in [78]). The execution process proceed with fetching data form the input port and the result is forwarded through each output port; a producing node send the data to the consuming node through edges. Due the partial ordering in a DFG, the nodes are not strictly dependent on each other for their execution of instruction [118]. For instance, consider a quadratic equation,  $ax^2 + bx + c = 0$ , with all coefficients different to zero, and whose solutions can be calculate as:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3.1)$$

A DFG for the calculation of solution  $x_1$  is depicted in the Figure 3.1. Here,  $S$  includes the input values and the output cell ( $S = \{a, b, c, 2, 4, x_1\}$ ); while  $P$  includes the operations used in the calculation ( $P = \{-1, pwr, sqrt, +, -, x, /\}$ ).

DFGs have been extensively used due their ability to specify algorithms which exhibit high degree of parallel and asynchronous activity. DFG have been used to simulate hardware. One advantage of DFG is its compactness and general amenability to direct interpretation. Algorithms expressed in a DFG are controlled by the arrival of data (token) at the transformation actors (nodes) [57]. Nevertheless, a complete analysis of algorithms requires the knowledge of the control flow of the program.

## 3.2 Data Dependence Graph

*Data Dependence Graph* (DDG) is related with DFG. A DDG represent the dependencies between data used in different instructions or statements. The edges represent dependencies, the same presented in 2.2, i.e., true dependence, antidependence, and output dependence [79]. The nodes represent different instructions. Consider the imperative example code, in language C, presented in Listing 3.1. The DDG for the sample code is represented in Figure 3.2 (the instructions are numbered according to the line numbers in the Listing). The graph shows data relationships for all the instructions, even in the output value (instruction 23), that depends of prior calculated data. The edges present the data dependence type among nodes.

Data dependence graphs have provided some optimizing compilers with an explicit representation of the definition-use relationships implicitly present in a source program [44]. Some DDG have been design for hierarchical analysis of dependence relations in programs, and those graphs are threaded through a syntactic representation of the program as a multi list of tokens.

As stated in [98], this representation is a data structure that can be rapidly traversed to determine dependence information, and it is commonly used by compilers that perform wholesale reorganization of programs. DDG have been used to represent only the relevant data flow relationships of a program [44], but lack of control information. DDG is not an executable representation and does not incorporate information about the control flow of the application [98].

## 3.3 Control Flow Graph

A *Control Flow Graph* (CFG) is a direct graph in which the nodes represent basic code blocks and the edges represents control flow paths [6]. A CFG is defined as a directed graph augmented with a unique entry node *Entry* and a unique exit node *Exit*<sup>1</sup>, such that each node in the graph has at most two successors. Nodes with two successors have attributes *T* (*true*) and *F* (*false*) associated with the outgoing edges in the usual way[44]. Considering two nodes in a CFG, *X* and *Y*, *Y* is control dependent of *X* if:

1. there exists a directed path *P* from *X* to *Y*, with any *Z* in *P* post-dominated by *Y*.
2. *X* is not post-dominated by *Y*.

A node *Y* is post-dominated by a node *X* in a CFG if every direct path from node *Y* to *Exit* contains *X*. This definition does not include the initial node on the path. A node never post-dominates itself. If *Y* is control dependent on *X* then *X* must have two exits.

---

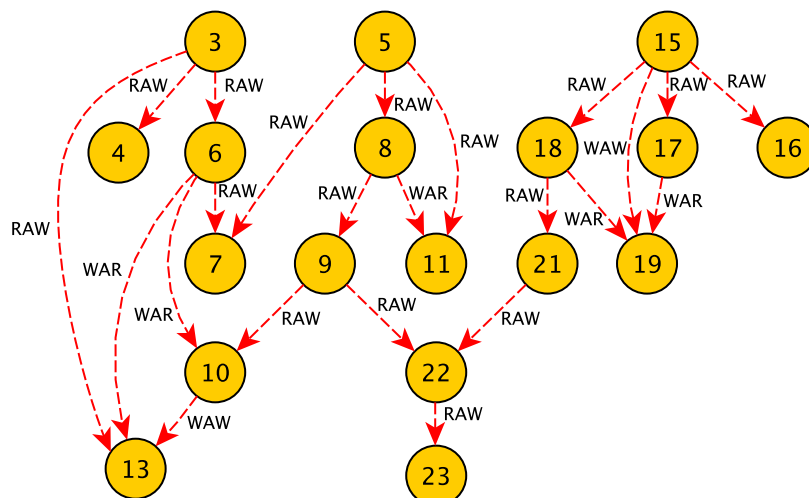
<sup>1</sup>These nodes are usually named as Start and Stop.

```

1 void code(int a, int b, int c, int * out){
2   int i, j, n, t, k, z, res;
3   i = 0;
4   while(i < a){ // loop1
5     j = 0;
6     n = i + 1;
7     while(j < n){ // loop2
8       t = j + 1;
9       c = a + t;
10      i = i + c;
11      j = j + 1;
12    }
13    i = i + 1;
14  }
15  k = 0;
16  while(k < 10){ // loop3
17    if(a > k)
18      b = k + a;
19    k = k + 1;
20  }
21  z = a * b;
22  res = z + c;
23  *out = res;
24 }

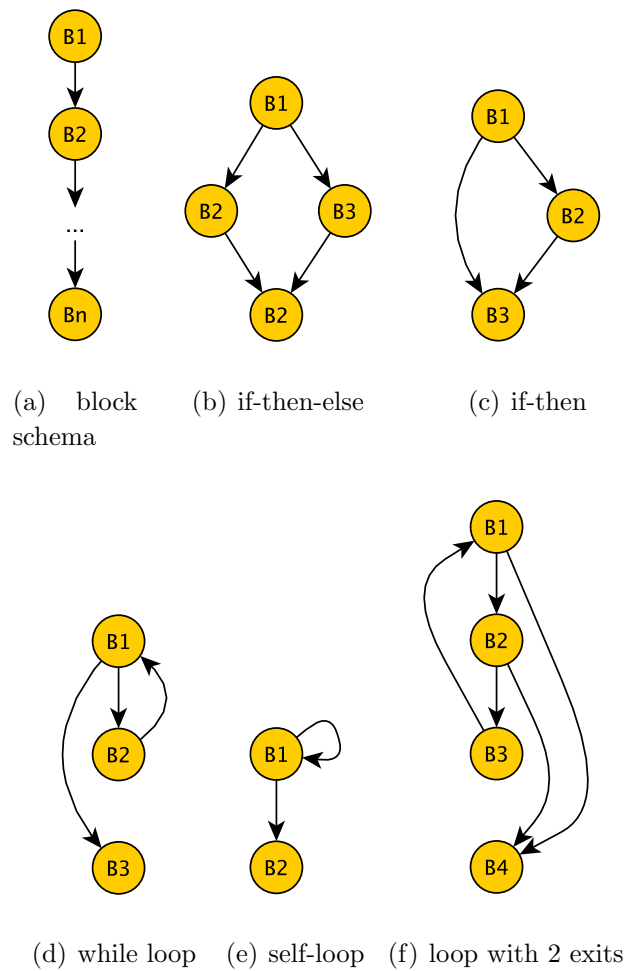
```

**Listing 3.1:** Example code 1 (Based on [86]).



**Figure 3.2:** DDG for example code in Listing 3.1 (Based in Figure from [86]).

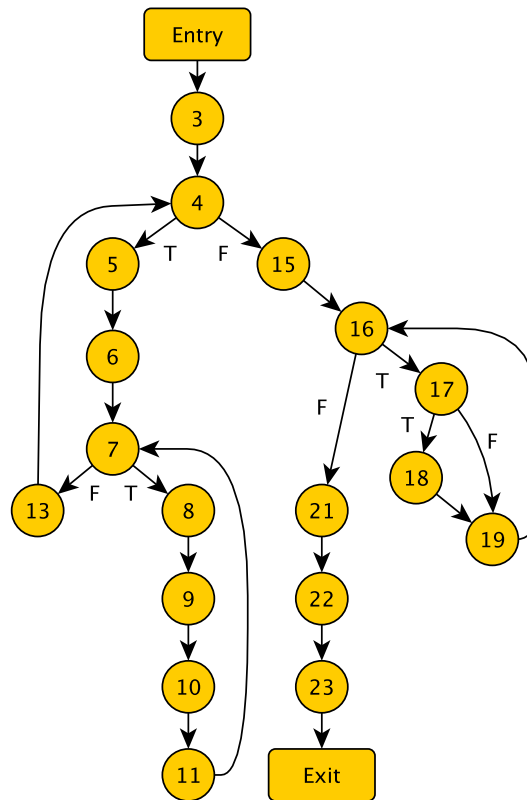
Following one of the exits from  $X$  always results in  $Y$  being executed, while taking the other exit may result in  $Y$  not being executed.



**Figure 3.3:** CFG representations for acyclic (a - c) and cyclic (d - f) code regions (Based on Figure from [40]).

Different code structures can be represented with a CFG, as depicted in Figure 3.3, where basic cyclic and acyclic code regions are illustrated with CFGs [40]. The Figure 3.4 presents the CFG for the code in Listing 3.1. In this example, the loops *loop1* and *loop2* are independent from *loop3*, since neither data nor control dependencies occur among the instructions in the corresponding bodies. The graph shows such independent loops are sequentialized through the edge between instructions 4 and 15 [86].

Most compilers store the programs in the form of CFGs, using the program's IR, and is used it for optimization purposes. CFG has been the usual representation for the control flow relationships of a program, the control conditions on which an operation depends can be derived from such a graph. An undesirable property of a control flow graph, however, is a fixed sequencing of operations that need not hold the usual way [44].



**Figure 3.4:** CFG for example code in Listing 3.1 (Based in Figure from [86]).

```

1 x = 1;
2 y = 2;
3 if (x == 1)
4     y = 3;
5 ...y... // represents an operation that involves y
6 x = 2;
  
```

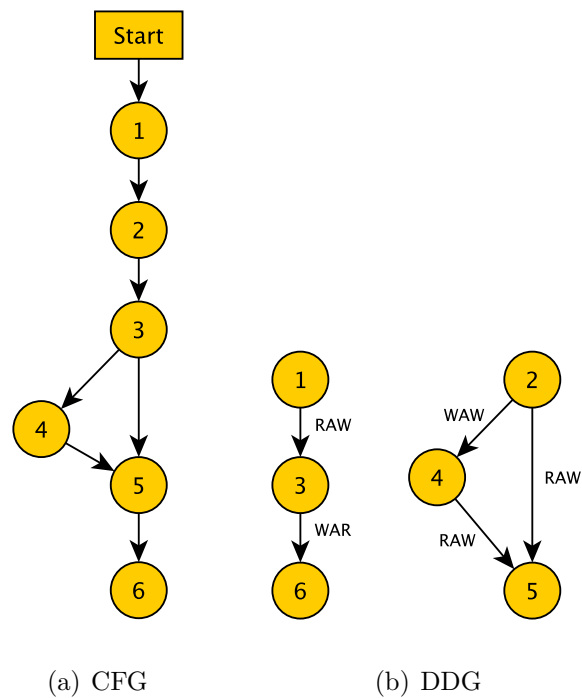
**Listing 3.2:** Example code 2 (Based on [98]).

## 3.4 Dependence Flow Graph

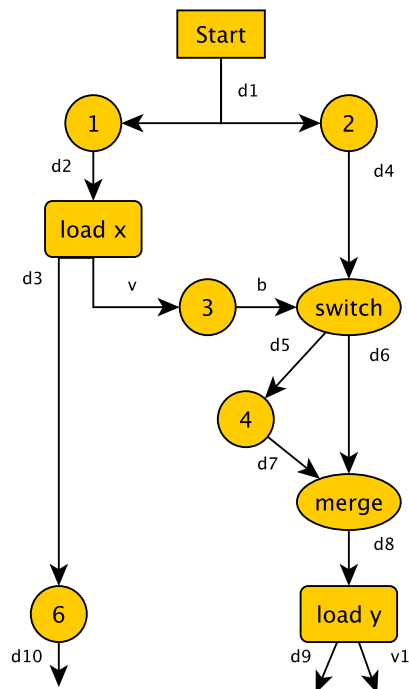
*Dependence Flow Graph* (DepFG) is model that allows the representation of a program proposed by [98]. DepFG is a data structure that can be rapidly traversed to determine dependence information and is a program in its own right, with a parallel, local model of execution. This representation naturally incorporates the best aspects of many other representations such as data and program dependence, static single assignment form and data flow programs graphs.

To describe this model, consider the small section of code in Listing 3.2. The Figure 3.5 presents the CFG and the DDG for this code. The Figure 3.6 depicts the corresponding DepFG.





**Figure 3.5:** CFG and DDG for code example in Listing 3.2 (Based on Figure from [98]).



**Figure 3.6:** DepFG for example code in Listing 3.2 (Based in Figure from [98]).

DepFGs are a synthesis of ideas from DDG and the data flow model of computation. As in the DDG, the DepFG can be viewed as a data structure in which the edges represent dependencies between the operations. For every dependence edge in the DDG, presented

in Figure 3.5(b), there is a corresponding path in the DepFG in the Figure 3.6. However, unlike DDG, DepFG are executable, and the execution semantics, called dependence-driven execution, is a generalization of the data-driven execution semantics available in DFG. In the latter, nodes represent functional operators that communicate with each other by exchanging value-carrying tokens along the edges of the graph; these edges can be considered as flow dependencies since they connect a node that produces a value with one that consumes it. These edges are called functional dependencies. In Figure 3.6, the edges names as  $v$ ,  $v1$  and  $b$ .

In DepFG, the data flow model is extended by adding an imperative, and updatable, global store and two operations called **load** and **store** which manipulate it. The **load** operator reads the contents of a storage location and outputs the value as a token. The **store** is the opposite than **load**, it receives a value on a token and stores it into a memory location. DepFG introduces new type of edge called imperative dependence. In the DepFG in the Figure,  $d2$  and  $d3$  are imperative dependencies that sequence operations on location  $x$ , corresponding to edges in the DDG. To preserve the token-pushing semantics of DFG, the load and store operators produce a special token when the operation is completed. These tokens flow down the imperative dependence edges to enable operators at the destinations of those edges. For instance, when the assignment of 1 is done to  $x$  it produces a token carrying on  $d2$ . This is said to satisfy the dependence  $d2$ , thereby enabling the **load**  $x$  operator for execution. When the **load**  $x$  executes, it produces tokens carrying on line  $d3$  and the value 1 on the line  $v$ . Thus operations on a given memory location are sequenced, but operations on different locations can be executed in parallel [98].

Imperative dependencies, denoted as  $d$  in the Figure 3.6, are classified as true dependence, antidependence and output dependence (as presented in Section 2.2). For instance,  $d2$  is classified as true dependence and  $d3$  as an antidependence. Dependence edges that sequence operations on location  $y$  are intercepted by **switch** and **merge** operators, which implement flow control. These operators are used to combine control information with data dependencies, which is missing in representations as the CFG and DDG.

To complete the explanation of DepFG, it is useful to execute the graph depicted in Figure 3.6 by pushing tokens. Execution begins when the **Start** operator sends a token carrying to the store operations, i.e,  $x = 1$  and  $y = 2$ . Depending on whether the token received on edge  $b$  is true or false, the **switch** operator outputs the token it receives on  $d4$  in either edge  $d5$  or  $d6$ . In this example, the **switch** routes the token to  $d5$ , and the assignment of 3 to  $y$  is executed. The **merge** operator receives a token on either one, but not both, of its inputs, and simply outputs this token. For this case, a token carrying the value 3 will be generated on edge  $v1$  [98].

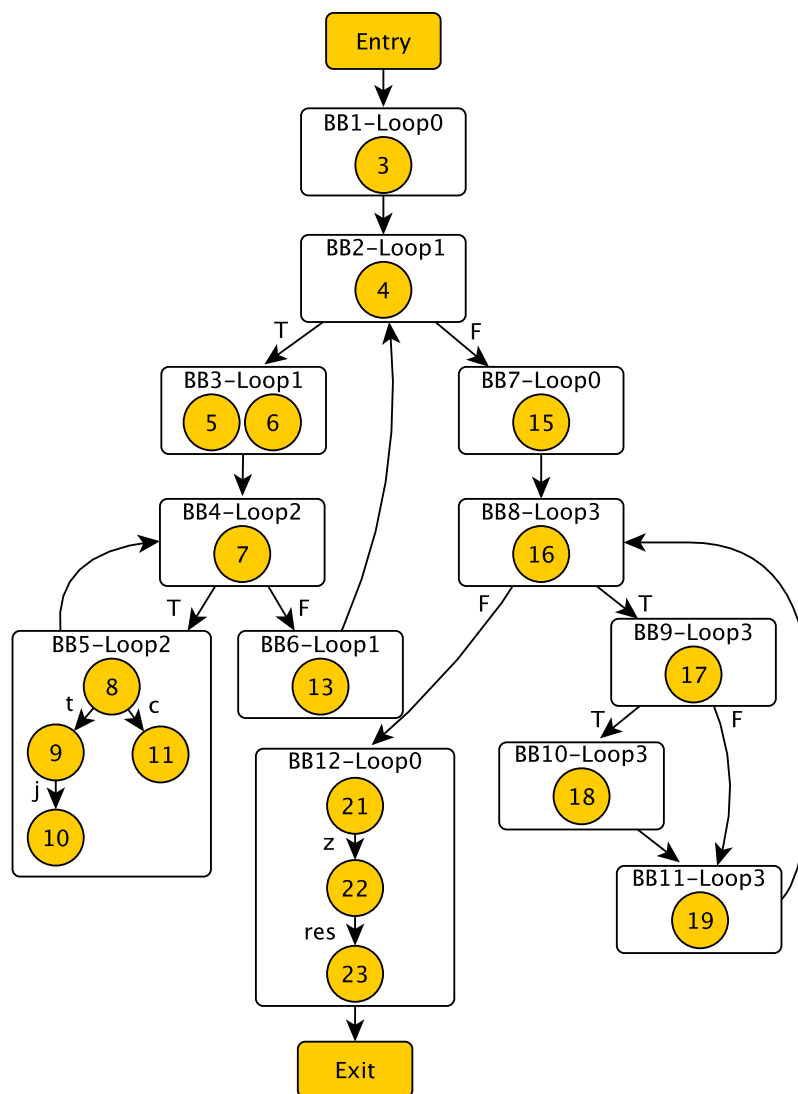


Figure 3.7: CDFG for example code in Listing 3.1 (Based in Figure from [86]).

### 3.5 Control Data Flow Graph

The *Control Data Flow Graph* (CDFG) is a basic block graph built by considering the data dependencies in the DDG and the control dependencies in CFG. In fact, the CDFG is based on the CFG [86]. The CDFG exposes the parallelism at basic block level, this is groups of instructions belonging to different basic blocks are sequentialized, while instructions belonging to the same basic block can be simultaneously executed, if they are data independent. The CDFG is a fundamental component of most compilers, where most optimization and design decisions are performed to improve frequency, power, timing, and area [143]. In Figure 3.7, the CDFG for the code in Listing 3.1 is presented, and the groups of instructions may be noticed.

CDFG provides the necessary information to determine whether a loop is parallelizable

or not [125]. To establish if a variable is privatizable it suffices to check if there are any incoming or outgoing data-dependence edges for this specific variable in the CDFG. In [125] an algorithm is presented to create a CDFG from the IR of a input source code, on which the parallelism detection is performed. The algorithm distinguishes between control and data flow items and maintains various data structures supporting dependence analysis. The control flow section constructs a global CFG of the application including call stacks, loop nest trees, and normalized loop iteration vectors [128]. The data flow section is responsible for mapping memory addresses to specific high-level data flow information. For this a hash table keeps the information of where data items are traced at byte-level of granularity. Data dependencies are registered as data edges in the CDFG. These edges are further annotated with the specific data sections, like array indices, that cause and preserve the dependencies. The profiling-based CDFG is the basis for the detection of parallelism in the Tournavitis approach [125].

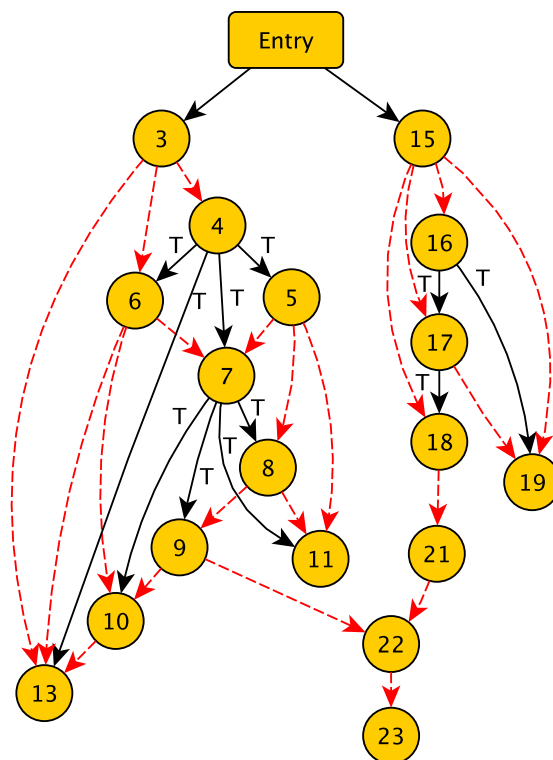
## 3.6 Program Dependence Graph

The *Program Dependence Graph* (PDG), proposed originally by [44], is a program representation that makes explicit both the data and control dependencies for each operation in a program. PDG represents a program as a graph in which the nodes are statements and predicate expressions (operators and operands), and the edges between nodes represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends. Control dependencies in PDG are introduced to similarly represent the essential control flow relationships of a program, and they are derived from the usual CFG.

PDG is an alternative to CDFG. Such graph contains the minimum data and control dependencies, thus not causing potential unnecessary sequencing. Figure 3.8 shows the PDG for the code in Listing 3.1, where the dotted edges represent data dependencies and the solid edges represent control dependencies.

The PDG explicitly represents both the essential data relationships, as present in the data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph. Since dependencies in the PDG connect computationally related parts of the program, a single walk of these dependencies is sufficient to perform many optimizations. These dependence relationships determine the necessary sequencing between operations, exposing potential parallelism [44]. Indeed, PDGs have been shown to be useful for automatic detection and management of parallelism and solving a variety of problems, including optimization, vectorization, code generation for VLIW machines, merging versions of programs [113].

PDG has been successfully used to extract parallelism. Nevertheless, since it does not contain control flow information at all, it lacks of the subset of control flow edges that ensure correct execution. This information associated to control constructs is needed



**Figure 3.8:** PDG for example code in Listing 3.1 (Based in Figure from [86]).

to accurately detect parallelism [86]. For instance, the instruction 21 in the PDG must be activated after the 18 due to a data dependence. Nevertheless, since instruction 18 belongs to a loop, the 21 must wait for the termination of *loop3* to guarantee correct execution. Such information, represented by the edge from 16 and 21 in the corresponding CFG (Figure 3.4), is missing in the PDG, which thus allows the instruction 21 to read the wrong value of the variable *b* (Listing 3.1). Another example is related with the instruction 16 in the example code which is data dependent on the prior instruction, the 15, and can be executed just after instruction 15 is executed. However, once the first iteration of *loop3* is terminated, instruction 16 must be reactivated. This information is represented in the CFG through the edge between 19 and 16, but it is lacking in the PDG.

The PDG effectively supports powerful program analysis by explicitly capturing dependence information between different program elements. Originally it was proposed for compiler optimizations, but the PDG has also been used as an effective foundation for debugging, testing, and maintenance. Originally constructed for procedural programs, PDG have been extended to support various object-oriented features such as classes and objects, inheritance, polymorphism, and dynamic binding [145].

```

1  for (i = 0; i < NUMAV; ++i) {
2      float sample_real[SLICE];
3      float sample_imag[SLICE];
4
5      int index = i * DELTA;
6      for (int j = 0; j < SLICE; ++j) {
7          sample_real[j] = input_signal[index + j] * hamming[j];
8          sample_imag[j] = zero;
9      }
10
11     fft(sample_real, sample_imag);
12     for (int j = 0; j < SLICE; ++j) {
13         mag[j] = mag[j] + (((sample_real[j] *
14         sample_real[j]) + (sample_imag[j] *
15         sample_imag[j])) / SLICE_2);
16     }
17 }

```

**Listing 3.3:** Section code of spectral benchmark [81].

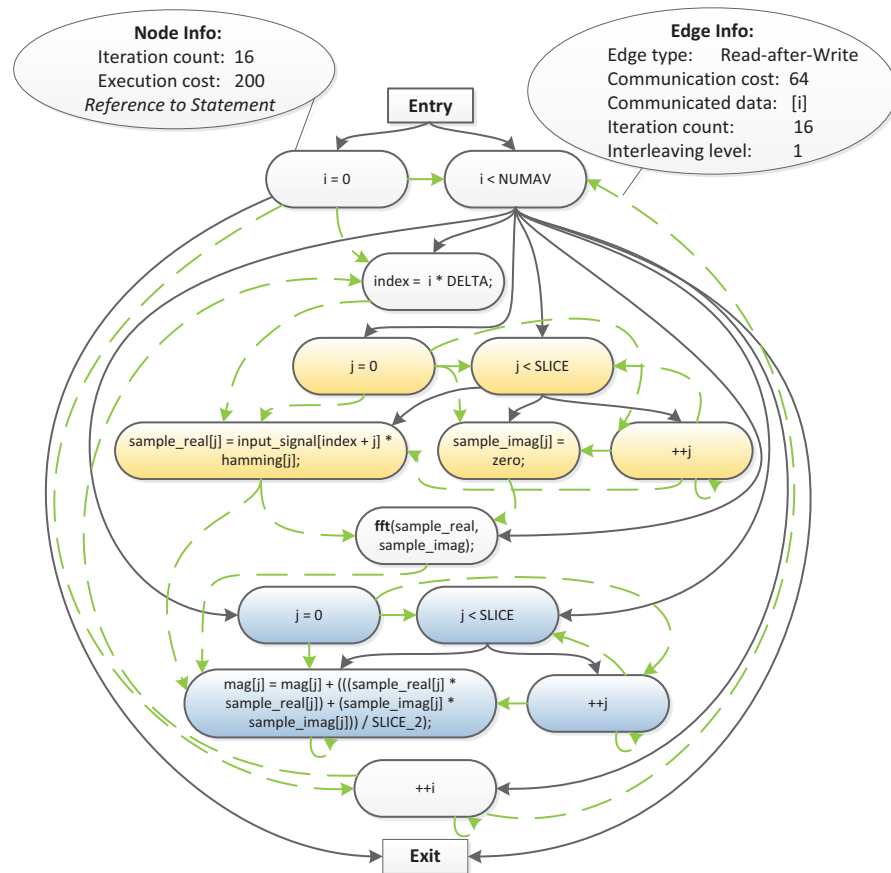
### 3.6.1 Augmented Program Dependence Graph

Related to PDG, an *Augmented Program Dependence Graph* (APDG) is presented in [33]. The PDG is extracted and augmented with cost information. As a regular PDG, the APDG combines control and data flow dependencies in one graph representation. In a APDG, each node of the PDG is augmented with the iteration count and execution costs of the statement represented by the node. The Figure 3.9 depicts the APDG for the code in Listing 3.3, where this node can be notice.

It is also essential to have information about the communication costs which have to be taken into account if the statements of the nodes are executed in separate tasks. Therefore, in the APDG, the edge type, communication costs, the communicated data, the iteration count as well as an interleaving level, describing the minimal amount of loop iterations which can be executed before the data is consumed at the target node, are annotated to the data dependence edges, as in the edge info node in Figure 3.9. Additional information like the estimated energy consumption or different objective values depending on the executing PE increases the value of one approach like this applied to embedded multi-core systems.

### 3.6.2 Parallel Program Graph

Also related with PDG, the *Parallel Program Graph* (PPG), introduced by [113], is a more general intermediate representation of parallel programs than Program Dependence Graphs. PPG contain *mgoto* edges that represent parallel control flow, and synchronization



**Figure 3.9:** APDG for example code in Listing 3.3 (Figure taken from [33]).

edges that impose ordering constraints on execution instances of PPG nodes.

As they execute, the PPG nodes perform read and write accesses to a shared memory. If a read and a write accessing to the same location are not properly guarded by **mgoto** edges or by **synchronization** edges, then the PPGs execution may incur in an anomaly access. If a read access is performed in parallel with a write access that changes the location's value, then the result of the read access is undefined. This representation uses the **mgoto** and **synchronization** edges as the only mechanisms available in the PPG for coordinating execution instances of PPG nodes.

A PPG can be built from a given PDG. The same set of nodes is used in both cases. The control and data dependence edges in the PDG correspond to **mgoto** edges and **synchronization** edges in the PPG. A control dependence edge from the PDG can directly be used as an **mgoto** edge in the PPG. A data dependence edge from the PDG can be used as a **synchronization** edge in the PPG by translating the context information to an appropriate synchronization condition on execution histories. However, not all PPG are derivable from PDG, this is the reason PPG is more general than PDG [113].

## 3.7 Hierarchical Task Graph

The *Hierarchical Task Graph* (HTG) is an intermediate parallel program representation, proposed in [49] and [101], which encapsulates minimal data and control dependencies, and which can be used for the extraction and exploitation of functional or task level parallelism. This is desirable when considering exploit parallelism at a coarse-grain for a better harnessing of computational capabilities in modern multi-core embedded systems. This representation model emerged as a proposal to exploit parallelism across loop and procedure boundaries.

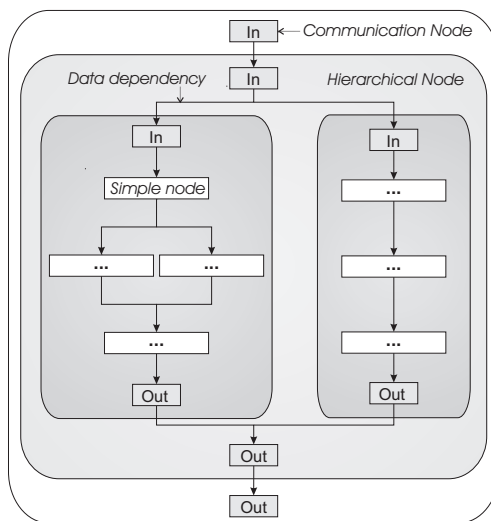
The HTG can be viewed as an IR of a parallel program, which encapsulates data and control dependence ordering as well as thread management and scheduling policies. Besides, as an abstract IR the HTG can be used as the code generation vehicle for a variety of processor architectures [101].

A HTG is a direct acyclic graph with unique nodes *start* and *stop*, such that there is a path from the *start* node to every node, and from every node to the *stop* node. The Figure 3.10 represents the basic structure of a HTG. The graph contains communication edges for data dependencies and four different kinds of nodes, this is:

- *simple* nodes: correspond to one basic statement in the original source code and they do not contain any child nodes (e.g.,  $a = b$ ),
- *hierarchical* nodes: loop or function bodies in the original code, the hierarchical nodes contain a communication in node, a communication out node and an arbitrary number of child nodes. These child nodes can be simple nodes or other hierarchical nodes.
- *communication in* nodes: are part of every hierarchical node. Communication from a node not contained in the hierarchical node to any inner node is redirected through this communication in node.
- *communication out* nodes, are also part of every hierarchical node. As in communication in nodes, the communication from a child node of the hierarchical node to any node not contained in the hierarchical node is redirected through this communication out node.

According to [101], a HTG can be constructed by the compiler. From natural loop recognition using CFG, the compiler constructs the loop level hierarchy. A loop and all statements immediately nested inside that loop constitute a node at a given hierarchy level. All nodes at a given hierarchy level together with all flow edges incident to nodes at the same level define the control flow graph at that level. The flow graph of a given hierarchy level is then represented as a single composite node at the next higher level of the hierarchy. On top, the entire program is represented as a single node, corresponding to the body of a fictitious loop with a single iteration, representing the main program. Using the control and data dependencies the task graph is derived separately for each hierarchy. The collection of task graphs at different hierarchies constitute the HTG. Due single entry and exit on different code regions, this allows parallelization in subregions in





**Figure 3.10:** HTG basic structure (Figure taken from [32]).

an isolated manner.

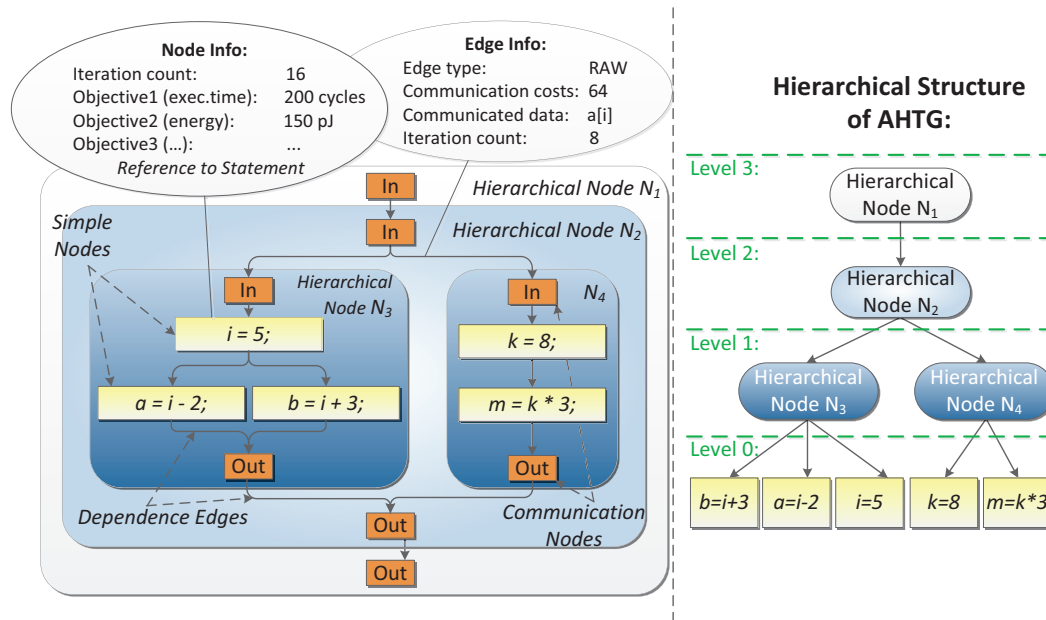
### 3.7.1 Augmented Hierarchical Task Graph

The *Augmented Hierarchical Task Graph* (AHTG), proposed in [33] and [32], is a HTG but with the addition of performance characteristics, all in one graph. Particularly, the classic HTG model is enhanced with special communication nodes to encapsulate the communication between different levels of the hierarchy. The communication between different levels of the hierarchy is always redirected through these special communication nodes, and this is used in the parallelization process in [32], because it enables the extraction of parallelism for each node separately.

Information from the graph structure containing nodes and edges is not enough to create well-balanced tasks from sequentially applications. Additional information about communication overhead and execution costs, is required particularly if two statements are executed in different tasks, as Figure 3.11 shows it. The nodes of an AHTG are augmented with additional cost information, such as:

- iteration count, which is equal to the iteration count of the statement represented by the node;
- objective values, such as execution costs in CPU cycles and the energy consumption in pJ are annotated to the nodes;
- and reference to statement, which keeps the relationship of nodes to the IR, used to adapt changes from the graph regarding the original application code and vice versa.

Essential information about the communication costs is also added to the edges, to augment the graph:



**Figure 3.11:** AHTG components and hierarchical structure (Figure taken from [33]).

- edge type: the dependence type (as presented in Section 2.2) is annotated to the edges, which is useful for parallelization purposes.
- communication costs: communication delay in CPU cycles and energy required to communicate the data, if the source and target node of the edge are executed in different tasks.
- communicated data: symbols or expressions that have to be communicated are annotated to the edges to be later available, especially for parallelization purposes.
- iteration count: number of times the communication takes place.

Even though AHTG present advantages for parallelism extraction, it is not the best IR to extract it from loops, especially those nested ones, for which PDG has a better performance. For instance, the APDG is a flat graph without hierarchical levels and communication redirection. Each statement of the loop to be parallelized is represented by a node in the PDG, and data and control flow edges are directly added between the nodes [33].

## 3.8 System Dependence Graph

The *System Dependence Graph* (SDG) is a direct graph, a representation model that combine PDG to model inter-procedural dependencies, developed by [63]. The SDG is an inter-procedural extension of the PDG. Each program consists of a main procedure and all auxiliary procedures. Procedures possess a distinguished entry node and an argument free return statement. All parameters are passed to by value-result. Compared to the previously presented models, the SDG allows parallelization of different procedures and the map of them into different PE, the compromise is present when dealing with the

dependencies and the communication and synchronization required.

The SDG are the basis for multiple applications in program analysis, such as slicing, debugging, testing and model-checking. E.g., SDG and precise slicing are used in a flow-sensitive, object-sensitive and field-sensitive information flow analysis for full Java Bytecode [52].

[83] presents a complete description of the SDG. This model contains one PDG for each procedure, and every PDG contains an entry vertex that represents the entry into a procedure. To model parameter passing, an SDG associates each procedure entry vertex with formal-parameter vertices: a formal-in vertex for each formal parameter of the procedure, and a formal-out vertex for each formal parameter that may be modified by the procedure.

An SDG associates each call site in a procedure with a call vertex and a set of actual-parameter vertices: an actual-in vertex for each actual parameter at the call site, and an actual-out vertex for each actual parameter that may be modified by the called procedure. At procedure entries and call sites, global variables are treated as parameters. These parameter vertices represent the assignments featuring the copy-in and copy-out scheme of parameter passing; with these vertices, a data flow analysis algorithm can be used to compute data dependencies among the parameters and statements in the procedures.

An SDG connects procedure dependence graphs at call sites. A call edge connects a call vertex to the entry vertex of the called procedure's dependence graph. Parameter edges represent parameter passing: parameter-in (parameter-out) edges connect actual-in; formal-in vertices (formal-out and actual-out vertices). For instance, the Figure 3.12 depicts the SDG for the code in Listing 3.4 (the nodes's name correspond to the ones in the code's comments). In this Figure solid lines represent control dependencies, dashed lines represent data dependencies, and dotted lines represent procedure calls and parameter bindings. In this example, the nodes represent program statements and parameter vertices. A particular parameter vertex is referenced by prefixing the parameter label with the call or entry vertex upon which it is control dependent (Table 3.1 presents the symbols and meaning used in the SDG example). For instance,  $C1 \rightarrow A1_{in}$  refers to the parameter vertex representing actual parameter *a* in the call to `proc1()` in *C1*. The statement *S4* is control dependent on the value of the predicate in *S3*; thus, there is a control dependence edge (*S3*,*S4*) in the SDG. The value of *b* in *S2* is passed into `proc1()` at *C1*; thus, there is a data dependence edge (*S2*, $C1 \rightarrow A2_{in}$ ) in the SDG. A parameter binding occurs between *a* in *main* and *x* in `proc1()` at the call to `proc1()` at *C1*; this binding results in parameter-in edge ( $C1 \rightarrow A1_{in}$ , $E1 \rightarrow F1_{in}$ ) and parameter-out edge ( $E1 \rightarrow F2_{out}$ , $C1 \rightarrow A2_{out}$ ).

For a small code, as the one in Listing 3.4, the SDG is a big representation. Typically, for real application, the SDG becomes a graph too large to represent the entire system at once.

```

1 main (void) { // E0
2   int a, b;
3   a = 0; // S1
4   b = 0; // S2
5   proc1(a,b); // C1
6   proc2(b,1); // C2
7 }
8
9 proc1 (int &x, int &y) { // E1
10  if (y > 0) // S3
11    x = x + y; // S4
12  proc2 (y,1); // C3
13 }
14
15 proc2 (int &z, int w) { // E2
16  z = z + w; // S5
17 }

```

Listing 3.4: Example code 3 (Based on [83]).

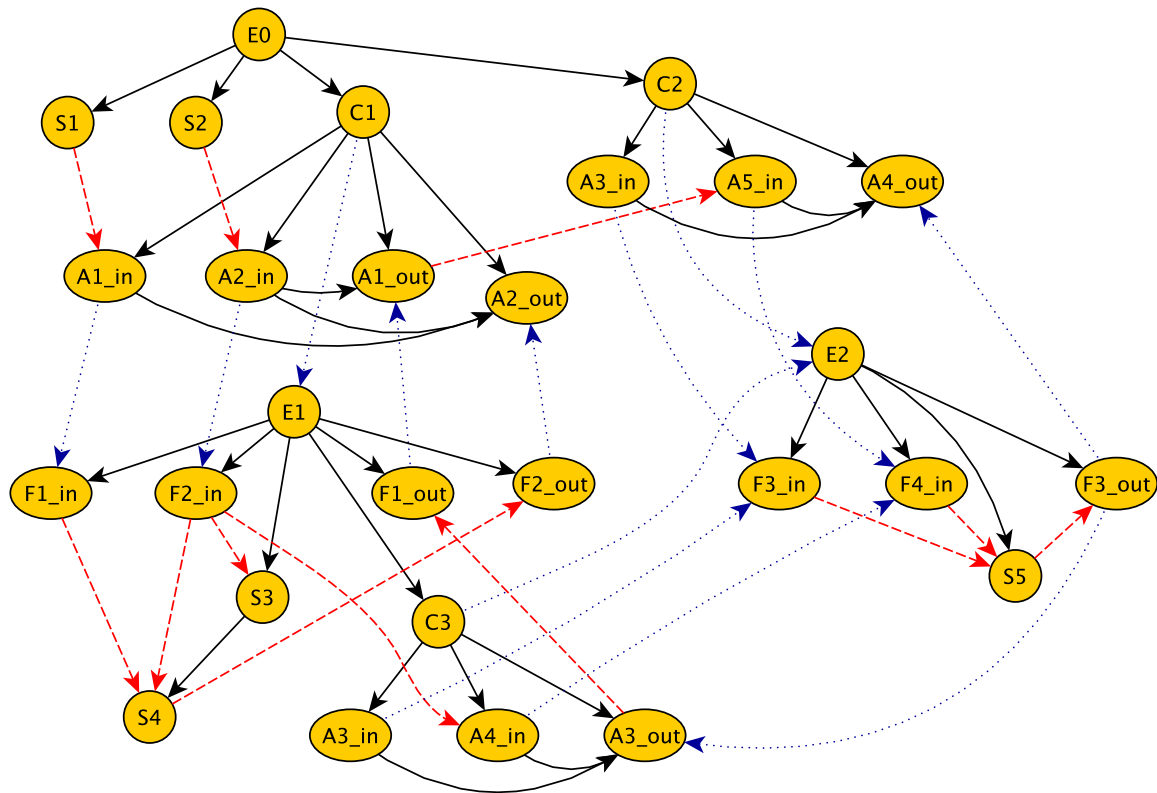


Figure 3.12: SDG for example code in Listing 3.4 (Figure taken from [83]).

**Table 3.1:** Auxiliar nodes.

In	Out
A1.in: $x_{in} = a$	A1.out: $b = y_{out}$
A2.in: $y_{in} = b$	A2.out: $a = x_{out}$
A3.in: $w_{in} = 1$	A3.out: $y = z_{out}$
A4.in: $z_{in} = y$	A4.out: $b = z_{out}$
A5.in: $z_{in} = b$	
F1.in: $x = x_{in}$	F1.out: $y_{out} = y$
F2.in: $y = y_{in}$	F2.out: $x_{out} = x$
F3.in: $w = w_{in}$	F3.out: $z_{out} = z$
F4.in: $z = z_{in}$	



# Chapter 4

## Code Analysis and Parallelizing Algorithms

Parallelizing compilers make use of different tools to detect and extract parallelism, even to optimize the code. Code analysis is required to determine and deal with dependencies in the code, and find where parallelism can be exploit without altering the program output. This code analysis can be done *statically*, without executing the code at compilation time; or *dynamically*, obtaining information at runtime, executing the program on a real or virtual processor. An *hybrid* analysis can be done mixing both static and dynamic analysis.

Algorithms have been developed and implemented to find parallelism in sequential code structures and generate a parallel code version, generally from the program intermediate representation. This chapter presents some relevant algorithms developed in recent years.

### 4.1 Code analysis

#### 4.1.1 Static

One of the most wanted characteristics in a compiler is the capacity to automatically extract parallelism from sequential specifications. Initial works on coarse-grained parallelism extraction were based on traditional static compiler analysis [23]. The principal strategy for finding useful parallelism is to look for a data decomposition in which parallel tasks perform similar operations on different elements [74].

*Static* code analysis, known as source code analysis or static analysis, is a software verification activity for analyzing source code, generally for quality and reliability. This analysis allows the identification and diagnosis of errors such as overflows, divide-by-zero, resource leaks, and illegally de-referenced pointers [3]. Static code analysis is performed at compilation time, without executing the program under analysis, or developing test

cases. This analysis can find errors in the source code, but also can try to prove the absence of certain critical errors.

Static code analysis that is augmented with formal methods (abstract interpretation) can be an important tool for improving the quality of embedded software used in high-integrity software systems [3]. Ideally, a static source code analyzer should be integrated with the everyday compiler to maximize use and reduce complexity of the toolchain. In addition, integrated checking enables source code parsing to be performed only once instead of twice [77].

A common compiler generate warnings and errors for some basic potential code problems, such as violations of the language standard or use of implementation-defined constructs. In the other hand, a static source code analyzer performs a full program analysis, finding bugs caused by complex interactions between pieces of code that may be in different source files. Static analysis looks for different kinds of flaws, looking for bugs that would normally compile without error or warning. Some common errors that modern static source code analyzer detect are [77]:

- Potential NULL pointer de-references.
- Buffer overflow.
- Writes to potentially read-only memory.
- Reads of potentially uninitialized objects.
- Resource leakages, like memory leaks and file descriptor leaks.
- Use of deallocated memory.
- Out-of-scope memory usage (e.g., returning the address of an automatic variable from a subroutine).
- Failure to set a return value from a subroutine.
- Buffer and array underflows.

Static code analysis plays a important role for parallelism extraction purposes. In [74], a wide variety of techniques are presented to manage dependencies, transform code, and detect parallelism<sup>1</sup>. For instance, part of the dependence testing is the process of determining if two references to the same variable, in a given set of loops, might access the same memory location. This implies different code transformations that support dependence testing, such as:

- *loop normalization*: a transformation that makes a loop run from a standard lower bound to an upper bound in steps of one,
- *constant propagation*: replace unknown variables with constants known at compile time,
- *induction-variable substitution*: eliminates auxiliary induction variables, replacing them with linear functions of the standard loop induction variable.

For parallelism detection, Kennedy and Allen [74] present techniques to deal with fine-

---

<sup>1</sup>For a complete review of this techniques, the reader should consult the result. In this work, some of the techniques are briefly presented.



and coarse-grained parallelism. Enhancing fine-grained parallelism, found primarily in innermost loops, a considerable amount of transformations are available, e.g, loop interchange, scalar expansion, scalar renaming, array renaming, node splitting, reduction recognition, index set splitting, symbolic resolution, and loop skewing. Having a large set of transformations provide several alternatives for exploiting parallelism. The adverse side is the complex task to choose the right transformation. There are at least two considerations that must be addressed in choosing transformations: making sure that the selected transformation actually improves the program over its original form, and making sure that selected transformation does not conflict or interfere with other transformations that offer more overall benefit for the program.

Coarse-grained parallelism, at the level of multi-core processors and using static code techniques, requires a different focus. Commonly parallelism is exploited in multi-core processors by creating a thread for each one of the PEs, executing the threads in parallel for a period of time with occasional synchronization, and synchronizing via a barrier at the end. To achieve a high degree of performance on such systems, it is needed to find and package parallelism with a granularity large enough to compensate the overhead of parallelism initiation and synchronization. [74] propose to focus finding parallel loops with significant amounts of computation within their bodies. This usually means parallelization of outer loops rather than inner loops, and it often means parallelization of loops with subroutine calls.

The challenge is to manage the trade-off between parallelism and granularity. For example, if there is insufficient parallelism, the multi-core system will not be effectively used. On the other hand, if the computation is too fine-grained, the start-up and synchronization costs of parallel execution will outweigh the performance gains. Thus, the challenge is to find parallelism at the coarsest possible granularity. In [74], three mechanisms are presented to address this challenge in three different loop contexts:

- In *single loops*, transformations including privatization, alignment, and replication, can be used to eliminate carried dependencies and thereby obviate the need for loop distribution, which reduces granularity.
- In *perfect loop nests*, loop interchange, loop reversal, and loop skewing are proposed to uncover parallelism and move it to the outermost position possible. The decision on how organize the loops for optimal performance is influenced by concerns related with architectural issues, such as performance of the memory hierarchy, and not only by the goal of maximizing parallelism.
- In *general loop nests*, which are not necessarily perfectly nested, multilevel loop distribution, followed by parallelization of the resulting loop nests and aggressive application of loop fusion can be used to extract the parallelism available and package it for high efficiency.

### 4.1.2 Dynamic

Researchers have realized that just static techniques for code analysis are not enough to optimally uncover parallelism. Different works, for HPC and embedded domains, have been developed, targeting on exploiting pipeline parallelism, and with less emphasis on coarse-grained data parallelism [23].

*Dynamic* code analysis uses code instrumentation to perform checks of the code as it executes [77]. For example, an instrumented program could check a pointer prior to be de-reference, in order to validate that it is not NULL. Some compilers have dynamic code analysis instrumentation available as a standard option. First the code is instrumented by the compiler, and then it is executed to obtain the desired information. During execution, the code calls a diagnostic function, provided by a library that is automatically linked to the program when using this option, which informs the user that a fault occurred, as well as the type and location of the error within the source code.

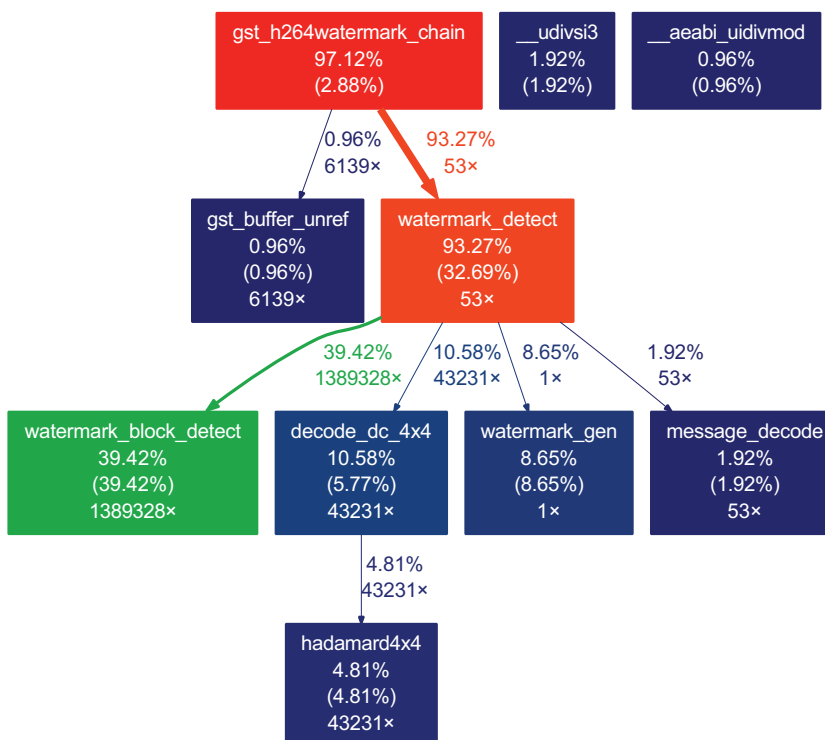
Dynamic analysis is able to detect dependencies that is not possible in static analysis, like dynamic dependencies using reflection, dependency injection, or polymorphism. Some dynamic analysis could also detect:

- Buffer overflows: when values are pretend to be stored in invalid array locations.
- Assignment bounds: when values assign to declare variables exceed the range of the variable type.
- Missing cases: in switch-case structures where the switch statement do not cover all the possible values of the control expression type.
- Memory leaks: a memory leaks occurs when a function allocates memory but never releases it, which can be detected with dynamic information.

Other important information can be obtained and analysis can be done using a dynamic approach, for instance, pointer analysis and dynamic dependencies to solve pointers, execution count and basic block count, loop execution count, code coverage, performance estimation and computation, applications timing (estimated or measured), detection of race conditions and potential dead locks, and profiling information, such as number of times a function is called and its proportional execution time respecting an entire program.

From the dynamic behavior of an application, it is important to identify the sections of code that could bring more benefit if parallelized, or that would bring a better resource compromise. Generally this sections are called *hotspots*. In the programming context, a *hotspot* is a region of a program that presents a high proportion of executed instructions respecting the entire program, or where most of time is spent during the execution of a program. Determine those sections in an application can help to make the decision of which section parallelize. Profiling tools can bring help in this task.

Profiling allows to obtain measurements like memory or time complexity of a program, the usage of particular instructions, the frequency and duration of function calls, among others. *Profis* is an example a of pro filer [53]. This tool creates a *call graph* with information



**Figure 4.1:** Diagram of the call tree for detection mode of the watermark algorithm, using block method.

such as the percentage of the total execution time of a function respecting the entire program, the number of times a function is called, the time spend, and the relations of time spent (in milliseconds) respecting the function per call, and a function and its descendants per call. A *call graph* is a direct graph that represents the calling relationship between subroutines in a program. Each node of the graph represents a procedure and the edges indicates that a procedure calls another procedure [75].

The Figure 4.1 represents a call graph for the detection mode of a watermark detection algorithm, based in the implementation of [146], which consist in insertion and detection of watermark on H.264 coded video. The graph is created with gprof and Gprof2Dot [46]. The Figure shows the procedures in each node with its function name, total percentage of the running time spent by a function alone and with all its children, and total number of times that a function is called, including recursive calls; the edges present information regarding the percentage of the running time transferred from the children to this parent and the number of calls the parent procedure called the children. From this example, it can be noticed that the `watermark_detect`, and particularly the `watermark_block_detect` consumes most of the execution time and it is called much more times than any other function. `watermark_block_detect` becomes a hot spot and a good candidate to be parallelized.

One key difference between a static and dynamic code analyzer is the how depth the code review process is done. By default, static code analysis combs through every single

line of source code to find flaws and errors. For dynamic analysis, the lines of code that get reviewed depend upon which lines of source code are activated during the testing process. Unless a line of code is interacted with, the dynamic analysis tool will ignore it and continue checking active codes for flaws. Static analysis provides a confirmation that each and every line of source code has been thoroughly inspected [16], while dynamic guarantees just the revision of those lines executed.

A profile based approach has some advantages compared to a static analysis. For example, it delivers very detailed information about access patterns and very fine grained dependency information. Thus, by using a profiling based approach, it is possible to identify for example that loop iterations modifying arrays or pointers are independent and therefore possible parallelization candidates. Usually, this is very hard to detect via static analysis techniques [32].

From the perspective of parallelism extraction, different approaches have used dynamic analysis to detect parallelism. For instance, the work in [126, 128, 127] employs profiling information to exploit and extract pipeline parallelism in multimedia applications, based on intermediate representation profiling and a hierarchical whole program representation. In this approach the sequential source program written in C is processed by the CoSy C compiler [42] and its IR is instrumented. The resulting executable is executed with one or more representative input files, and the results, a set of trace files, is presented to a trace analyzer for the dependence analysis. Due profiling for data and control dependencies is inherently unsafe, it is important to mention that a profiled based technique can not guarantee correctness.

In [107] profiling information is also used to uncover pipeline parallelism in general purpose applications. The measuring of data dependencies in a profiled execution of a program is used to determine control flow and data flow. Even though it is not clear the profiling tool employed, the profiled information is used to construct a function call graph and an interprocedural data flow graph to detect function-level parallelism. In this approach the functions in a program are clustered such that strongly dependent functions are members of the same clusters. TLP between function clusters is detected by analyzing inter-cluster data flow.

### 4.1.3 Hybrid

Automatic parallelization can not be achieved because classic compiler analysis are not powerful enough and the program behavior depends in many cases of the inputs. An *hybrid* code analysis implies the combination of static and dynamic techniques to detect and extract parallelism.

For instance, in [108] a hybrid approach is presented, a *Hybrid Analysis* (HA), which unifies the two approaches into framework, modifying both methods in order to integrate them seamlessly. So instead of only answering the question of whether an optimization is legal it also generates the dynamic conditions under which it could be legal. The HA

statically verifies memory reference properties, and when this is not possible, generates the conditions for which these properties can become true during program execution. These parallelization conditions can be evaluated dynamically with potentially little overhead and predicate the execution of the optimized parallelized blocks of code or loops. The Hybrid Dependence Analysis represents a process that extracts independence conditions from dependence equations at compile-time and then evaluate them efficiently at runtime. It does not only validate transformations but also generates sufficient, simple conditions which can validate optimizations at runtime.

## 4.2 Algorithms

Over the years, different algorithms and techniques have been developed to extract parallelism and generate parallel code. Most of the early effort was dedicated to extracting parallelism from regular nested loops. Loop parallelization algorithms have worked in finding a favorable loop transformation that reveals parallelism, but keeping in mind that generating parallel loops is not sufficient for creating efficient parallel executable programs. The design of these parallel loop detection algorithms considered many other optimizations related to the granularity of the parallel program, data distribution, and communications, to name a few. For instance, many loop transformations have been proposed to change the enumeration order so as to increase the efficiency of the code [22]. Some *classic* algorithms for loop parallelization are: Lamport [80], Feautrier [43], Allen and Kennedy [7], Wolf and Lam [140], and Darte and Vivien [35].

The complexity of the programs to parallelize and the emergence of multi-core systems, have motivated the development of new algorithms capable to produce parallel code that better harness the computational power available. Some of these algorithms use artificial intelligence algorithms as a base to find optimal solutions to the parallelism extraction challenge. This section presents four recent algorithms based on Machine Learning techniques, Integer Linear Programming, Genetic Algorithms and Decoupled Software Pipelining. This section does not pretend to cover all existing algorithms developed to parallelize code, but aims to present some with important impact to multi-core embedded systems.

### 4.2.1 Machine learning

In [128] a *Machine Learning* based parallelism mapping is presented, to exploit DLP. The parallelism mapping stage in this approach is responsible to decide if a parallel loop candidate is profitable to be parallelize, and to select a scheduling policy from the four options offered by OpenMP: cyclic, dynamic, guided and static. Determine which parallelizable loop brings advantages is a challenge. Incorrect classification would produce slowdowns due synchronization overhead.

This machine learning technique is based on Support Vector Machines (SVM) to determine if parallelize a loop candidate or not, and how it should be scheduled. The SVM classifier is used to construct hyper-planes in the multi-dimensional space of program features to identify profitably parallelizable loops. The classifier implements a multi-class SVM model with a radial basis function kernel, which is capable of handling linear and non-linear classification problems[21].

Program features are extracted to describe the relevant aspects of a program and present it to the SVM classifier. The static features: IR instruction count, IR load/store count, IR branch count, and loop iteration count; are obtained from the CoSy internal code representation. These features characterize the amount of work carried out in the parallel loop. The dynamic features: data access count, instruction count, and branch count; capture the dynamic data access and control flow patterns of the sequential program, and this information is obtained from the same profiling execution that this approach does for parallelism detection [128].

Off-line supervised learning is used. Pairs of program features and desired mapping decisions are used for the training. These are generated from a library of known parallelizable loops through repeated, timed execution of the sequential and parallel code with the different available scheduling options and recording the performance on the target platform. Once the prediction model is built using all the available training data, no further learning is done.

When new programs, with parallel annotations, are treated with this approach, some steps are required to follow [128]:

- Feature extraction: collect the static and dynamic features from the sequential version of the program, which is done with the profiler, part of the approach.
- Prediction: for each parallel loop candidate the corresponding feature set is presented to the SVM predictor and it returns a classification indicating if parallel execution is profitable and which scheduling policy to choose.
- User interaction: in case that the parallelization appears to be possible and profitable, but correctness can not be proven by static analysis, the user's final approval is required.
- Code generation: OpenMP annotation with the appropriate scheduling clause is placed, otherwise delete if the parallelization does not present any performance improvement or the user rejects it.

### 4.2.2 Integer Linear Programming

[32] presents a parallelizing algorithm based on *Integer Linear Programming* and used to extract TkLP and PLP. This algorithm extracts parallelism from the source code of an application represented in an AHTG.

The parallelization process started by calling the parallelize function with the root node

of the AHTG. This parallelization function is recursively called in a depth-first-search-manner initially. As a consequence, the algorithm starts the parallelization step at the innermost nodes of the graph. Since moving one node to a separate task and wait for its completion is not something wise, these nodes are skipped in the parallelization algorithm and a solution set containing only the sequential solution of this one task is returned. The algorithm is then moving upwards in the graph hierarchy and will reach a hierarchical node. Since all child nodes have already been processed, parallel sets are available for them. These parallel sets are combined to be used as input for the integer linear programming based parallelization approach.

This parallelization algorithm brings the option to limit the number of generated concurrently executed tasks. The Integer Linear Programming based (ILP-based)<sup>2</sup> parallelization step is executed several times to compute solutions for maximum amount of tasks, and down to 2 tasks. This step is used to parallelize a node for a set upper bound number of concurrent tasks and the already generated parallel sets of its child nodes. It tries to minimize the critical path, or the most expensive path, within a hierarchical node, which means that the ILP-based approach aims for the minimization of the costs of the path from the hierarchical node.

Due every child node has parallel sets with different execution times, depending on the number of tasks, deeper in the hierarchy, the ILP-based solver is able to choose solutions with different granularity for the child nodes, as long as the maximum number of concurrent tasks is not exceeded. To adopt this approach to different hardware platforms, the authors proposed that the user can specify two parameters:

- Task creation overhead, added to the computed path for every created task. This parameter can be used to steer the granularity of the parallelization step, depending on the utilized hardware platform.
- Communication costs are needed if data is transferred from one task to another. The communication overhead can also be changed by the user to model different hardware platforms (communication takes place only at the beginning and at the end of a task).

This parallelization approach divides the hierarchical node into three sections. All statements of the first section are executed sequentially on the same processor, which started the execution of the hierarchical node. The second one, called parallel section, where different concurrent tasks can be executed. In order to leave such a parallel section, all tasks have to synchronize and all data has to be communicated back to the parent task. This last section is sequential, where statements without explicit data communication can be executed. This is something similar to OpenMP and MPA directives. The details of the Integer Linear Programming formulation are presented in the [32] and [33].

---

<sup>2</sup>Do not confuse, at this point, ILP with Instruction Level Parallelism.

### 4.2.3 Thread extraction algorithm

In [94], the author present an automatic approach to non-speculative *thread extraction algorithm* from applications loops, that uses Strongly Connected Component (SCC) and Decoupled Software Pipelining (DSWP). This algorithm partitions operations into pipeline stages and presents the following steps [94, 129]:

1. Constructing PDG. The PDG the loop being parallelized is constructed. This graph contains all register, memory, and control dependencies present in the loop.
2. Building the dependence graph. The algorithm operates on a PDG which is transformed into a directed acyclic graph by identifying and clustering the SCCs formed by data and control dependencies. To ensure that there are no cyclic cross-thread dependencies after partitioning, the SCCs would be the minimum scheduling units.
3. Thread partitioning. Ensure an acyclic partitioning by finding the SSC and creating the directed acyclic graph of them. Each SCC is allocated to a thread while ensuring that no cyclic dependencies are formed between the threads. SCCs are partitioned among the desired number of threads according to a heuristic that tries to balance the execution time of each thread
4. Splitting the code. Involves the computation and creation of relevant basic blocks for each previous partition, place instructions assigned to partitions in corresponding basic blocks, and fix branch targets.
5. Inserting the flow. The last step implies the insertion of **produce** and **consume**, two special instructions used to send and receive values respectively, to guarantee correctness of the transformed code.

DSWP is a non-speculative pipelined multi-threading transformation that parallelize a loop by partitioning the loop body into stages of a pipeline [129]. DSWP allows the extraction of pipeline parallelism, which is efficient for many embedded applications. Like conventional software pipelining (SWP), each stage of the decoupled software pipeline operates on a different iteration of the loop, with earlier stages operating on later iterations.

In this algorithm, the parallelism is detected by computing the SCCs, like cycles, on the PDG of a loop. Each SCC represents a group of instructions that are cyclically dependent. As such, they cannot be split across pipeline stages. The SCCs are clustered in pipeline stages using basic block execution frequencies and inter-SCC dependences in order to load balance the pipeline. Parallel-stage pipelines are possible when an SCC does not have a loop-carried dependence with itself.

DSWP can be applied efficiently to various loops, even if they are not operating on recursive data structures. The extracted pipeline stages are balanced by an heuristic which merges the node with the highest estimated cycles, extracted by profiling in the compiler backend, to the currently processed pipeline stage. This step is repeated until



the estimated cycles of the current partition reaches the overall estimated cycles divided by the number of extracted stages [33].

Unlike other attempts about automatically multi-threading sequential programs, this algorithm does not try to partition programs into totally independent threads. Instead, it pipelines programs into dependent communicating threads. This makes DSWP a very practical multi-threading technique. Nevertheless, there is an important challenge, DSWP operates on assembler level which drastically limits portability, readability and the possibility to present the extracted results, in a comprehensible form, to the user. Another challenge presented by this algorithm is that operates in a representation that is too low level to perform the data privatization [33].

In [129], this algorithm is extended. The proposal is based in adding speculation to DSWP and evaluates an automatic approach for its implementation. By speculating past infrequent dependencies, the benefit of DSWP is increased by making it applicable to more loops, facilitating better balanced threads, and enabling parallelized loops to be run on more cores. This approach focuses on breaking dependence recurrences, and by doing so, instructions that were formerly restricted to a single thread to ensure decoupling are now free to span multiple threads. If the dependencies are speculatively ignored, large dependence recurrences, in the forms of SCCs, may be split into smaller ones with more balanced performance characteristics.

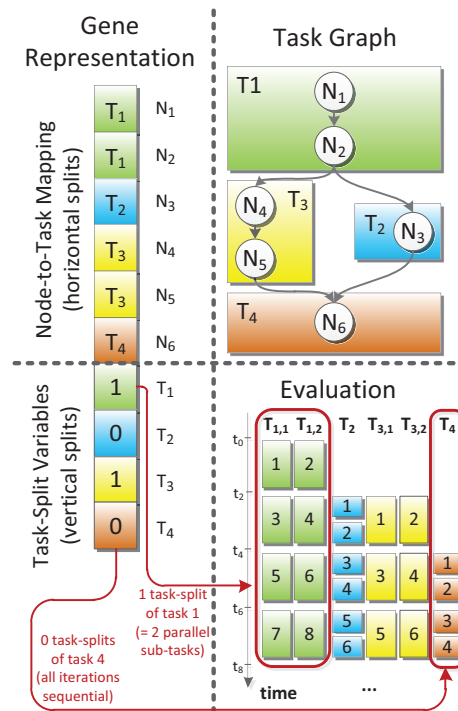
This work is also extended by [64], in something called DSWP+. DSWP+ looks beyond the performance benefits of DSWP, and explores DSWP as an enabling transformation for various loop parallelization techniques. Different than DSWP, which tries to maximize performance by balancing the stages of the pipeline, DSWP+ tries to assign work to stages so that they can be further parallelized by other techniques. After partitioning, DSWP+ allocates enough threads to the parallelizable stages to create a balanced pipeline.

#### 4.2.4 Genetic Algorithms

A *Genetic Algorithm* (GA) is implemented in [28] for automatic extraction of TkLP and PLP. This algorithm is able to extract pipeline parallelism from loops of sequential C applications considering multiple objectives, i.e., contemplating not only an optimization in the execution time but also, for instance, energy consumption or communication overhead.

Streaming structure found in many embedded applications, such as multimedia application, are best parallelized by extracting pipeline parallelism. In this approach, each loop is transformed into an APDG and processed by a GA. Then, a front of Pareto, with optimal solutions, represents the parallelized versions of the considered loop.

GAs are algorithms that facilitate the task of solving optimization problems in a multi-objective aware manner. GAs operates as follows: they start with an initial population consisting of individuals representing possible solution candidates for a optimization



**Figure 4.2:** Gene configuration in genetic algorithm (Figure taken from [28]).

problem. Each individual is characterized by a gene sequence called chromosome which describes the configuration of the optimization values. In each optimization step, some promising individuals are chosen to be recombined with other individuals to create a new population. This process is then repeated until a predefined termination criterion, for example a maximum number of generated populations, is met [28]. The challenge for using GA is to map the optimization problem to genes in a way that they can be efficiently evaluated for different objectives.

For genetic pipeline parallelization, disjoint pipeline stages of a loop which are executed in an interleaved way. This is done splitting horizontally the body of the loop. To increase the performance, the genetic algorithm should also be able to vertically split these pipeline stages to allow different loop iterations of the stage to run concurrently.

The Figure 4.2 shows the gene representation and its relationship with the task graph. Each chromosome is divided in two. The mapping of PDG nodes to tasks (nodes that represents different statements of the body of the loop) are placed in one section of the chromosome. Each node is mapped to exactly one task which represents one of the extracted pipeline stages due to horizontal splits. In the rest of the chromosome, an integer variable declares how often each task, pipeline stage, is split into sub-tasks which are executing different loop iterations of the stage in parallel.

Figure 4.2 shows how the nodes are mapped to task in this algorithm. For instance, nodes  $N_1$  and  $N_2$  are mapped to the first pipeline stage,  $T_1$ ,  $N_4$  and  $N_5$  are mapped to stage  $T_3$ ,  $N_3$  and  $N_6$  are mapped to stages  $T_2$  and  $T_4$ .  $T_1$  starts with the execution of the first iteration of nodes  $N_1$  and  $N_2$ . The generated data is sent to pipeline stages  $T_2$

and  $T_3$  allowing the next iteration of  $T_1$  to be executed concurrently respecting the first iteration of  $T_2$  and  $T_3$ . If one node is moved from one task to another, e.g. by mutating an individual, the dependencies between the stages may change which also influences the execution order of the tasks. Even small changes in the mutation steps may have big influences for the evaluation of different objectives.

Depending on the task creation and communication costs, one configuration of genes could represent a good solution candidate for the given example regarding executing time. This depends of the splitting of each stage into subtasks, and its behavior respecting to the objectives that the algorithms consider. The solutions from GA primarily depends on the population sizes used and the number of generated populations, the configuration of a chromosome must be evaluated very efficiently. This implies perform simulations for certain solution candidates, evaluation different objectives in the implementation, for instance, execution time, energy consumption and communication overhead. The evaluation of each of these objectives is explained in [28].

GA generates new solution candidates when values of the genes of existing individuals are modified. This is done by a combination of mutation (one gene of the chromosome is modified with a given probability, e.g. one statement is moved from one task to another) and recombination (cut the chromosomes of two individuals at a random position to combine the left-hand side of the first chromosome with the right-hand side of the other side and vice versa). Mutation and recombination are processed until a given number of populations are evaluated and the front of Pareto, with optimal solutions of the parallelized loop, is returned.

The GA is extended to HTG representations [30]. First the code is analyzed to extract necessary information to create the corresponding HTG. Once the graph is extracted, the parallelization process starts to extract parallelism in a bottom-up search strategy in the hierarchical structure of the graph. Each hierarchical node is processed in isolation. The GA moves child nodes of the hierarchical nodes to tasks which are then evaluated for all considered objectives.

A front of Pareto-optimal is obtained, with solutions for the different objectives, and it is attached to the hierarchical nodes. When all nodes on the same level of the hierarchy are processed, the parallelization algorithm continues with the parent node. There, the algorithm is also able to move child nodes to new tasks. The algorithm chooses one of the solutions of the front of Pareto of each child node which may contain additional tasks deeper in the hierarchy. This procedure continues until each node in the hierarchical task graph is processed and the top node is reached. The Pareto-front of parallel solutions is then returned to the user, whom can chose the solution with the best trade-off scenario.



# Chapter 5

## Parallel Programming Models and Parallelism Extraction Tools

Programming models represent an abstraction of the capabilities of the hardware to the programmer. A programming model is a bridge between the actual machine organization and the software that is going to be executed on it [39].

There are multiple general purpose parallel programming models that have been developed over the years, mainly created to target the HPC and desktop world. Some of the most used are models built on top of sequential languages like C, by providing libraries, like MPI and Pthreads, or language extensions, like OpenMP. The models are typically classified by the way tasks or processes interact, commonly, shared memory (e.g. OpenMP and Pthreads) and distributed memory (e.g. MPI) [23].

Most current parallelizing tools perform source-to-source transformations, this is, the original sequential code is transform into a parallel version. For this target, many tools employ code libraries and language extension commonly used and increasingly supported by different chip manufacturers in their tools. Due that most of the available tools and approaches generates parallel code, this chapter presents some of the most representative.

Since the parallelism extraction challenge has been on the research table for decades, many attempts to approach a solution have been developed over the years, each one with certain degree of success. In this chapter some tools are presented.<sup>1</sup> These tools could be considered to used them to generate parallel code for embedded systems, or as point of comparison to analyze and determine desired characteristics in the design and development of a parallelizing compiler for state-of-the-art embedded multi-core systems. The chapter ends with a taxonomy, a comparison table, of highlighted features in parallelizing tools presented throughout this document.

---

<sup>1</sup>It is not an exhaustive list of all the existing parallelizing tools, but some relevant are considered by the author due its possible application to embedded multi-core systems

## 5.1 Parallel programming models

The programming models in the embedded domain have been hardware or formalism centered. Hardware centered approaches aim for efficiency, require expert programmers, and expose architectural features in the language, such as memory and register banks. Programming models used by SoC companies have been typically hardware centered. In academia, formalism centered programming models are more common [23].

Programming models for embedded systems have been based on different Model of Computations (MoC). Early work implemented Synchronous Dataflow (SDF), Boolean Dataflow (BDF) and Cyclo-static Dataflow (CSDF). Other approaches have used Dynamic Dataflow (DDF), Petri-nets, and Kahn Process Networks (KPN). For instance, MAPS tool [24] currently uses C for process networks (CPN) as input language, an easy-to-use C extension that models concurrent processes and applications as well as legacy sequential code [136].

These models, based in states and processes, present good characteristics when designing and developing embedded systems. They have used by High Level Synthesis (HLS) tools, but they were not conceived to parallelize sequential code. When embedded multi-core platforms are available, these approaches, in most cases, requires re-factoring the input code, which is not the main idea for a parallelizing compiler.

Applications such streaming, pretty common in embedded systems, have motivated the creation of specialized languages, such as StreamIt [123, 34]. StreamIt is an architecture-independent language for high-performance stream programming. The MoC in StreamIt is SDF. In this model, the user implements independent actors, called filters, which translate data items from input channels to output channels. Filters are composed into graphs that represent the overall computation. One key property of this SDF is that the number of items consumed and produced during each execution of a filter is known at compile time, allowing the compiler to perform static scheduling and optimization. A language like StreamIt allows a better exploitation of PLP, which is propitious for exploiting multi-core embedded systems capabilities in real-life embedded applications.

In the other hand, even though the initial motivation for libraries and language extensions have not been multi-core embedded systems, they have reached the embedded world. Having standard Application Programming Interfaces (API) increases the portability of parallelized code across different embedded platforms, even from HPC to embedded systems. This could harness the available computational power in modern MPSoC. Clearly, the corresponding hardware and software support is required, at a bare-metal or operative systems level.

In this section, language extensions and libraries to write parallel programs are presented. Some of them are currently used by parallelizing tools and approaches to write the parallel outputs. According to [39], these extensions and libraries can be classified as shared-memory, distributed-memory, and heterogeneity models. This last term is due the nonautonomous origin of cores like GPU and GPGPU, that commonly are used together with a CPU that off-loads code regions to be executed in these cores.

### 5.1.1 OpenMP

OpenMP is an industry standard API for parallel programming [93], it provides portable high-level programming constructs that enable users to easily expose a program's task and loop level parallelism in an incremental fashion. The language, created in the late 90s, has continuously evolved and is currently at the version 4.0. OpenMP extends FORTRAN, C and C++ and was designed to make it straightforward for application programmers to create portable programs for shared memory computers (both symmetric multiprocessors systems (SMP) and non-uniform memory access (NUMA)) [119]. Shared memory models, as OpenMP, are based on the assumption that the execution entities, called workers, that carry the actual execution of the program instructions, have access to a common memory area, where they can store objects, uniformly accessible to all [39].

The basic idea with OpenMP is to incrementally add parallel constructs to the source code to convert a serial program into a parallel version, whereby directives can be added gradually. Hence, OpenMP constructs are designed to have a small (or even neutral) impact on the semantics of a program. OpenMP constructs are composed of a compiler directive and a block of code associated with that directive. In C and C++, the directives are in the form of code `pragmas` annotations. OpenMP presents three fundamental aspects [119]:

- Parallelism in OpenMP is explicit. The constructs are placed into the source code which tell the compiler how to turn the sequential program into a parallel program. While the compiler assists the programmer by managing the low-level details of how the parallelism is realized, it is up to the programmer to expose the concurrency in the program and express it for parallel execution.
- OpenMP is multi-threaded programming language extension. OpenMP assumes that an operative system will provide a set of threads that execute in a shared address space on behalf of the program. All of the issues common to multi-threaded programming environments such as race conditions, livelock, and deadlock apply when using OpenMP.
- OpenMP is fundamentally built on top of tasks. Tasks are the basic unit of computation when using OpenMP. A task defines the body of code to execute and the data environment required to support that execution. Tasks are defined by the programmer and scheduled for execution by the threads in the OpenMP program. Any time a thread is created, an implicit task is created as well, and this is tied to the thread, which means that the implicit task will only be executed by its associated thread.

OpenMP supports a parallelism model called fork-join. This is, the programs begin executing on a single, master thread which spawns additional threads as parallelized regions are encountered. Parallel regions can be nested although the compiler is not

required to exploit more than one level of parallelism. At the end of the outermost parallel region the master thread joins with all worker threads before continuing execution. A parallel region essentially replicates a job across a set of spawned threads. The threads may cooperate by performing different parts of a job through work-sharing constructs. The most prominent of such constructs is the `omp for` directive,<sup>2</sup> in C/C++, where the iterations of the adjacent loop are divided and distributed among the participating threads. This allows for easy parallelization of regular loop nests and has been the main strength of OpenMP since these loops are prevalent in scientific codes such as linear algebra or simulation of physical phenomena on rectangular grids.

OpenMP defines support for several features, among them [130]:

- Parallel regions: sections of code that can be executed in parallel by multiple threads.
- Synchronization primitives: co-ordination between parallel regions.
- Data-sharing attributes: OpenMP supports both thread-private and shared memory constructs, as well as directive-driven partitioning of data and variable values among threads.
- Nested parallelism, including task nesting.
- Thread pool sizing and management.

OpenMP-enabled platforms were uncommon in the embedded domain [23], however this had changed. Embedded multi-core chip vendors and developers are giving supporting to OpenMP in their tools and platforms. Nowadays many embedded multi-core platforms are OpenMP compliant. For instance, Texas Instruments offers support for the OpenMP API in its KeyStone multi-core architecture [66]. For the DSPs, TI has integrated OpenMP support into its optimized TMS320C66x compilers and DSP runtime software. For the Cortex-A15 processors, TI is leveraging industry standard GCC Compiler and Linux for OpenMP support. OpenMP is a powerful programming model for this architecture as it allows reuse of existing C/C++ software investment, leverages the ARM and DSP shared memory architecture for multi-core, makes use of dedicated DSP hardware for fast synchronization, and runs efficiently on industry-standard Linux and on TI's SYS/BIOS real-time operating system (RTOS). TI has defined OpenMP Accelerator Model, a subset of OpenMP 4.0 specification that enables execution on heterogeneous SoC with host CPUs and one or more on-chip target accelerators. For instance, in the TI 66AK2H MPSoC, the host is a Quad Core ARM Cortex-A15 cluster running SMP Linux and the target accelerator is a single cluster consisting of 8 C66x DSP cores, so the ARM's cores can offload computation, code and data, to the target accelerators [67]. This is truly convenient for programming and parallelizing code to multi-core heterogeneous systems.

For instance, the Figure 5.1 shows the speedup achieved using OpenMP directives for a simple  $\pi$  ( $\pi$ ) number computation, present in Listing 5.1 (the number of steps used was 10000 just to increase the workload). The sequential code was parallelized using available

---

<sup>2</sup>The details of the code implementations for OpenMP in C/C++ are not presented in this work, the reader may consult any of the abundant sources available.



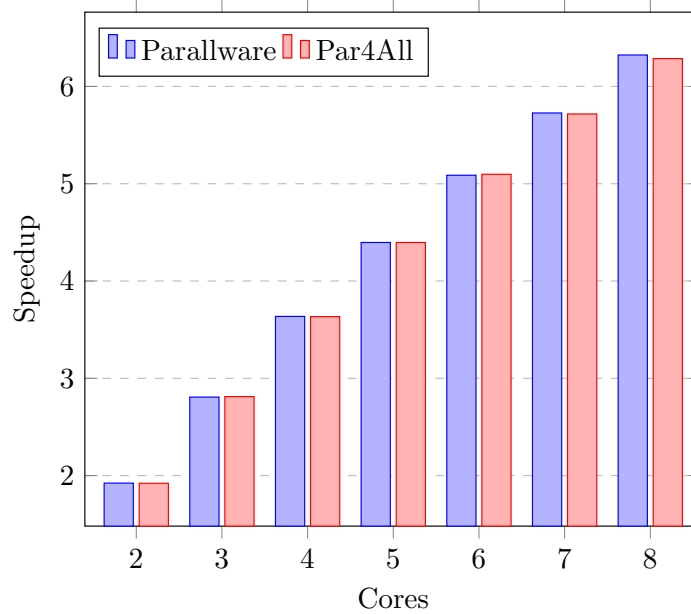
```
1 int pi(int num_steps, double * pi) {
2
3     double x;
4     double sum = 0.0;
5     double step = 1.0/(double) num_steps;
6
7     int i;
8     for (i=0; i < num_steps; ++i) {
9         x = (i+0.5)*step;
10        sum = sum + 4.0/(1.0+x*x);
11    }
12    *pi = step * sum;
13
14    return 0;
15 }
```

**Listing 5.1:** Code for pi ( $\pi$ ) calculation.

parallelizing tools, in this case Parallware [11] and Par4All [117], which produce OpenMP versions of the sequential input code (the parallel versions obtained are presented in the Appendix A). Many of the current existing tools and approaches for automatic and semiautomatic parallelization produce OpenMP versions from the sequential inputs, as presented further in this document. The KeyStone C6678 platform was used as target to execute these parallel versions, and the code was compile using the TI toolchain for the C66x. From the Figure 5.1, it can be observed that a speedup up to  $6\times$  can be achieve when using the 8 available cores and for both parallelized codes.

Due OpenMP was originally designed for HPC systems, there are some challenges that prevent the direct use for embedded multi-core systems. One of the hurdles is that embedded systems may lack some of the features which can be commonly found in general-purpose processors. For example, a traditional OpenMP compiler translates non-executable pragmas (parallel, for, and so on) into multi-threaded code with function calls to a customized runtime library, which typically relies on POSIX threads and SMP GNU/Linux. The example presented takes the advantage of an OpenMP-compliant operative system. Nevertheless, as exposed, this is not always the case for all systems in the embedded world. Some works have tried to target the issue of OpenMP support when no operative system is available.

In [134], the authors developed a runtime system to explore mapping of OpenMP on multi-core embedded systems, with no operative system. This effort, with the support of the OpenMP alliance, leverages the Multicore Association (MCA) APIs as an OpenMP translation layer. The MCA APIs is a set of low-level APIs handling resource management, inter-process communications and task scheduling for multi-core embedded systems. It is only required to maintain a single OpenMP code base which is compatible by various compilers, while on the other hand, the code is portable across different possible types



**Figure 5.1:** Speedup achieved for a simple pi calculation on TI DSP multi-core platform using OpenMP.

of platforms. Even though the worked is developed and testes using the Freescale QorIQ P4080 multi-core platform, the aim of is to have a flexible library that supports a wide spectrum of platforms of different vendors. This approach supports non-cache-coherent systems and its performance comparable to customized vendor-specific implementations. It implements OpenUH as the front-end source-to-source compiler, which translates the source code into C with OpenMP runtime function calls, employs a GCC back-end to generate the object file and libraries and finally generates executable files by linking the object file, the OpenMP runtime library developed, libEOMP, and the MCA runtime library.

Other approaches have addressed the problem for multi-core DSPs platforms. In [120] a work, at a bare-metal level (between the runtime and the device level), is presented for a OpenMP software stack to be used in the KeyStone II platforms from TI, which presents multiple ARM and DSP cores in the same chip. This work is particularly developed considering tasks running in ARM Linux-powered cores and the possibility to map parallel tasks, created through OpenMP, to the DSPs cores available. The executables for the DSP are compiled from the mapped code using the proprietary TI compiler for this cores. In [60] another approach to port OpenMP to DSPs is presented, targeting the Freescale MSC8156 chip, a DSP processor with six cores, The authors introduce OpenMDSP, an extension of OpenMP 2.5 for multi-core DSP, together with the implementation of a compiler and runtime system for the mentioned platform.

### 5.1.2 POSIX Threads

*POSIX Threads*, usually referred to as Pthreads, is a POSIX (Portable Operating System Interface) standard for threads. A thread is defined as an autonomous execution entity, with its own program counter and execution stack which is usually described as a lightweight process. A normal process (consider heavyweight) may generate multiple threads; while autonomous, the threads share the process code, its global variables, and any heap-allocated data [14]. The POSIX provides a standard interface to create threads and control them in a user program. POSIX is a standardization of the interface between the operating system and applications, mainly in the Unix-like operating systems, but it has reached other operating systems like many RTOS for embedded systems (e.g, FreeRTOS, ThreadX, QNX Neutrino RTOS, NuttX).

In parallel applications, multiple threads can be created with Pthreads, for shared-memory multi-processors. Because the execution speed and sequence of different threads is unpredictable and without order by default, programmers must be aware of race conditions and deadlocks. Synchronization should be used if operations must occur in certain order [39]. This is an important trade-off to consider, especially in time-constraint systems.

POSIX provides condition variables as their main synchronization mechanism. Condition variables provide a structured means for a thread to wait, i.e. block, until another thread signals that a certain condition becomes true. Real-time extensions to POSIX define barriers as an additional synchronization mechanism for Pthreads. Pthreads provide mutex objects as a primary way of achieving mutual exclusion, which is typically used to coordinate access to a resource which is shared among multiple threads. A thread should lock the mutex before entering the critical section of the code and unlock it right after leaving it. Pthreads also provide semaphores as another mechanism for mutual exclusion.

Pthreads have been widely used on HPC and embedded systems. Nevertheless, few works have attempt to use it as a final output in a automatic or semiautomatic parallelism tool. For instance, the automatic parallelization tool S2P, a source-to-source conversion tool developed by KPIT Cummins Infosystems Ltd. India. The author of this tool, according to [105] and [13], claim that is a fully automated parallelizing compiler that converts sequential C source code to a parallel multi-threaded source code by adding threading constructs of OpenMP and Pthreads. AESOP [2], an automatic tool generates parallel code using Pthreads calls.

Pthreads could be use for semiautomatic tools that only analyze and detect parallelism, and indicates to the user where potential parallelism could be exploited, so the insertion of threads directives can be done manually. To use Pthreads for parallelizing compilers is important to note that target platforms must support it, due these libraries usually rely on specific architecture or operative system; this is effective in a broad amount of current multi-core embedded systems, but not all. On the other hand, Pthreads puts too much burden on the programmer. Explicitly managing and manipulating the execution entities can sometimes give ultimate control to an expert programmer, but this comes at

the cost of increased complexity and lower productivity since larger Pthreads programs are considered hard to develop, debug, and maintain [39].

### 5.1.3 MPI

The message-passing model assumes a collection of execution entities, particularly processes, which do not share anything and are able to communicate with each other by exchanging explicit messages. This is a common and natural model for distributed-memory systems where communication cannot be achieved through shared variables. It is also an efficient model for NUMA systems where, even if they support a shared address space, the memory is physically distributed among the processors [39]. *Message Passing Interface* (MPI) is a specification for message-passing operations and is implemented as a library which can be used by C and FORTRAN programs [55].

An MPI program consists of a number of identical processes with different address spaces, where data is partitioned and distributed among different PEs. The interaction among the processes is done through messaging operations. MPI provides send and receive operations between a named sender and a named receiver, called point-to-point communications. These operations are cooperative or two-sided, as they involve both sending and receiving processes, and are available in both synchronous and asynchronous versions.

MPI is generally considered an efficient but low-level programming model. Like Pthreads, the programmer must partition the work to be done by each execution entity and derive the mapping to the actual processors. Different than Pthreads, there is need of partition and distribution of the data on which the program operates.

In the clustered computing world, MPI is the *de-facto* standard for message passing among the clusters of workstations. Additionally, there exist implementations that allow applications to run on larger computational grids. Respecting the embedded multi-core systems, there is a lightweight implementation specialized for this world, called Lightweight MPI (LMPI) [5] for heterogeneous distributed embedded system. Some other works have tried to address the implementation of MPI on embedded systems, such as LAM (Local Area Multicomputer) implementation [84], the MSG library and MSG-core layer [65], the TMD-MPI, a MPI-based programming model for MPSoC implemented in FPGAs [111] and later used in HPC Reconfigurable Computers (HPRCs)<sup>3</sup> [112]. Some implementations based on MPI have been worked for multi-processor signal systems [109, 110]. Also, there is a case of study about the design of a scheduler based on Open MPI for large-scale heterogeneous distributed embedded systems [71].

Although there are implementations and efforts to provide support for MPI in embedded systems, by the time this document was written there are not parallelizing tools or approaches that produce MPI-based parallel outputs from sequential input code.

---

<sup>3</sup>One or more standard micro-processors tightly-coupled with one or more reconfigurable FPGAs

### 5.1.4 CUDA

CUDA (*Compute Unified Device Architecture*) is a parallel computing platform and programming model created by NVIDIA and implemented by GPUs that this company produce [91]. With CUDA, GPUs can be used for general purpose processing, and not just for graphics purposes, which is known as General-Purposes Computing on GPU (GPGPU). Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly, which allows the exploitation of TLP [76].

Shared memory is a powerful feature for writing well optimized CUDA code [59]. However, with the demanding needs of more intensive workloads, porting GPU programs to more scalable distributed memory environment, such as multi-GPUs, has been worked [25]; but to achieve this, programs need to be re-written mixed, e.g, CUDA and message passing.

With CUDA, the threads are organized in a two level hierarchy, i.e. the block and the grid. The first is a set of tightly coupled threads where each thread is identified by a thread ID while the second is a set of loosely coupled blocks with similar size and dimension. The grid is handled by the GPU, which is organized as a collection of multiprocessors. Each multiprocessor executes one or more of the blocks and there is no synchronization among the blocks [39].

CUDA is implemented as an extension to the C language, which is beneficial for porting legacy sequential C programs. Tasks to be executed on the GPU, called kernels, are functions marked with qualifiers in the code. Thread management is implicit; programmers need only specify grid and block dimensions for a particular kernel and do not need to manage thread creation and destruction. The memory model of CUDA consists of a hierarchy of memories. In particular, there is per-thread memory (registers and local memory), per-block memory (shared memory, accessed by all threads in a block), and per-device memory (read/write global and read-only constant memory, accessed by all threads in a grid and by the host).

Even that GPUs first appeared in the HPC and desktop scene, state-of-the-art MPSoC includes several power efficient GPUs. For instance, the NVIDIA Tegra K1 SoC, which incorporates NVIDIA Kepler GPU with 192 CUDA cores and 4-Plus-1 quad-core ARM Cortex A15 CPUs [92]. Also, there are parallelizing tools and approaches that produce output code in CUDA, considering HPC systems as a target. Tools as Par4All [117] and PLuTo [18, 19] (no longer maintained) parallelize C code into CUDA parallel versions. [15] presents an automatic code transformation system that generates parallel CUDA code from input sequential C code for regular affine programs; [38] presents a compiler framework for tiling and parallelizing loop nests with uniform dependencies into CUDA code, built upon the based of PLuTo tool.

### 5.1.5 OpenCL

*Open Computing Language* (OpenCL) is a standardized, cross-platform, parallel-computing API based on the C99 language and designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. It is similar to CUDA, however it can be somewhat more complex as it strives for platform independence and portability [39], program written with OpenCL can be executed across heterogeneous platforms: CPU, GPU, DSP, and FPGA [56]. Although OpenCL is based on C, it excludes support for some features, such as function pointers, recursion, variable length arrays, bit fields; while adding some features to support specific concepts such as work items, work groups, synchronization and new data types. In addition it has a powerful library of built-in functions, e.g. for image processing [130]. OpenCL provides parallel computing using task and data level parallelism.

The execution platform on which an OpenCL program is executed is modeled as a collection of OpenCL devices, managed by a host processor, responsible for dispatching work to the devices [130]. Internally an OpenCL device consists of one or several compute units, each one divided into one or more PEs that may have single-instruction multiple-data or single-program multiple-data characteristics. The basic execution model of OpenCL relies on the concepts of kernel and program. A kernel is the basic unit of executable code, similar to a function in another language which can be applied over a data set, for data parallel execution, or as one instance, to model a task. A program is a collection of kernels and supporting functions for managing kernels.

Execution of an OpenCL program is based on the concept of a command queues. The host will place commands into one of the command queues, which are then scheduled for execution on the available devices. There are three types of commands supported in OpenCL:

- Kernel execution: a kernel is scheduled for execution using an index space.
- Memory management: transferring data between memory objects and mapping memory objects between address spaces.
- Synchronization: constrain the execution order of commands.

OpenCL supports two styles of command execution:

- In order, where commands are executed serially, the next command cannot start before the previous one completes, and
- Out-of-Order, where commands are still issued in order for execution, but do not wait on each other, thus any synchronization requires the use of explicit mechanisms.

OpenCL is designed to offer a set of C APIs and abstract the hardware layer into something like a computing layer, where the target platform is a second matter, but what is more important is the OpenCL version supported by the hardware and the level of API that your machine can offer. For instance, in the OpenCL design if a CPU and GPU are working in the same way, they can be activated to compute the same thing on both

or only 1 of them. This may be beneficial for porting applications among different and heterogeneous PEs capable to handle OpenCL.

Current embedded systems presents PEs that supports OpenCL. ARM Mali GPU systems presents full support to be programmed using OpenCL, and they are used in modern MP-SoC chips as the ARM big.LITTLE platforms. In the other hand, High Level Synthesis (HLS) tools had implemented OpenCL to develop applications for FPGA platforms. The Altera SDK for OpenCL allows the easy implementation of applications by abstracting away the complexities of FPGA design, allowing the development of hardware-accelerated kernel functions in OpenCL C, an ANSI C-based language with additional OpenCL constructs [8].

Few attempts exists to automate the process of generating OpenCL parallel applications from sequential inputs. Par4All claims to have limited support for C and FORTRAN parallelization into OpenCL parallel outputs [117]. The authors in [124] present a methodology for parallelization of sequential C programs with function calls to equivalent OpenCL programs with little assistance from programmer, identifying function calls and converting them into kernels to be executed in parallel on GPU devices. The GPSME toolkit aims to automatically convert C/C++ into OpenCL programs for GPU [138, 102]. [54] presents a compiler based approach to automatically generate optimized OpenCL code from data parallel OpenMP programs for GPUs. This approach brings together the benefits of a clear high level language, like OpenMP, and an emerging standard OpenCL for heterogeneous multi-cores systems. The OpenMP Accelerator model host runtime implementations uses the TI OpenCL Runtime as a back-end [67], so that OpenCL is, in a way or another, used in TI's tools and multi-core platforms.

### 5.1.6 Cilk

Cilk is a multi-threaded language derived from ANSI C [17, 95]. Cilk is an algorithmic language, which is an important distinction to make relative to other multi-threaded languages, as this means that the Cilk runtime guarantees both efficient and predictable performance. Cilk is based on both a language definition and a capable runtime system. The purpose of the runtime system is to remove the responsibility of thread scheduling, load balancing, and inter-thread communications from programmers, leaving them with the primary concern of identifying parallelism within their program [119]. Cilk Plus is the extension for both C and C++ programming languages.

The predictability of Cilk programs means that their analysis in a formal framework will yield predictions of performance that are close to those observed in practice. A programmer can analyze the performance properties of parallel programs based on two metrics: the work and span of the program. These metrics can be computed at runtime through automatic instrumentation inserted during compilation. This allows programmers to observe the actual parallel properties of their program under realistic workloads to understand how well the program parallelized and how it scales. Cilk achieves this through its work-

stealing scheduler in order to dynamically balance work across processors as the program executes.

Cilk Plus provides support for task and data parallelism. A user can perceived Cilk as a very easy to pick up language extension if already well versed in C, as the extensions Cilk makes to C are minimal and not disruptive to the base C language. Cilk Plus implements only three keywords to implement task parallelism [100]:

- *cilk\_for*: allows iterations of the loop body to be executed in parallel.
- *cilk\_spawn*: specifies that a function call can execute asynchronously, without requiring the caller to wait for it to return. This is an expression of an opportunity for parallelism, not a command that mandates parallelism. The Intel Cilk runtime will choose whether to run the function in parallel with its caller.
- *cilk\_sync*: specifies that all spawned calls in a function must complete before execution continues. There is an implied *cilk\_sync* at the end of every function that contains a *cilk\_spawn*.

Cilk allows automatic load balancing which provides good behavior in multi-programmed environments, and existing algorithms are easily adapted for parallelism with minimal modification. Cilk is available in commercial and open source licenses, which allow to have it in GCC and Clang/LLVM compilers and not just the Intel compilers.

Unfortunately, this extension is limited for x86 based architectures, of 32 and 64 bits, which reduces the applicability to a vast majority of embedded multi-core platforms. Nevertheless, Cilk presents a very interesting concept. By the time this document was written there are not tools nor approaches found to transform sequential C code into a Cilk parallel version.

### 5.1.7 Taxonomy

The Table 5.1 presents a comparative and summary table regarding the previous presented parallel programming models. The Table considers the type of parallelism exploited, if there is support for multi-core embedded systems, if there is parallelization tools or approaches that uses the library to produce output parallel codes. All except Cilk, present at least one approach to support it by embedded systems. Considering the tools available for OpenCL and CUDA are just a few, a couple, compared to the large amount using OpenMP for parallel outputs.

## 5.2 Tools

Many tools have been developed over the years to parallelize sequential code, with different success and limitations. Cover all the existing tool in this survey is a very ambitious task. Also, since most of the tools have been developed for HPC systems, their applicability



**Table 5.1:** Comparison chart for parallel programming models.

Language	Programming model	Parallelism	Embedded support	Tools
OpenMP	Shared-memory	DLP,TLP	Yes	Yes
Pthreads	Shared-memory	TLP	Yes	Yes
MPI	Distributed-memory	TLP	Yes	No
CUDA	Heterogeneity	DLP,TLP	Yes	Yes
OpenCL	Heterogeneity	DPL	Yes	Yes
Cilk	Shared-memory	DLP,TLP	No	No

to multi-core embedded systems is limited. In this section some tools are presented and described. They were selected, in part, for their potential usability for multi-core embedded systems.

### 5.2.1 ParallWare

ParallWare is a novel commercial source-to-source parallelizing compiler, designed for HPC systems and for sequential simulation programs. ParallWare automatically discovers the parallelism available in the input sequential code and automatically generates parallel equivalent source code annotated with OpenMP compiler directives [11], which preserves the maintainability and human readability of the code.

One field of application of this tool is in computational electromagnetic (CEM) simulations, important to the design and modeling of antenna, radar, satellite and other communication systems, nanophotonic devices and high-speed silicon electronics, medical imaging and cell-phone antenna design, among other applications. ParallWare has been able to provide a speedup of 4.85 on a quad-core platform.

This tool is intended to keep developing applications in a sequential manner, and run it in parallel. For this, the user manual presents programming guidelines, so the work of detecting parallelism is facilitated for the tool. Actually, the tool presents several unsupported features and provides the re-factoring suggestions. This implies that for an existing application it is required the modifications to make it compliant for the tool, otherwise parallelism would no be detected.

Details of the tool internals are not available due its commercial condition. The tool is available as a limited free-trial from the company web page<sup>4</sup>. To access to a full license the cost rises up to 20 000 €.

### 5.2.2 PaxES

PaxES stands for Parallelism Extraction for Embedded Systems [31], a parallelization

<sup>4</sup>[www.appentra.com](http://www.appentra.com)

framework optimized for embedded multi-core systems. This is one tool, with three different approaches, developed at TU Dortmund, that includes the works presented in [32, 29, 30]. These first works were developed for homogeneous systems, but extended to heterogeneous systems, as presented in [33]

This tool is among the few design for multi-core embedded systems and considering different objectives, i.e., rather than increase just the speedup of the applications, the tool consider energy and communication costs. This is needed for embedded systems due the constraints imposed by the applications.

The approaches in this tool are [31]:

- [32] presents an approach to extract very coarse-grained TkLP, e.g. two parallel function calls. An ILP-based algorithm is used for the parallelism extraction and HTG is used as an intermediate representation.
- [29] presents an approach to extract more fine-grained PLP from loops and loop nests from sequential C applications. Due many embedded applications have a streaming-oriented structure, is a strong motivation for extracting this type of parallelism. In this approach the loops are divided into concurrently executed task, horizontal splits are done to move statements to tasks and vertical splits to split iterations of tasks. PDG is used in this case for intermediate representation.
- [30] presents another approach for multi-objective aware extraction of TkLP using an algorithm based on GA. A multi-objective approach is a embedded systems motivation, due this systems must be optimized for multiple objectives. This approach observes three objectives while parallelizing an application: execution time, energy consumption, and communication overhead. For instance, a multi-objective compromise could be reduce the amount of extracted parallelism to save energy. This approach bring as output a Pareto-front with the optimal solutions, so the best implementation can be choose for a specific scenario. An augmented HTG for intermediate representation in this approach.

The previous works were expanded adapting PLP extraction to a multi-objective approach, as presented in [28]. The Figure 5.2 depicts the general concept of this parallelization tool. This Figure shows the extraction of a HTG and the application of a GA-based algorithms to determine the parallelism; but from the presented recently, this could be the PDG extraction and the application of ILP-based algorithm.

The dependency analyzer in this tool takes the IR of the optimized source code as input to extract all data dependencies of the application. This information is required to build the PDG or the HTG. Note that information regarding execution time and energy estimation is used to create this graphs. In this case, a profiling based approach is used. Due to profiling driven analysis techniques, parallelization hints might ignore dependencies which are not manifest in the profiling run.

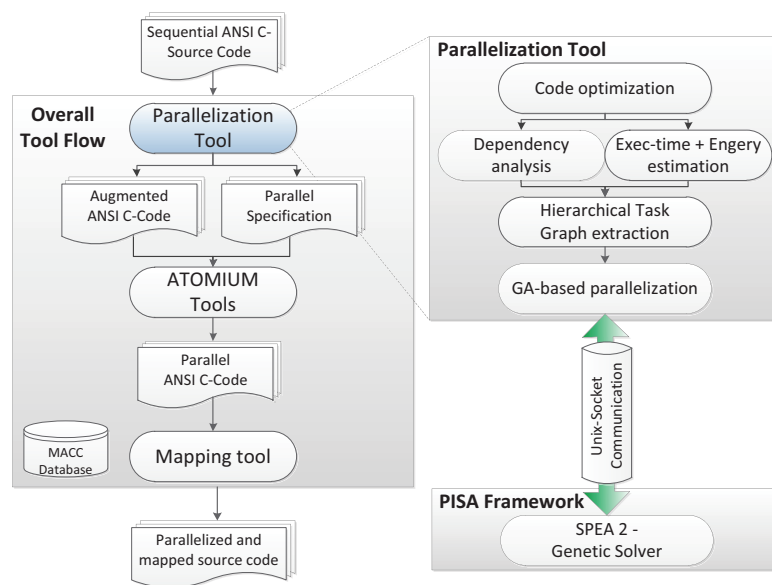


Figure 5.2: PaxES parallelization tool (Figure taken from [30]).

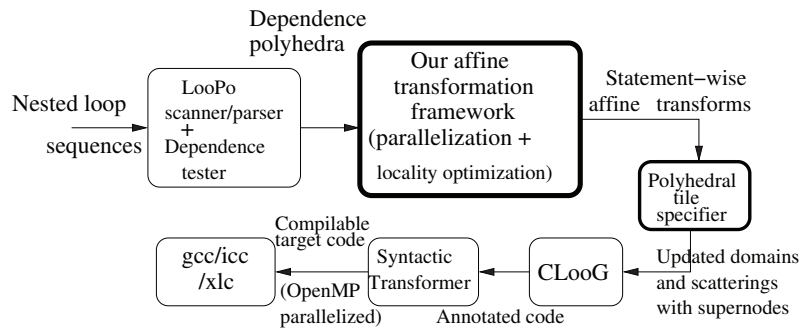
In all the approaches, the augmented C code, produce by the parallelization tool and a parallel specification are passed to the ATOMIUM tool suite, which implements the extracted parallelism. MPA tool is used to implement the extracted parallelism; this MPA requires a special runtime library, called RTLIB, which had to be ported to the different platforms. At the end of the process, a mapping tool is used to map the different tasks to the target platform.

### 5.2.3 PLuTo

PLuTo is a source-to-source transformation system based on the polyhedral model [18, 19, 20, 1]. The polyhedral model for compiler optimization provides an abstraction to perform high level transformations such as loop-nest optimization and parallelization on affine loop nests. This model is a geometrical representation for programs that utilizes machinery from Linear Algebra and Linear Programming. PLuTo transforms C programs from source to source for coarse-grained parallelism and locality simultaneously. The core transformation mainly works by finding affine transformations for efficient tiling and fusion.

PLuTo automatic transform C programs to OpenMP versions. Nevertheless, it requires the user to specify the regions that want to parallelize. This is done using `pragma scop` and `pragma endscop` around the target section code. The knowledge of the application that the user have is important to determine which code sections would provide a better profit if parallelized.

Several options are provided to tune aspects like tile sizes, unroll factors, and outer loop fusion structure. Outer, inner, or pipelined parallelization is achieved, using OpenMP programs, besides register tiling and making code sensitive to auto-vectorization. A CUDA



**Figure 5.3:** PLuTo source-to-source transformation system (Figure taken from [19]).

version of the tool was developed, but currently unsupported. The Figure 5.3 shows a flow diagram of this tool.

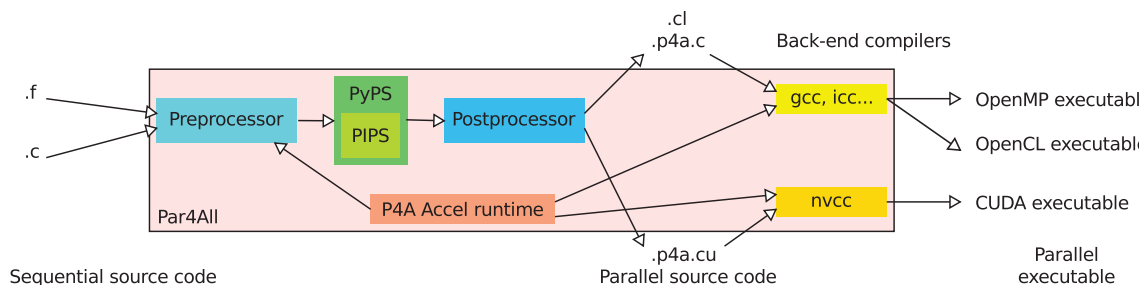
Due many applications often spend most of their execution time in nested loops, the polyhedral model provides a powerful abstraction to transform such loop nests by viewing a dynamic instance, or iteration, of each statement as an integer point in a well-defined space called the statement’s polyhedron. With such a representation for each statement, and a precise characterization of inter or intra-statement dependencies, it is possible to reason about the correctness of complex loop transformations in a completely mathematical setting relying on structures from linear algebra and linear programming [1]. The transformations reflect in the generated code, as reordered execution with improved cache locality and loops, that they have been parallelized. The polyhedral model is applicable to loop nests in which the data access functions and loop bounds are affine combinations of the enclosing loop variables and parameters.

PLuTo enables a model-driven or guided empirical search to be applied to arbitrary affine programs. Due the transformation system in PLuTo operates entirely in the polyhedral abstraction, it is not limited to C code, but could applied to any high level language from which polyhedral domains can be extracted and analyzed.

PLuTo depends of three libraries: PipLib, a parametric integer linear programming solver [99]; PolyLib, a library of polyhedral functions that operates on objects made up of unions of polyhedra of any dimension [85]; and CLooG, a free software library to generate code for scanning Z-polyhedra, and to solve the code generation problem for optimizing compilers based on the polytope model [27].

## 5.2.4 Par4All

Par4All is an automatic parallelizing and optimizing compiler for C and FORTRAN sequential programs [117]. It is a source-to-source compiler that aims to adapt existing applications to various hardware targets such as multi-core systems, HPC systems, GPUs, and some parallel embedded heterogeneous systems. The main goal of this Par4All is to be naturally independent of the target architecture details and to take advantage of the best back-end tools, such as highly optimized vendor compilers for a given proces-



**Figure 5.4:** Par4All tool flow (Figure taken from [10]).

processor or platform, including for embedded systems, open-source compilers and tools, and high-level hardware synthesizers [10].

Par4All aims at easy code generation for parallel architectures from sequential source codes written with almost no manual code modification required. Par4All is based on multiple components, mainly on the PIPS, a source-to-source compiler infrastructure, and it benefits from its interprocedural capabilities like memory effects, reduction detection, parallelism detection, but also polyhedral-based analyses such as convex array regions and preconditions. The Figure 5.4 presents the internal flow of this tool. Par4All allows the integration of third-party tools into the compilation flow, like in this case PIPS. The PIPS internal representation is a hierarchical control flow graph (HCFG) with a compact representation and a good trade-off between simplicity and source representation precision.

The current version of Par4All can generate CUDA and OpenCL code from C code and OpenMP from C and Fortran 77 code with a simple easy-to-use high-level script called p4a. This outputs allow that original source codes of existing applications to remain mainly unchanged for well formed programs. The future version of this tool is based on Clang/LLVM. Some architectural aspects can be expressed or generated in special source constructs to capture architectural details when needed (SIMD or accelerators intrinsics). The source-to-source aspect makes Par4All *de facto* interoperable with other tools as front-end or back-end compilers to build complete tool chains.

### 5.2.5 AESOP

AESOP is an automatic parallelizer is a tool that converts serial code from C, C++ and FORTRAN programs, mainly loops structures, to a parallel code and targets shared-memory machines [2]. This is an open source project<sup>5</sup> developed at the University of Maryland, College Park. AESOP leverages the LLVM infrastructure and presently works with LLVM-3.0. It targets parallelism for dense array-based codes with affine-based analysis using traditional methods.

AESOP is an affine automatic parallelizer. The Figure 5.5 shows the diagram for this tool.

<sup>5</sup>Available in [aesop.ece.umd.edu](http://aesop.ece.umd.edu).

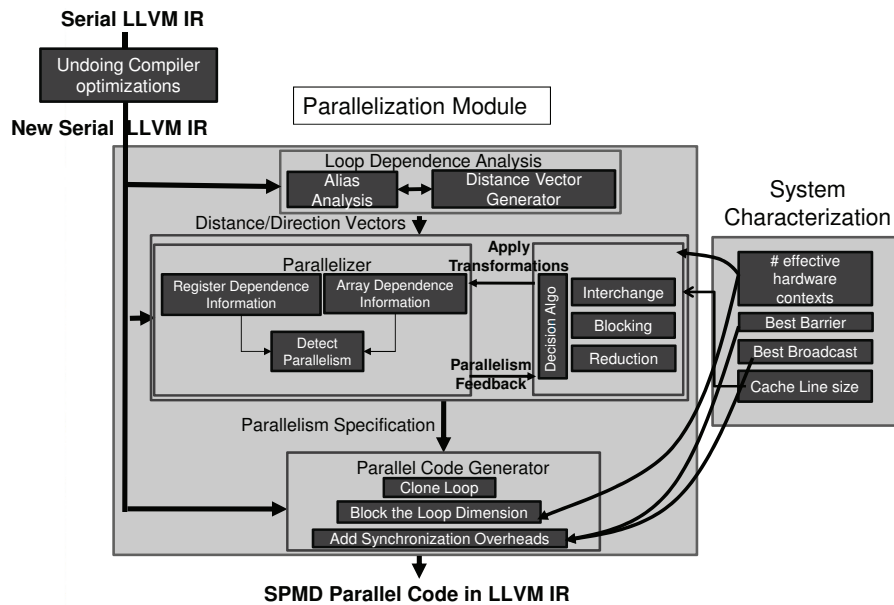
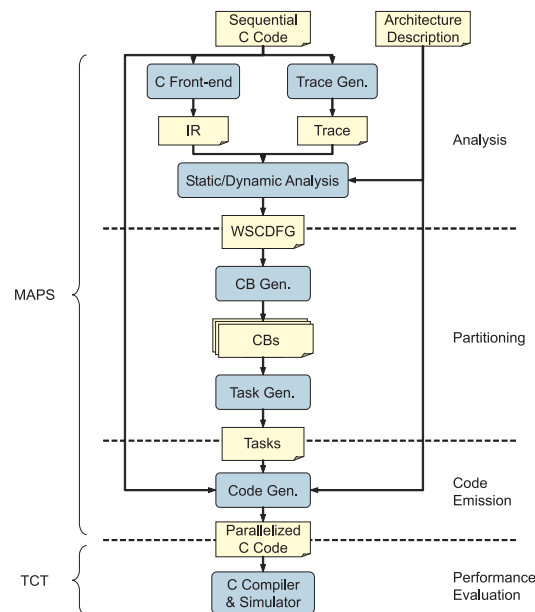


Figure 5.5: AESOP Parallelizer (Figure taken from [2]).

As can be seen from the flow, first the serial IR produced by LLVM is fed into the undoing compiler optimizations module, which includes the Loop Simplify and Induction Variable Simplify passes from LLVM. Doing this, the LLVM IR obtained after this remains serial, but the loops are simplified, i.e. contain only one exit block when possible and canonical induction variables are introduced into loops whenever possible. Such simplification of loops is essential for this tool to execute the affine analysis on such loops and generate parallel code [2]. After this first block, new serial LLVM IR is passed into the loop dependence analysis block, which includes an alias analysis module and a distance vector generator. Every pair of memory accesses in a loop are passed into the alias analysis module and the distance vector generator. The alias analysis passes are the standard ones present in LLVM.

The distance/direction vectors and the new serial LLVM IR are then passed into the parallelizer block which communicates with the decision algorithm block. The decision algorithm decides which loop dimensions to parallelize. Then, the parallel code generator block generates SPMD, Single-Program Multiple-Data, parallel code for each of parallelizable loops. This is done using POSIX-compliant Pthreads calls. AESOP is one of the few tools that produce Pthreads parallel version of the sequential input. This could be useful for multi-core embedded systems due many operative systems support this output, like Linux.

In the code generated, only the main thread executes serial code between parallel loops. The parallel threads produce by AESOP only execute loop code. When a parallel thread finishes one loop it waits for the main thread to inform it which loop to execute next in a broadcast. The broadcast also contains the values of registers calculated by the main thread that are needed by the parallel loop threads. A barrier is inserted into the binary at the end of every loop.



**Figure 5.6:** MAPS sequential partitioning (Figure taken from [24]).

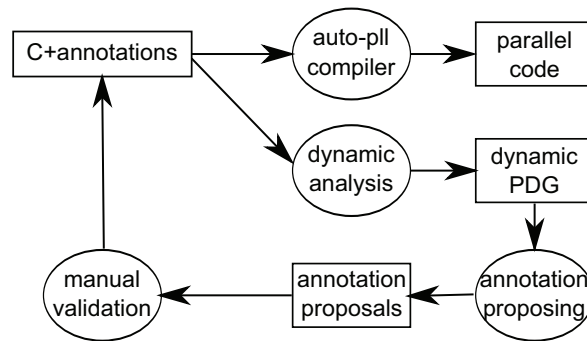
## 5.2.6 MAPS

The MPSoC Application Programming Studio, MAPS, [24, 82] developed by the ICE at RWTH-Aachen. This tool performs a semiautomatic parallelization technique in which the user can manually steer the granularity of the extracted parallelism. In this parallelization process, TkLP, DLP and PLP can be exploited, which present better benefits on multi-core embedded systems. The Figure 5.6 shows the sequential partitioning stage of MAPS. As shown, the parallelization process in MAPS can be divided into three phases: analysis, partitioning and code emission.

Two inputs are expected by MAPS, the sequential C code and the description of the target MPSoC platform. In the analysis phase, both static and dynamic approaches are applied in order to provide profiling information to the programmer, and data and control flow information to the MAPS partitioning tool which search for parallel tasks in the target application. MAPS combines information from static and dynamic profiling code analysis, in order to extract a Weighted Statement Control Data Flow Graph (WSCDFG) annotated with cost information.

The partitioning phase of MAPS was first approach by a heuristic clustering algorithm was applied to the WSCDFG to group statements subsequently to coarse-grained tasks. The partitioning phase of MAPS now uses two algorithms: SCC improvement and load balancing.

In the code emission phase, the resulting tasks are translate to parallelized C code. MAPS uses the Tightly-Coupled-Thread framework (TCT) as a backend for implementation and simulation of the extracted parallelism, in order to evaluate the parallel performance. The output of the tool are C source files with parallel constructs.



**Figure 5.7:** Parallax tool flow (Figure taken from [131]).

### 5.2.7 Parallax

Parallax compiler is a parallelizing compiler that is constructed specifically for parallelizing irregular pointer-intensive applications, developed in the Ghent University [132, 131]. Parallax is built on top of the LLVM compiler framework and coarse-grain loops operating on whole data structures. The Figure 5.7 shows the Parallax tool flow.

The compilation process starts with data structure analysis (DSA), a unification-based shape analysis that recovers the identity and type, e.g. array or structure, of all memory objects by inspecting the instructions that access them. DSA also identifies when one object holds a pointer to another one and indicates when pointers to objects may escape or when objects are aliased to other objects.

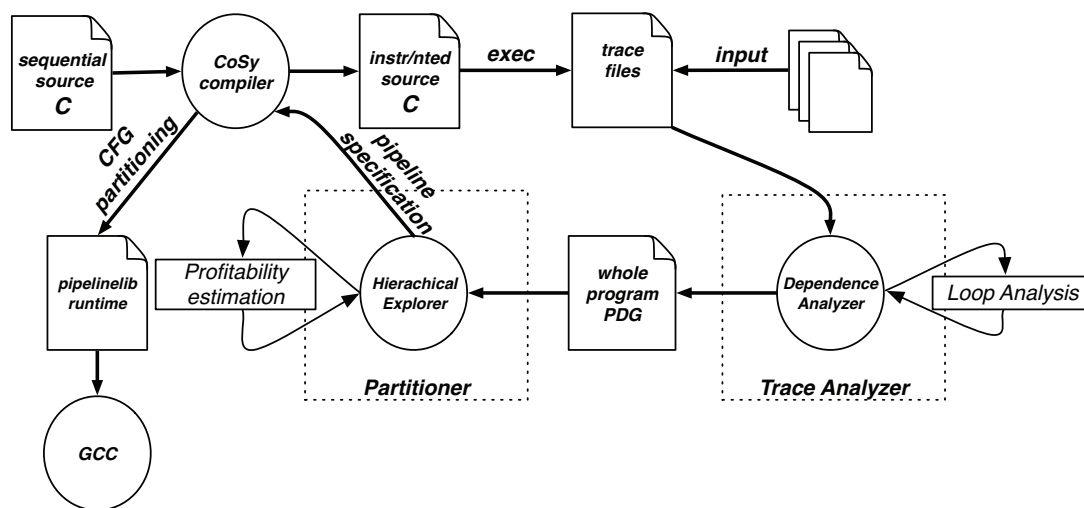
The parallelization follows by using DSWP. It is performed by analyzing the PDG. The PDG captures all dependencies, data, control and memory, in a program. Due the PDG may contain cycles, these cycles must always be contained within a single thread to minimize inter-thread communication. SCCs are computed from PDG and build the directed acyclic graph of strongly connected components (DAG-SCC). Using the DAG-SCC, various parallel loop patterns can be recognized such as pipelines, parallel-stage pipelines and DOALL loops, as well as a parallel sections in non-loop code.

The code is split across multiple threads using the multi-threaded code generation algorithm of DSWP. This algorithm determines what basic blocks must be replicated in each thread, which control transfers must be replicated and when values must be sent and received. For this, it is required the existence of queues, FIFOs, between the threads and two primitives: produce and consume, to perform and consume values from a queue.

Parallax assumes a lumped communication model where all values produced in a pipeline stage are sent at once at the end of the stage and they are all received at once at the start of the stage. This communication is effected by defining for each parallelized loop a communication structure. This is a structure in the C language that contains all values that are communicated between pipeline stages

In order to aid the Parallax compiler in finding significant TLP, we present a light-weight programming model (LWPM) to fill in the semantic gaps. The LWPM adds annotations to





**Figure 5.8:** Tournavitis parallelization work-flow (Figure taken from [127]).

a program that describe well-defined properties of functions, variables and data structures; information that a static compiler can not infer. The annotations are designed such that verification of their correctness is fairly easy. Paralax provides a dynamic analysis tool that proposes program locations to the user where annotations could improve the parallelization process.

## 5.2.8 Tournavitis

The Tournavitis [128, 126, 127, 125] approach integrates profile-driven parallelism detection and machine learning based mapping in a single framework, to exploit and extract pipeline parallelism in multimedia applications. The profiling data brings the actual control and data dependencies and enhance the corresponding static analyses with dynamic information. for this purpose the CoSy compiler is used to run instrumented IR code and obtain the traces for the dependence analysis. A trained machine learning based prediction mechanism is used to each parallel loop candidate to decide if the parallel mapping should be performed and how to do it, as summarized in the subsection 4.2.1.

This approach generates parallel code using standard OpenMP annotations, due to the low complexity of generating the required code annotations and the widespread availability of native OpenMP compilers. This approach is semiautomatic because expect the user to finally approval regarding the loops where parallelization is likely to be beneficial. DLP and PLP are exploited within this tool. The dependence analysis uses CDFG and CFG to represent the code. The Figure 5.8 depicts the work-flow for this approach.

### 5.2.9 Thies

[122] presents a tool presented in assists the programmer in extracting pipeline parallelism by a semiautomatic profiling-based technique. This tool aims to leveraging coarse-grained pipeline parallelism in C programs, present in streaming applications, such as audio, video, and digital signal processing, which exhibit regular flows of data. To exploit PLP, this tool provides to the user with a simple set of annotations, indicating pipeline boundaries, and a dynamic analysis. This tool depends and relies in the user knowledge of the application to properly parallelize it, but brings interaction to help y the process of partitioning the code.

The first step in this tool is to identify the main loop in the application, which is typically iterating over frames, packets, or another long-running data source. The user annotates the start and end of this loop, as well as the boundaries between the desired pipeline-parallel partitions. The tool reports the percentage of execution time spent in each pipeline stage, to bring guidance in the placement of pipeline boundaries.

The tool presents some restrictions on the placement of the partition boundaries. All boundaries must appear within the loop body itself, rather than within a nested loop, within nested control flow, or as part of another function. The user may work around these restrictions by performing loop distribution or function in lining. *for* and *while* loops are supported, but can not be any break or continue statements within the loop; such statements implicitly alter the control flow in all of the partitions, an effect that is difficult to trace by the dynamic analysis.

Once a loop has been annotated with partition boundaries, the user selects a set of training inputs and runs our dynamic analysis to trace the communication pattern. The tool outputs a stream graph, which is a list of producer and consumer statements, and a set of communication macros for automatically running the code in parallel. If the user is not satisfied with the extracted speedup, the user has to redefine pipeline boundaries over several steps, which is tricky and not scalable. Communication and synchronization directives are finally inserted by the parallelization framework, based on the profiling information. The dynamic analysis tool used is built on top of Valgrind, which is a framework for dynamic binary instrumentation.

A standard inter-process communication mechanism, like pipes, is used to send and buffer data from one process to another. Thus a producer sends its latest value for a given location, and the consumer reads that value into the same location in its private address space. At the end of the execution of one loop, all of the processes copy their modified data, as recorded by the tool during the profiling stage, into a single process that continues after the loop.

## 5.3 Taxonomy

In this section, a comparison table is presented to summarize several of the most important features that allow the characterization of tools for parallelizing sequential codes. The Table 5.2 and 5.3 includes information regarding:

- The **input** language, sequential program, and the corresponding output language, parallel program. Most of the tools receive as input C, while others also handle FORTRAN and C++.
- The type (or types) of **parallelism** exploited by the tool.
- The type of graph used as a **representation** model to abstract the input program.
- The type of code **analysis**, i.e., static, dynamic or hybrid.
- The **target** system of the tool. Normally this is either high performance computing (HPC) or embedded systems (ES), and in rare cases for both.
- The **method**, refers to the algorithm or technique used, if known, to extract parallelism.
- **Automation**, this refers if the tool produces the output automatically, without any user intervention, or semiautomatically, with input from the user in the parallelization process.
- The **objective** (or objectives) for the parallelization process.
- The **type** of tool, commercial or academic.
- **Source code**. Due to the different license conditions of the tools, the source code is open or closed; in some cases source code or executable is not available (n.a.).
- **Architecture**, this refers if the target platform for a tool is homogeneous or heterogeneous, in rare cases supports both.

In some cases, some tool characteristics are unknown and therefore some cells are empty. As can be noticed from the Tables, most of the tools have been developed considering HPC as target where the parallelized code will be executed. For this reason, the main objective enhanced with parallelism extraction, over the time, has been the execution time. Due to PaxES implementation has been developed for multi-core embedded systems, has considered multiple objectives, as presented before. Even though almost all tools are academic, in few cases the source code of the tools is available for modifications and experimentation.

**Table 5.2:** Taxonomy of tools for parallelism extraction.

Framework	Input	Output	Parallelism	Representation	Analysis	Target
Parallware	C	OpenMP	DLP,TLP	—	Static	HPC
PaxES	C	MPA	TkLP,DLP,PLP	(A)HTG,(A)PDG	Dynamic	ES
PLUTO	C	OpenMP,CUDA	DLP	—	Static	HPC
Par4All	C,FORTRAN	OpenMP,CUDA,OpenCL	DLP	HCFG	Static	HPC/ES
AESOP	C,C++,FORTRAN	Pthreads	DLP	CFG	Static	HPC
MAPS	C	CPN	DLP,TLP,PLP	WSCDFG	Hybrid	ES
Paralax	C	C	PLP	PDG	Hybrid	HPC
Tournavitis	C	OpenMP	DLP,PLP	PDG,CFG	Dynamic	HPC
Thies	C	C	PLP	—	Dynamic	HPC

**Table 5.3:** Taxonomy of tools for parallelism extraction. *Continuation*

Framework	Method	Automation	Objectives	Type	Code	Architecture
Parallware	—	Automatic	Speedup	Commercial	Close	Homogeneous
PaxES	ILP, Genetic	Semiautomatic	Multiple	Academic	n.a.	Homo/Hetero
PLuTo	Polyhedral	Semiautomatic	Speedup	Academic	Open	Homogeneous
Par4All	Polyhedral	Automatic	Speedup	Academic	Open	Homogeneous
AESOP	Affine	Automatic	Speedup	Academic	Open	Homogeneous
MAPS	Clustering	Semiautomatic	Speedup	Academic	Close	Homo/Hetero
Paralax	SCC	Semiautomatic	Speedup	Academic	n.a.	Homogeneous
Tournavitis	Machine learning	Semiautomatic	Speedup	Academic	n.a.	Homogeneous
Thies	Manual annotation	Semiautomatic	Speedup	Academic	n.a.	Homogeneous



# Chapter 6

## Conclusions and future work

There is huge amount of legacy code developed in a sequential fashion over the years, particularly in C language, that have been used by single-core embedded systems. Nowadays, powerful multi-core embedded systems are available. To harness this new computational capabilities, it is required not only to write programs in a parallel manner, which are inherently more difficult to write than sequential ones, but to transform existing code to take advantage of these systems.

This work has presented a survey through different aspects considered over the time to detect and extract parallelism form sequential programs, and that could present any degree of applicability in modern multi-core embedded systems. This work does not aim to cover everything that has been done in more than three decades of work from the industry and mainly from the academia, to address the development of parallelizing compilers to expand the use of sequential code in multi-core platforms. In this chapter, conclusions and possible futures lines of work are presented.

### 6.1 Conclusions

From the information presented in this work, it is notable that there is no a definitive approach for parallelizing compilers, nothing as a holy grail nor a silver bullet that allows to take any sequential code and produce a parallel version capable to run in a wide spectrum of multi-core systems. This remains as an open end challenge.

The following list presents important key aspects, derived from the information presented in this work, that should be considered when designing and developing a parallelizing compiler, automatic or semiautomatic, for multi-core embedded systems.

- *Multi-core era.* The multi-core era is here to stay. Multi-core systems have shown a better compromise between performance and energy consumption than the previous design trends. Multi-core systems for HPC and embedded systems differ significantly. For instance, in embedded systems constrained nature, the increase

in performance should be done considering energy, thermal, communication, and time constraints, and how the number of multiple cores, running simultaneously, may affect these constraints. Multi-core era also present challenges in the way these systems are programmed, and therefore how sequential code can be parallelized to run efficiently on them.

- *Input information.* Many existing tools are target agnostic, i.e., do not consider information of the target platform where the parallelize code will run, or even information from the application itself. Usually the only input information available, besides the sequential input code, is the number of desired threads in the output code. Input information about target platform capabilities and desired performance metrics is missing in most frameworks. Distinct programs and platforms have different characteristics, that if known in the parallelization process, would provide useful information for a optimal parallel output. For example, image processing and computer vision applications can be optimally parallelized with a pipeline scheme, but if this criterion is unknown by the tool, the resulting parallelized code may not be the optimal. An ideal parallelizing compiler should be a resource-aware tool in order to obtain high utilization of the chip resources as well as computational and energy efficiency.
- *More than execution time.* Most of the tools and approaches developed have focused in optimizing the execution time, from the original sequential to the parallel version, mainly because of its applicability to HPC systems. Due few works, e.g. [24] and [33], have been developed considering embedded multi-core systems as a target platform where parallelized code will be executed, other important objectives, such as energy, communication and time, have not been considered when parallelizing programs. An ideal parallelizing compiler should consider not only the potential speedup with the parallelization, but be aware of the energy consumption, communication overhead, timing and thermal management that this implies. For instance, exploit all the available parallelism in an application may produce an undesired increase in energy consumption, once the parallel version of the application is executed in the multi-core platform.
- *Coarse-grained parallelism.* Coarser parallelism have shown better results for multi-core mapping, and representation models such as HTG and PDG have been used for this end. However, when generating parallel code for coarse-grained architectures, many delicate trade-offs must be managed. One of the most difficult is minimizing communication and synchronization overhead while balancing the load evenly across all the PEs. For this, input information, previously mentioned, plays an important role. Information regarding constraints of the parallel execution is needed to evaluate of objectives, as presented in [28].
- *Static and dynamic information.* An ideal parallelizing compiler should consider both static and dynamic dependencies information to better determine the manner an application can be parallelized. Currently, few approaches incorporate both



code analysis, while the static has been more used. For instance, runtime information from an application can help to detect hot spots, regions of code where the parallelization would bring a better profit, and avoid the risk of parallelizing code regions, with low execution rates, that would consume resources, communication and synchronization, but do not contribute significantly in performance gain.

- *Automation.* To have a totally automatic tool, without any user intervention, is a golden dream. Nevertheless, doing so, the expert judgment, which can provide useful information for the parallelization process, might be despised. Usually, the user better knows the application and can bring hints to the tool about where to focus or where not to, in order to enhance the parallelism discovery and extraction. The parallel code generation from a sequential input is a desired, totally automated process, this is because manual code adaptation and rewrite is a error-prone process.
- *Supported outputs.* Existing parallelizing tool are source-to-source compilers, which transforms sequential C into parallel version on OpenMP and CUDA, for instance. This is in part motivated by the tools available to program parallel machines. Due this programming models are reaching the embedded systems world, it is worthy to include this type of standard outputs in an ideal parallelizing compiler. As presented in previous chapters, and as an example, modern multi-core embedded systems support OpenMP, so a parallelizing tool can take advantage of the work done by chip manufacturers and tools vendors to include the support in their toolchains, like the OpenMP Accelerator Model of TI to support OpenMP in heterogeneous MPSoC. On the other hand, this software abstraction layer could result in a benefit due the parallel outputs might be used for a wider amount of multi-core platforms.

## 6.2 Future work

There is plenty of work to develop smarter parallelizing frameworks. Nevertheless, develop an entire tool for discovering and extracting parallelism from sequential programs is not an easy task. A couple of proposed lines of future work, using existing tools as a starting point, are:

- *Explore existing tools.* Due the increased OpenMP-enabled multi-core embedded systems, several existing tools, whose outputs are OpenMP C code can be tested on a set of platforms and for a set of typical embedded applications. For instance, execution time and energy consumption can be measured and evaluated to characterize each tool parallel outputs, and to explore the compromise of speedup vs energy for each tool. If a parallelizing tool is being designed and developed, this experimentation can provide useful information about techniques or characteristics present in other frameworks that can be worthy to replicate and implement in the own tool.

- *Expand existing tools.* If the source is available for an existing tool or approach, increase its capabilities by adding input information or more objectives, rather than only execution time, may be valuable to treat. For instance, a speedup focused framework can be expand by adding support to heterogeneous PEs, like the ones found in ARM big.LITTLE chips, so a program might be map either to high-performance or energy-efficient cores depending on time or energy requirements, information that the tool should receive as input for the parallelization.

# Bibliography

- [1] Pluto - an automatic parallelizer and locality optimizer for multicores. URL <http://www.ece.lsu.edu/jxr/pluto/>.
- [2] T. Creech A. Kotha and R. Barua. Aesop: The autoparallelizing compiler for shared memory computers. Technical report, Department of Electrical and Computer Engineering, University of Maryland, College Park, April 2013.
- [3] Jay Abraham. Using formal methods for sophisticated static code analysis, June 2012. URL <http://www.embedded.com/design/debug-and-optimization/4374801/1/Using-formal-methods-for-sophisticated-static-code-analysis>.
- [4] Adapteva. Epiphany-iv 64-core 28nm microprocessor (e64g401), 2014. URL <http://www.adapteva.com/epiphanyiv/>.
- [5] Adnan Agbaria, Dong-In Kang, and Karandeep Singh. Lmpi: Mpi for heterogeneous embedded distributed systems. In *ICPADS*, pages 79–86. IEEE Computer Society, 2006.
- [6] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.
- [7] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.
- [8] Altera. Altera sdk for opencl. URL <http://www.altera.com/products/software/opencl/opencl-index.html>.
- [9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM.
- [10] Mehdi Amini, Batrice Creusillet, Onil Goubier, Serge Guelton, Ronan Keryell, and Grgoire Pan. Par4all user guide, January 2014. URL [http://download.par4all.org/doc/par4all\\_user\\_guide/par4all\\_user\\_guide.pdf](http://download.par4all.org/doc/par4all_user_guide/par4all_user_guide.pdf).
- [11] Appentra. Parallware: Source-to-source parallelizing compiler, 2014. URL <http://www.appentra.com>.

- [12] ARM. big.little technology. URL <http://www.arm.com/products/processors/technologies/biglittleprocessing.php>.
- [13] Aditi Athavale, Priti Randive, and Abhishek Kambale. Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel codes. URL <http://www.kpit.com/downloads/research-papers/automatic-parallelization-sequential-codes.pdf>.
- [14] Blaise Barney. Posix threads programming, 2014. URL <https://computing.llnl.gov/tutorials/pthreads/>.
- [15] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC'10/ETAPS'10*, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.
- [16] Jason Benway. The 3 key differences between static and dynamic code analysis tools. URL <http://www.savagenomads.net/2013/01/02/the-3-key-differences-between-static-and-dynamic-code-analysis-tools/>.
- [17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95*, pages 207–216, New York, NY, USA, 1995. ACM.
- [18] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.
- [19] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.
- [20] Uday Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical report, Ohio State University, 2007.
- [21] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT '92*, pages 144–152, New York, NY, USA, 1992. ACM.
- [22] Pierre Boulet, Alain Darte, and Georges-André Silber. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing -*

- Special issues on languages and compilers for parallel computers archive*, 24:421 – 444, 1997.
- [23] Jerónimo Castrillón. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, RWTH Aachen University, Chair for Software for Systems on Silicon, apr 2013.
- [24] Jianjiang Ceng, Jeronimo Castrillon, Weihua Sheng, H. Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, T. Isshiki, and H. Kunieda. Maps: An integrated framework for mpsoC application parallelization. In *45th Design Automation Conference (DAC '08)*, pages 754–759, Anaheim, CA, USA, jun 2008. ACM.
- [25] Dehao Chen, Wenguang Chen, and Weimin Zheng. Cuda-zero: a framework for porting shared memory gpu applications to multi-gpus. *SCIENCE CHINA Information Sciences*, 55(3):663–676, 2012.
- [26] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA '90*, pages 239–248, New York, NY, USA, 1990. ACM.
- [27] CLooG. The cloog code generator in the polyhedral model's home, 2013. URL <http://www.cloog.org>.
- [28] Daniel Cordes, Michael Engel, and Peter Marwedel Olaf Neugebauer. Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms. *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '12*, pages 73–82, 2012.
- [29] Daniel Cordes, Andreas Heinig, Peter Marwedel, and Arindam Mallik. Automatic extraction of pipeline parallelism for embedded software using linear programming. *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 699–706, 2011.
- [30] Daniel Cordes and Peter Marwedel. Multi-objective aware extraction of task-level parallelism using genetic algorithms. *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.
- [31] Daniel Cordes and Peter Marwedel. Paxes arallelism extraction for embedded systems: Three approaches ne tool, mar 2012. Research Poster at The Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2012) (DATE Workshop).
- [32] Daniel Cordes, Peter Marwedel, and Arindam Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*, pages 267–276, 2010.

- [33] Daniel Alexander Cordes. *Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models*. PhD thesis, Fakultät für Informatik, Technischen Universität Dortmund, 2013.
- [34] CSAIL. Streamit. URL <http://groups.csail.mit.edu/cag/streamit/>.
- [35] Alain Darte and Frdric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *IEEE PACT*, pages 281–291. IEEE Computer Society, 1996.
- [36] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nein Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted mosfet’s with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.
- [37] Intel Developer Zone. Granularity and parallel performance, February 2012. URL <https://software.intel.com/en-us/articles/granularity-and-parallel-performance>.
- [38] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *ICPP*, pages 350–359. IEEE Computer Society, 2012.
- [39] Vassilios V. Dimakopoulos. *Smart Multicore Embedded Systems*, chapter Parallel Programming Models, pages 3 – 17. Springer, 2014.
- [40] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced structural analysis for c code reconstruction from ir code. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems, SCOPEs ’11*, pages 21–27, New York, NY, USA, 2011. ACM.
- [41] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA ’11*, pages 365–376, New York, NY, USA, 2011. ACM.
- [42] ACE Associated Compiler Experts. Cosy compiler development system.
- [43] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.
- [44] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.
- [45] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

- [46] Jose Fonseca. Gprof2dot, May 2014. URL <https://code.google.com/p/jrfonseca/wiki/Gprof2Dot>.
- [47] Geoffrey C. Fox, Roy D. Williams, and Giuseppe C. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.
- [48] Vladimir Getov, Adolfo Hoisie, and Harvey J. Wasserman. Codesign for systems and applications: Charting the path to exascale computing. *Computer*, 44(11):19–21, November 2011.
- [49] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):166–178, March 1992.
- [50] Milind Girkar and Constantine D. Polychronopoulos. Extracting task-level parallelism. *ACM Trans. Program. Lang. Syst.*, 17(4):600–634, July 1995.
- [51] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII*, pages 151–162, New York, NY, USA, 2006. ACM.
- [52] Jurgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10*, pages 105–114, Washington, DC, USA, 2010. IEEE Computer Society.
- [53] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004.
- [54] Dominik Grewe, Zheng Wang, and Michael F. P. O’Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO*, pages 1–10. IEEE Computer Society, 2013.
- [55] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1999.
- [56] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. URL <https://www.khronos.org/opencl/>.
- [57] Sameer Gupta and Dominik Luecke. Comparison of different data flow graph models, 2006. URL [http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/11\\_report.pdf](http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/11_report.pdf).

- [58] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Comput. Archit. Lett.*, 8(1):25–28, January 2009.
- [59] Mark Harris. Using shared memory in cuda c/c++, January 2013. URL <http://devblogs.nvidia.com/paralleforall/using-shared-memory-cuda-cc/>.
- [60] Jiangzhou He, Wenguang Chen, Guangri Chen, Weimin Zheng, Zhizhong Tang, and Handong Ye. Openmdsp: Extending openmp to program multi-core dsp. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 288–297. IEEE Computer Society, 2011.
- [61] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2012.
- [62] Mark D. Hill and Michael R. Marty. Amdahl’s law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.
- [63] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.
- [64] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’10, pages 121–130, New York, NY, USA, 2010. ACM.
- [65] Shih-Hao Hung, Po-Hsun Chiu, and Chi-Sheng Shih. Building a scalable and portable message-passing library for embedded multicore systems. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS ’11, pages 31–37, New York, NY, USA, 2011. ACM.
- [66] Texas Instrument. Openmp programming for keystone multicore processors. Online, 2012. URL <http://www.ti.com/lit/ml/sprt620a/sprt620a.pdf>.
- [67] Texas Instruments. Openmp accelerator model 0.3.3 - user’s guide, April 2014. URL [http://processors.wiki.ti.com/index.php/OpenMP\\_Accelerator\\_Model\\_0.3.3](http://processors.wiki.ti.com/index.php/OpenMP_Accelerator_Model_0.3.3).
- [68] Intel. *Intel Hyper-Threading Technology*. 2003. URL [http://cache-www.intel.com/cd/00/00/01/77/17705\\_htt\\_user\\_guide.pdf](http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf).
- [69] Intel. Intel xeon phi coprocessor 7120a, 2014. URL [http://ark.intel.com/de/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1\\_238-GHz-61-core](http://ark.intel.com/de/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1_238-GHz-61-core).



- [70] Ahmed Amine Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*, chapter The What, Why, and How of MPSoCs, pages 1–18. Systems on Silicon. Morgan Kaufman, 2004.
- [71] YoungHoon Jung, Jinhyung Park, Michele Petracca, and Luca P. Carloni. netship: A networked virtual platform for large-scale heterogeneous distributed embedded systems. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 169:1–169:10, New York, NY, USA, 2013. ACM.
- [72] David Kaeli and David Akodes. The convergence of hpc and embedded systems in our heterogeneous computing future. In *Proceedings of the 2011 IEEE 29th International Conference on Computer Design, ICCD '11*, pages 9–11, Washington, DC, USA, 2011. IEEE Computer Society.
- [73] Lina J. Karam, Ismail AlKamal, Gene A. Frantz, David V. Anderson, and Brian L. Evans. Trends in multicore dsp platforms. *IEEE Signal Processing Magazine*, 26:38–49, November 2009.
- [74] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [75] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.
- [76] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
- [77] David Kleidermacher and Mike Kleidermacher. *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Newnes - Elsevier, 2012.
- [78] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '81*, pages 207–218, New York, NY, USA, 1981. ACM.
- [79] David L. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., New York, NY, USA, 1978.
- [80] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2), February 1974.
- [81] Corinna G. Lee. Utdsp benchmark suite, 1997. URL <http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html>.

- [82] Rainer Leupers and Jeronimo Castrillon. Mpsoc programming using the maps compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 897–902, Piscataway, NJ, USA, 2010. IEEE Press.
- [83] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.
- [84] Jinfeng Liu. Mpi for embedded systems: A case study. Technical report, Department of Electrical & Computer Engineering, University of California, Irvine.
- [85] Vincent Loechner. Polylib - a library of polyhedral functions, 2010. URL <http://icps.u-strasbg.fr/polylib/>.
- [86] Silvia Lovergine. *Harnessing Adaptivity Analysis for the Automatic Design of Efficient Embedded and HPC Systems*. PhD thesis, Dipartimento di Elettronica, Informazione e Bioingegneria. Politecnico di Milano, 2013.
- [87] Silvia Lovergine and Fabrizio Ferrandi. Harnessing adaptivity analysis for the automatic design of efficient embedded and hpc systems. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 2298–2301, Washington, DC, USA, 2013. IEEE Computer Society.
- [88] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.
- [89] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.
- [90] Luís Moura and Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2, chapter Parallel Programming Models and Paradigms, pages 4–27. Prentice Hall, 1999.
- [91] Nvidia. Cuda. URL [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html).
- [92] Nvidia. Nvidia jetson tk1. URL <http://www.nvidia.de/object/jetson-tk1-embedded-dev-kit-de.html>.
- [93] OpenMP. The openmp api specification for parallel programming. URL <http://openmp.org/wp/>.
- [94] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

- [95] Hans Pabst. Intel system studio - multicore programming with intel cilk plus, October 2014. URL <https://software.intel.com/en-us/articles/signal-processing-with-intel-cilk-plus>.
- [96] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.
- [97] Santiago Pagani, Heba Khdr, Waqaas Munawar, Jian-Jia Chen, Muhammad Shafique, Minming Li, and Jörg Henkel. Tsp: Thermal safe power: Efficient power budgeting for many-core systems in dark silicon. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 10:1–10:10, New York, NY, USA, 2014. ACM.
- [98] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 67–78, New York, NY, USA, 1991. ACM.
- [99] PipLib. The parametric integer programming's home. URL <http://www.piplib.org>.
- [100] Clik Plus. Intel clik plus. URL <http://www.cilkplus.org>.
- [101] Constantine D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 252–263, New York, NY, USA, 1991. ACM.
- [102] GPSME Project. A general toolkit for utilisation of sme applications (gpsme). URL <http://www.gp-sme.eu>.
- [103] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M.K. Martin. Computational sprinting on a hardware/software testbed. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 155–166, New York, NY, USA, 2013. ACM.
- [104] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. Computational sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [105] Priti Ranadive and Vinay G. Vaidya. Parallelization tool. URL <http://www.kpit.com/downloads/research-papers/parallelization-tools.pdf>.
- [106] Margaret Rouse. Multi-core processor, March 2007. URL <http://searchdatacenter.techtarget.com/definition/multi-core-processor>.

- [107] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism driven by data dependencies. *SIGARCH Comput. Archit. News*, 35(1):55–62, March 2007.
- [108] Silvius Rus and Lawrence Rauchwerger. Hybrid dependence analysis for automatic parallelization. Technical report, 2005.
- [109] S. Saha, S. S. Bhattacharyya, and W. Wolf. A communication interface for multi-processor signal processing systems. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, ESTMED '06*, pages 127–132, Washington, DC, USA, 2006. IEEE Computer Society.
- [110] Sankalita Saha, Jason Schlessman, Sebastian Puthenpurayil, Shuvra S. Bhattacharyya, and Wayne Wolf. An optimized message passing framework for parallel implementation of signal processing applications. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '08*, pages 1220–1225, New York, NY, USA, 2008. ACM.
- [111] Manuel Saldaa and Paul Chow. Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. pages 1–6. IEEE, 2006.
- [112] Manuel Saldaa, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. Mpi as a programming model for high-performance reconfigurable computers. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):22:1–22:29, November 2010.
- [113] Vivek Sarkar. *Languages and Compilers for Parallel Computing*, chapter A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs, pages 16 –30. Springer-Verlag, 5th edition, 1992.
- [114] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008.
- [115] Kiruthika Selvamani and Tarek M. Taha. Estimating critical region parallelism to guide platform retargeting. In *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 1*, ACM-SE 43, pages 168–173, New York, NY, USA, 2005. ACM.
- [116] Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran, and Jörg Henkel. Dark silicon as a challenge for hardware/software co-design: Invited special session paper. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES '14*, pages 13:1–13:10, New York, NY, USA, 2014. ACM.

- [117] SILKAN. Par4all, 2011. URL <http://www.par4all.org>.
- [118] Pushpendra Singh, Devesh Chaurasiya, Ankita Joshi, and Kumar Sambhav Pandey. A study of data flow graph representation analysis with syntax and semantics. In *International Journal of Advanced Research in Computer Science and Software Engineering*, volume 2, February 2012.
- [119] Matthew Sottile, Timothy G. Mattson, and Craig E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC, 1st edition, 2009.
- [120] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P. Rendell, and Ian Lintault. *OpenMP in the Era of Low Power Devices and Accelerators*, chapter OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip, pages 114 – 127. Springer, 2013.
- [121] Michael Suess. An interview with dr. jay hoeflinger about automatic parallelization, August 2007. URL <http://www.thinkingparallel.com/2007/08/14/an-interview-with-dr-jay-hoeflinger-about-automatic-parallelization/>.
- [122] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.
- [123] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction, CC '02*, pages 179–196, London, UK, UK, 2002. Springer-Verlag.
- [124] Krishnahari Thouti and S. R. Sathe. Article: A methodology for translating c-programs to opencl. *International Journal of Computer Applications*, 82(3):11–15, November 2013. Full text available.
- [125] Georgios Tournavitis. *Profile-driven Parallelisation of Sequential Programs*. PhD thesis, Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh, 2011.
- [126] Georgios Tournavitis and Björn Franke. Towards automatic profile-driven parallelization of embedded multimedia applications. *MULTIPROG-2009: Proceedings of the Second Workshop on Programmability Issues for Multi-Core Computers*, pages 53–64, 2009.
- [127] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10*, pages 377–388, New York, NY, USA, 2010. ACM.

- [128] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O’Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’09, pages 177–187, New York, NY, USA, 2009. ACM.
- [129] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT ’07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.
- [130] András Vajda. *Programming Many-Core Chips*, chapter Multi-core and Many-core Processor Architectures, pages 9–43. Springer Publishing Company, Incorporated, 1st edition, 2011.
- [131] Hans Vandierendonck and Koen De Bosschere. Automatic parallelization in the paralax compiler. In *SCOPES*, 2011.
- [132] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *PACT*, 2010.
- [133] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.
- [134] Cheng Wang, Sunita Chandrasekaran, Peng Sun, Barbara Chapman, and Jim Holt. Portable mapping of openmp to multicore embedded systems using mca apis. *SIGPLAN Not.*, 48(5):153–162, June 2013.
- [135] Martin Wei and Howard Sholl. An expression model for extraction and evaluation of parallelism in control structures. *IEEE Transactions on Computers*, c-31(9):851–863, September 1982.
- [136] Maximilian Odendahl Rainer Leupers Weihua Sheng, Stefan Schrmans and Gerd Ascheid. Automatic calibration of streaming applications for software mapping exploration. In *Proceedings of the International Symposium on System-on-Chip (SoC)*, pages 136–142, nov 2011.
- [137] Wikipedia. Automatic parallelization tool, 2014. URL [http://en.wikipedia.org/wiki/Automatic\\_parallelization\\_tool](http://en.wikipedia.org/wiki/Automatic_parallelization_tool).
- [138] David Williams, Valeriu Codreanu, Po Yang, Baoquan Liu, Feng Dong, Burhan Yasar, Babak Mahdian, Alessandro Chiarini, Xia Zhao, and Jos BTM Roerdink. Evaluation of autoparallelization toolkits for commodity graphics hardware. *10th International Conference on Parallel Processing and Applied Mathematics, 2013*, 2013.

- [139] Linda Wills, Tarek Taha, Lewis Baumstark Jr, and Scott Wills. Estimating potential parallelism for platform retargeting. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 55–, Washington, DC, USA, 2002. IEEE Computer Society.
- [140] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991.
- [141] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
- [142] Wayne Wolf, Ahmed Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.
- [143] David C. Zaretsky, Gaurav Mittal, Robert Dick, and Prith Banerjee. *Languages and Compilers for Parallel Computing, 18th International Workshop*, chapter Generation of Control and Data Flow Graphs from Scheduled and Pipelined Assembly Code, pages 76 –90. Springer-Verlag, 2007.
- [144] Christian Zebelein, Tobias Schwarzer Christian Haubelt, Joachim Falk, and Jürgen Teich. Representing mapping and scheduling decisions within dataflow graphs. In ECSI European Electronic Chips and Systems design Initiative, editors, *2013 Forum on specification and Design Languages (FDL), Paris, France, September 24 - 26, 2013*, volume FDL, pages 184–191. ECSI - European Electronic Chips and Systems design Initiative, 2013.
- [145] Jianjun Zhao and Martin Rinard. System dependence graph construction for aspect-oriented programs, 2003.
- [146] Martin Zlomek. Video watermarking. Master's thesis, Faculty of Mathematics and Physics, Charles University of Prague, 2007.





# Appendix A

## Pi parallel codes

The code produce by the Parallware tool is presented in Listing A.1, while the one generate by Par4All is in Listing A.2.

```
1 int pi_parallel(int num_steps, double *pi)
2 {
3     //Declarations
4     double step;
5     double sum;
6     double x;
7     int i;
8
9     sum=0.;
10    step=(1.)/(num_steps);
11    i=0;
12    #pragma omp parallel private(i,x)
13    {
14    #pragma omp for reduction(+:sum) schedule(static)
15    for (i=0; i<num_steps; i=i+1){
16        x=((i)+(0.5))*(step);
17        sum=(sum)+((4.)/((1.)+((x)*(x))));
18    }
19    }
20    pi[0]=(step)*(sum);
21    return 0;
22 }
```

**Listing A.1:** Parallel version of pi calculation produced by Parallware.

```
1 int pi_code(int num_steps, double *pi)
2 {
3
4     double x;
5     double sum = 0.0;
6     double step = 1.0/((double) num_steps);
7
8     int i;
9 #pragma omp parallel for private(x) reduction(+:sum)
10    for(i = 0; i <= num_steps-1; i += 1) {
11        x = (i+0.5)*step;
12        sum = sum+4.0/(1.0+x*x);
13    }
14    *pi = step*sum;
15
16    return 0;
17 }
```

**Listing A.2:** Parallel version of pi calculation produced by Par4All.