Model Based Granularity Optimization for High-Performance Computing Systems in Astronomy

Ophélie Renaud

▶ **To cite this version:**

Ophélie Renaud. Model Based Granularity Optimization for High-Performance Computing Systems in Astronomy. Hardware Architecture [cs.AR]. INSA de Rennes, 2024. English. NNT : 2024ISAR0010 . tel-04984281

HAL Id: tel-04984281

https://theses.hal.science/tel-04984281v1

Submitted on 10 Mar 2025

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

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

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Signal, Image, Vision*

Par
Ophélie RENAUD

Model Based Granularity Optimization for High Performance Computing Systems in Astronomy

Thèse présentée et soutenue à Rennes, le 09 octobre 2024
Unité de recherche : IETR

Rapporteurs avant soutenance :

Kevin MARTIN — Maitre de conférences HDR, Lab-STICC, Université Bretagne-Sud
Christian PEREZ — Directeur de Recherche, INRIA ENS LYON

Composition du Jury :

Président :	Alix MUNIER KORDON	Professeur des Universités, LIP6, Sorbonne Université
Examinateurs :	Kevin MARTIN	Maitre de conférences HDR, Lab-STICC, Université Bretagne-Sud
	Christian PEREZ	Directeur de Recherche, INRIA ENS LYON
Dir. de thèse :	Jean-François NEZAN	Professeur des Universités, IETR, INSA Rennes
Encadrant :	Karol DESNOS	Maitre de conférences HDR, IETR, INSA Rennes

Invité(s) :

Adam DELLER — Astronome, CAS, Université de Swinburne

Acknowledgement

Je tiens tout d'abord à remercier mes directeurs de thèse, Jeff et Karol, pour m'avoir donné la chance de réaliser cette thèse. Merci patrons pour votre soutien constant et vos conseils toujours justes, empreints de bienveillance. Votre encadrement a été essentiel tout au long de ce parcours.

Un grand merci à Corinne pour toute l'aide précieuse que tu m'as apportée. Le laboratoire ne tournerait certainement pas aussi bien sans toi. Merci pour ton soutien, particulièrement dans les moments difficiles.

Je remercie également mes collègues de bureau, Hugo, Nicolas, Nicolas, Alban, Paul et Quentin, pour la joie de vivre que vous avez apportée au quotidien, ainsi que pour nos discussions toujours farfelues, mais tellement enrichissantes.

Merci à Meriem pour ta bienveillance tout au long de mon cycle ingénieur, et pour m'avoir ouvert les portes du monde de la recherche.

Enfin, un merci tout particulier à ma mamie. Merci pour tes messages remplis d'amour, ta fierté inébranlable et tes grands conseils qui m'ont tant aidée.

Table of Contents

INTRODUCTION

General Context

The rapid growth in computational power has led to the rise of High-Performance Computing (HPC) systems, transforming the possibilities across diverse applications. HPC enables the processing of data and execution of complex calculations at high speeds. For comparison, a laptop or workstation with a 3 GHz processor can perform around 3 giga-Floating-point Operations Per Second (FLOPS). While this figure already significantly exceeds human capabilities, it remains minimal compared to modern HPC solutions, which can perform up to tens of tera-FLOPS, such as the H100 GPU with 60 tera-FLOPS.

From the world's first supercomputer, the CDC 6600, designed by Seymour Cray in 1964, to the current Top 1 supercomputer according to the TOP500 [1], the Frontier, manufactured by HPE in 2022, the race for power is growing fast. Today, Frontier's remarkable processing speed of 1.102 exa-FLOPS highlights the significant advancements in HPC, demonstrating a substantial increase in computational power over the decades. However, reported performance figures are based on peak capabilities under optimal conditions. In practice, HPC systems are often are often highly utilized in terms of job occupancy but resource usage within these jobs can be inefficient. For instance, an analysis of NERSC's Perlmutter system revealed that around 64% of jobs used 50% or less of available memory, and 50% of GPU-enabled jobs used up to 25% of GPU memory, indicating inefficiencies in resource management [Li+23]. Other performance metrics, such as computational efficiency or memory bandwidth usage, can also highlight the gap between peak performance and real-world performance.

The continuous advancements in HPC capabilities, particularly with the integration of AI-driven workloads and heterogeneous computing architectures, have introduced both opportunities and challenges in optimizing the performance and resource management of these complex systems. As applications ranging from astronomical processing to cutting-edge artificial intelligence models like ChatGPT require increasingly vast computational resources, addressing the gap between theoretical performance and practical utilization

1. bi-annual ranking of the world's 500 most powerful HPC systems: `https://www.top500.org/`

remains crucial.

SKA and the Exascale Challenge

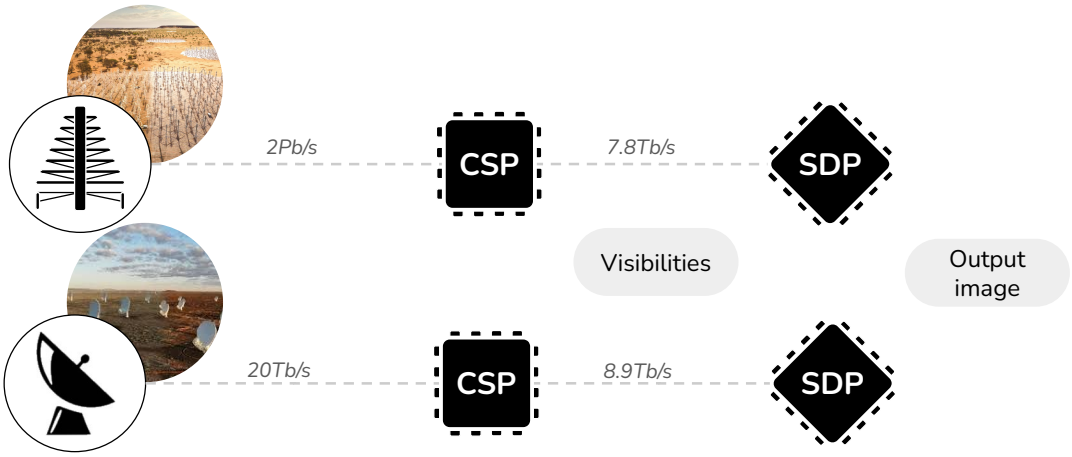

Figure 1 – Simplified representation of the SKA pipeline

The SKA [Dew+09], an advanced exascale radio telescope and the motivation of this thesis, marks a new era in astronomy. This groundbreaking project imposes significant computational challenges on its core, the Science Data Processor (SDP) pipeline. Responsible for handling data from telescopes at an impressive rate of several terabit per second, the SDP operates within constraints of limited storage and a strict energy budget of 5 Megawatt for 250 peta-FLOPS. To address the SKA computational challenges, the SDP supercomputer will integrate a not yet built HPC system. It will be hardly possible without efficient co-design methods and rapid prototyping tools.

The project is a collaboration between more than 10 countries, involving hundreds of scientists and engineers worldwide, including France [Ace+17]. In parallel, the Dataflow Algorithm aRchitecture co-design of SKA pipeline for Exascale Radio Astronomy (Dark Era) consortium [Cha+21] [2], with the support of SKA-France and in collaboration with Eviden, aims to develop advanced tools and technologies to support astronomical research. Their goals include creating exascale simulation tools, exploring new energy-efficient computer architectures, and contributing to the development of SKA computing infrastructure. This initiative aligns with RISE International Network for Solutions Technologies

2. `https://dark-era.pages.centralesupelec.fr/`

and Applications of Real-time Systems (Rising STARS) [3] commitment by encouraging young researchers who can contribute to and benefit from these innovative advancements in radioastronomy and computational sciences.

From Standard Computing System to modern HPC System

Computing systems, as outlined in [Des14], are fundamentally comprised of two main components: hardware and software.

— **Hardware** refers to the physical components and electronic circuitry of a computing system, encompassing elements such as processors, memory modules, storage devices, input/output devices, and other tangible components that enable the execution of computational tasks. It forms the foundation of a computing system, providing the necessary infrastructure for software applications to run and interact with the user.

— **Software** refers to the intangible, non-physical instructions, programs, and data that control and coordinate the operation of a computing system. It encompasses a wide range of applications, operating systems, utilities, and programming languages that enable users to perform specific tasks, manage hardware resources, and interact with the computer. Software acts as the intermediary between the user and the hardware, providing the set of instructions and algorithms necessary for the hardware components to execute and fulfill various computational functions.

Expanding on the hardware and software components, constructing a HPC architecture involves interconnecting compute servers, known as nodes, to form clusters. These clusters facilitate the simultaneous execution of software programs and algorithms on servers within the same cluster, enabling parallel processing. Additionally, the clusters are integrated with data storage to preserve results. This integrated setup of compute nodes and storage components ensures efficient execution of various tasks. In essence, supercomputers designed for standard HPC share the foundational elements of conventional desktop computers, such as processors, memory, disk storage, and an operating system. However, each of these components is notably more powerful, and the architecture incorporates a higher quantity of them. This enhanced configuration, coupled with the interconnection of nodes into clusters, enables supercomputers to deliver high computational performance.

3. `https://www.risingstars-project.eu/`

Reducing the Design Productivity Gap

As Moore's Law [Moo98], which predicts a doubling of transistors within integrated circuits approximately every two years, faces its limits, particularly with diminishing returns in processor frequency gains, multi-core processors and Graphics Processing Units (GPUs) systems have emerged as a strategic response to these challenges, enabling increased parallelism and efficiency. These systems leverage parallelism to overcome the constraints imposed by hardware limitations. The introduction of system heterogeneity further enhances their efficiency. Additionally, a new paradigm known as *More than Moore* [Wal16] is coming to the forefront, acknowledging the need to explore beyond traditional transistor scaling to sustain advancements in computing capabilities.

However, the complexity of these systems makes them increasingly intricate to manage introducing a productivity gap [Hoe12]. The *design productivity gap* and *software productivity gap* represent disparities between hardware advancements and the efficiency of design and software development processes. The former reflects challenges in designing complex hardware systems, while the latter indicates difficulties in creating software that fully utilizes evolving hardware capabilities. Closing these gaps is essential for maximizing the potential of computing technologies.

As mentioned in [Des14], one contributing factor to the software productivity gap is the continued popularity of a low-level imperative language, like the C programming language, which is not well suited for modern massively parallel and heterogeneous systems. Recognizing these challenges, recent efforts in HPC focus on the development of high-level programming languages and rapid prototyping techniques. Paradigm such as Open Multi-Processing (OpenMP) [CJV07], Open Computing Language (OpenCL) [Khr11] or dataflow, detailed in Chapter 1, aim to enhance software productivity by facilitating the use of accelerators like GPUs and Field-Programmable Gate Arrays (FPGAs). While OpenMP emerged after the limitations of High Performance Fortran, and OpenCL serves as an open standard for accelerator programming, no single solution has yet gained widespread adoption.

The Dataflow Paradigm

The dataflow model introduced by Kahn in [Kah74] is popular for its accessibility and performance. This model is designed to capture parallelism and consists of modeling algorithms through a graph where nodes represent individual calculations, and directed

arcs depict the data exchanges between these nodes. This graphical representation allows for a clear visualization of the parallel execution of tasks, making it easier for programmers to conceptualize and implement parallel algorithms efficiently.

The SKA presents a compelling use case for the dataflow model due to its complex, large-scale data processing requirements. While the dataflow approach is still emerging in the HPC landscape, its potential to efficiently manage and distribute computational tasks across a vast network of telescopes and data processing facilities could lead to significant improvements in efficiency and scalability. Exploring the dataflow model in the context of SKA may provide insights into how this paradigm can better meet the growing demands of modern astronomical data processing, following the example of promising but under-utilized implementations seen in frameworks such as Apache Flink and TensorRT. Therefore, the approach taken in this thesis is that of dataflow. Existing tools such as Data Activated 流 Graph Engine (DALiuGE) [Wu+17] and Dask [Roc15] have been investigated to parallelize on HPC systems. These framework have been designed to facilitate the distribution of tasks across multiple Processing Element (PE), aiming to improve efficiency in handling large datasets. However, they currently face significant challenges when it comes to scaling up to exascale applications, which require not only higher performance but also the ability to manage huge data flows and complex interdependencies. This thesis explores strategies to improve dataflow to address large-scale complexity.

Scope of this Thesis and Contributions

This thesis addresses the complex challenges associated with both standard and modern HPC systems. It aims to explore methodologies such as Algorithm-Architecture Matching (AAM), previously Algorithm-Architecture Adequation (AAA) and initially developed for embedded systems [GLS99], aimed at optimizing resource utilization, enhancing software productivity, and advancing the co-design of HPC architectures and applications. This adaptation aims to leverage the unique demands and opportunities presented by HPC environments, contributing to the originality of this research.

The main contributions of this thesis are:

1. A methodology focusing on optimizing resource allocation at the granularity level of multi-core processors. The method is based on clustering and accelerates both the resource allocation process and the application execution. This contribution has been published in [Ren+23a], and extended in [Ren+23b] and in [Ren+24a].

2. A methodology for distributing resources among several HPC heterogeneous processors. The method is based on a balanced workload partitioning and facilitates the fast and efficient deployment of data-intensive applications on complex target architectures. This contribution has not been published yet. However, a pre-print version is available through [Ren+ed].
3. A co-design methodology for HPC application. The proposed method allows to automatically identify a suitable multi-node, multi-core, and network topology architecture for a given application. The proposed method facilitates the in-depth Design Space Exploration (DSE) to identify Pareto-optimal solutions in HPC systems. This contribution has been published in [Ren+24b].

Figure 2 illustrates how our contributions are related to the design process. All have been implemented in the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) application development framework within the IETR's VAADER team.

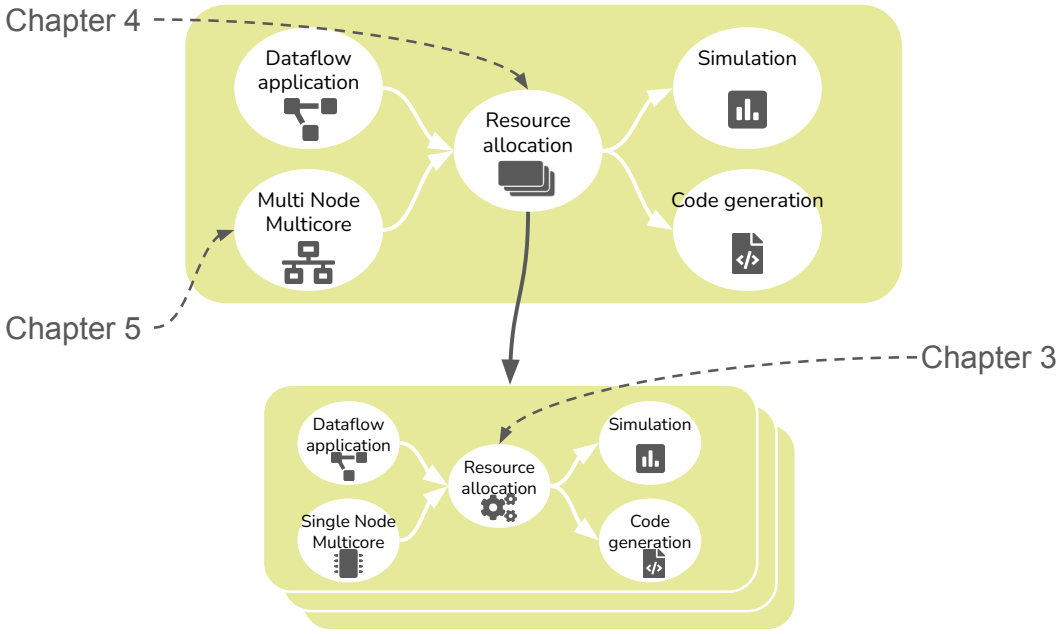

Figure 2 – Flow chart of the contributions

Outline

This thesis is organized into two parts: Part I presents the general context and motivation of this thesis and Part II introduces and evaluates the contributions of this thesis.

In Part I, Chapter 1 provides a general overview of the HPC architecture representation and presents the AAA process. Then, Chapter 2 presents the main programming model paradigms for parallel computations along with dataflow scheduling optimization techniques.

In Part II, Chapter 3 discusses the optimization of dataflow application granularity through static analysis for homogeneous single-node multicore architectures. Then, Chapter 4 elaborates on the static distribution of dataflow applications on heterogeneous multi-node multi-core architectures. Chapter 5 introduces a method for deploying and evaluating the performance of applications across diverse multi-node networks. Finally, Chapter 6 concludes this work and proposes potential research directions for future research.

Part I

Background & Motivations

Chapter 1

Hardware/Software Co-Design Requirements for SKA

Introduction

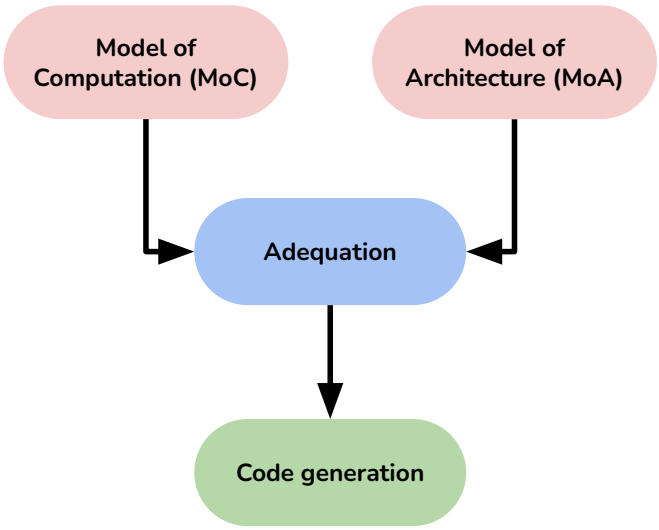

Figure 1.1 – Y-chart representation of the AAA Methodology

In addition to progress in the semiconductor industry, coupled with advances in compilation and synthesis tools, the search for a better match between algorithms and architectures is proving to be a key lever for increasing computing speed. This chapter is constructed to demonstrate how the Algorithm-Architecture Adequation methodology can address the matching problem at the exascale level of SKA. The methodology AAA was originally developed for embedded systems, leading to the development of academic or industrial rapid prototyping tools such as Synchronized Distributed Executive (SynDEx)

[Sor04], PREESM [Pel+14] or Spear [Pet+11], for example, to automate architectural exploration. The AAA methodology is based on a Y-chart representation as illustrated in Figure 1.1. The Y-chart model, introduced in [Kie+97], consists of separating into two independent models the application and the architecture design. The AAA methodology presented in [GLS99] uses graph models to specify both the algorithm, referred to as MoC, and the hardware architecture, referred to as Model of Architecture (MoA). Subsequently, the matching phase, known as adequation, maps the application to the architecture through graph transformation, resulting in an optimized graph implementation of the application on the architecture.

In this chapter, Section 1.1 shows how graph-based MoA can represent both embedded and HPC systems and Section 1.2 presents an overview of programming MoCs, including dataflow MoC.

1.1 High-Performance Computing MoA

The concept of MoA is defined in [Pel17] as follow:

Definition 1. A MoA is an abstract efficiency model of a system architecture that provides a unique, reproducible cost computation, unequivocally assessing an architecture efficiency cost when supporting the activity of an application described with a specified MoC.

As mentioned in [Pel10], the most common architecture models are based on the Von Neumann model, characterized by its fundamental elements: a memory unit, an Arithmetic and Logic Unit (ALU), input / output units and a control unit as illustrated in Figure 1.2. It operates as a black box, executing instructions, managing memory, and facilitating communication with external devices. This architecture may function as a processor core or a specialized co-processor, depending on its design and tasks. In this section, the focus is put on system-level models and design rather than on detailed hardware design, already been addressed by large sets of existing literature. One example of system-level models is the System-Level Architecture Model (S-LAM) [Pel+09b]. S-LAM defines a quasi-MoA $\Lambda = \langle M, L, t, p \rangle$ where M is the set of components, L is the set of links connecting them, and t and p respectively give a type and a property to component. The following section details the elements involved in modeling an HPC system decomposed into two categories: *node modeling* and *network modeling*. Here a node relates to a

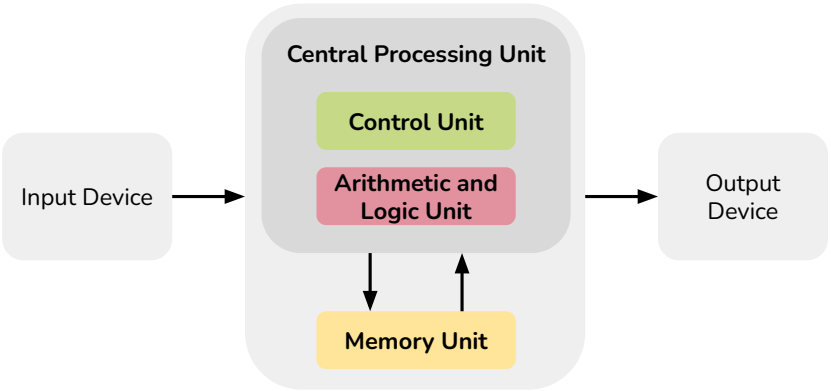

Figure 1.2 – Illustration of the Von Neuman model architecture

computer machine. The *node modeling* refers to the process of modeling mathematical or computational representations of individual computing units within a system architecture. This involves characterizing the performance, capabilities, and interactions of each node within the system, including factors such as processing power, memory, communication interfaces, and latency. The *network modeling* refers to the process of creating mathematical or computational representations of the topology and behavior of a network infrastructure. This includes modeling the connections, bandwidth, latency, and other characteristics of the network components such as routers, switches, and cables.

1.1.1 Single-Node and Multi-Node Architectures

Single-node systems refer to a configuration in which all computational resources, including processors, memory, and storage, are housed within a single machine. This configuration is characterized by its ability to efficiently handle parallel tasks within the constraints of a standalone system. Within the realm of single-node systems, it is essential to distinguish between the following configurations:

— **Multi-core System**: A Multi-core system involves the integration of multiple processor cores onto a single chip. Each core operates independently, allowing for parallel processing within a single machine. The coordination between cores is managed by the operating system, enhancing computational throughput and responsiveness. This configuration is particularly advantageous for tasks that can be parallelized at the core level. Consider a typical Multi-core system, such as the Intel Core i9

processor, which features multiple cores capable of executing parallel tasks concurrently.

Definition 2. Considering a Multi-core system, let C denote the set of processor cores within a single machine. Each core $c_i \in C$ operates independently, contributing to parallel processing capabilities.

$$C = \{c_1, c_2, ..., c_n\}$$

— **Multi-processor System**: In a Multi-processor system, distinct processors, each with its own set of cores, are employed within a single machine. These processors work collaboratively on parallel tasks, providing enhanced processing power. The coordination between processors is facilitated by specialized communication channels, and this configuration proves beneficial for applications demanding a higher degree of parallelism. For instance, while an AMD EPYC server processor is indeed a multi-core processor, a multi-processor system might include several of these EPYC processors working together within the same server unit, thus achieving parallelism at both the processor and core levels.

Definition 3. For a Multi-processor system, let P represent the set of processors within a single machine, each equipped with its own set of cores. Processors collaborate on parallel tasks, enhancing overall processing power.

$$P = \{p_1, p_2, ..., p_m\}, \quad p_i = \{c_{i1}, c_{i2}, ..., c_{ik}\}$$

Multi-node configurations contrast single-node systems, Multi-node HPC systems extend computational capabilities across multiple interconnected machines. In this configuration, each node consists of multiple processors, and communication between nodes is essential for parallel processing. Examples include clusters of servers, where each server houses multiple processors.

Definition 4. For a Multi-node system, let N represent the set of interconnected nodes, with each node $n_k \in N$ having multiple processors or cores for fine-grained parallelism. Communication between nodes enables coarse-grained parallelism.

$$N = \{n_1, n_2, ..., n_o\}, \quad n_k = \{c_{k1}, c_{k2}, ..., c_{kr}\}$$

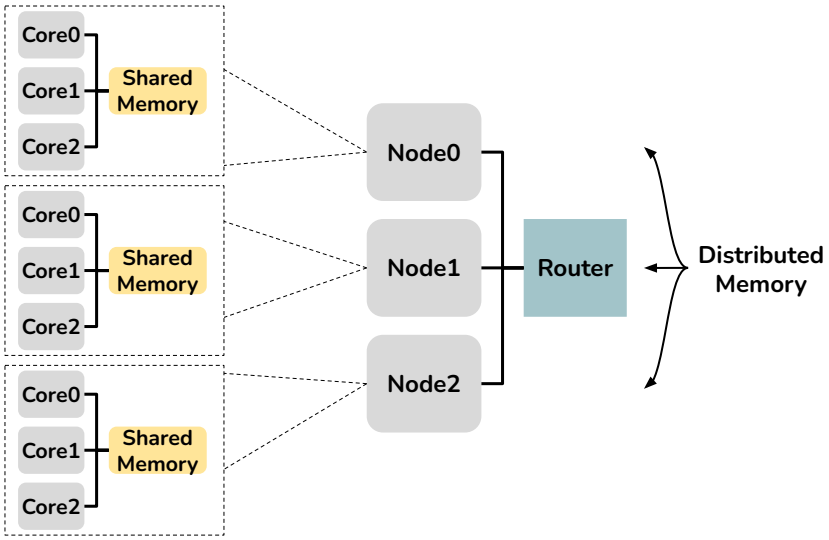

Figure 1.3 – Illustration of a Multi-node and Multi-core architecture

The primary objective of HPC systems is to increase parallelism to enhance speedup. Speedup is a key metric in evaluating the performance of parallel systems and quantifies the improvement achieved by using parallel processing compared to sequential execution. Formally, speedup (S) is defined as the ratio of the execution time of a task on a sequential system (T_1) to the execution time on a parallel system with p processors (T_p). This relationship is expressed by the equation:

$$S = \frac{T_1}{T_p}$$

Where S represents speedup, T_1 denotes the execution time on a sequential system, and T_p represents the execution time on a parallel system with p processors. An ideal parallelization scenario would yield a linear speed increase proportional to the number of cores ($S = p$), indicating perfect scalability. However, in certain cases, such as when cache effects are favorable, the performance could exceed linearity, resulting in superlinear speedup. This occurs when the overhead of memory access is reduced due to better utilization of the cache hierarchy, leading to enhanced efficiency as more cores are employed. However, two fundamental laws, Amdahl's Law and Gustafson's Law, provide valuable insights into the practical limitations of parallelization. Amdahl's Law underscores the inevitable limitations imposed by the sequential portion of a program on the

achievable speedup. Formally, this can be expressed as:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

where S is the speedup, p is the proportion of the program that can be parallelized, and n is the number of processors. This principle reminds us that parallelism alone cannot indefinitely improve the performance of HPC systems. Conversely, Gustafson's Law offers a contrasting viewpoint. It posits that by scaling the size of the problem with the number of processors available, the overall speedup can increase linearly, expressed as:

$$S = p \times n + (1-p)$$

This perspective aligns with the nature of many HPC applications, where larger problem sizes can be effectively tackled with additional computational resources. However, it is essential to consider factors such as communication overhead and load imbalance that can influence the actual speedup achieved in real-world HPC applications.

1.1.2 Homogeneous and Heterogeneous Configurations

Homogeneous Systems consist of uniform hardware components, where all nodes and processors share similar architectures and capabilities. This uniformity simplifies resource management and programming but may pose challenges in accommodating diverse workloads, such as handling varying data types, differing computational requirements, and optimizing performance across multiple applications. For instance, a homogeneous system may struggle to efficiently process both memory-intensive and compute-intensive tasks simultaneously, potentially leading to resource under-utilization. The single-node and multi-node architectures in these systems often mirror each other in terms of hardware specifications.

Definition 5. Let P represent the set of processors within an HPC system. The system is deemed homogeneous if all processors $p_i \in P$ share identical architectures, instruction sets, and capabilities.

$$P = \{p_1, p_2, ..., p_n\}, \quad \forall p_i,\ \text{architecture}(p_i) = \text{architecture}(p_j), \forall p_j \in P$$

Such a system can be modeled with the Linear System-Level Architecture Model

(LSLA) model introduced in [Pel+18]. LSLA defines MoA $\Lambda = \langle P, C, L, \hat{c}, \lambda \rangle$ where P is the set of PEs, C is the set of Communication Nodes (CNs), L is a set of undirected links connecting PEs and CNs, $\hat{c}$ is a property function associating a cost to different elements in the model, and λ is a Lagrangian coefficient setting the cost of a single communication quantum relative to the cost of a single processing quantum. LSLA is categorized as linear because the computed cost is a linear combination of the costs of its components.

Heterogeneous Systems, in contrast, incorporate diverse hardware components with varying architectures, accelerators, and specialized processors. While offering the potential for enhanced performance and energy efficiency for specific workloads, managing heterogeneity introduces complexities in programming models, resource allocation, and system optimization. Although heterogeneous systems, such as those using GPUs—specialized processors that accelerate image and video processing through parallel execution— and FPGAs—versatile integrated circuits that can be programmed post-manufacturing for customized hardware acceleration—are becoming increasingly common in HPC, this thesis does not focus on their use. The contribution proposed in Chapter 4, focusing on distributed resource allocation, is particularly relevant in the context of standard heterogeneous HPC configurations. A GPU-accelerated system is an example of a heterogeneous architecture. In such a system, Central Process Units (CPUs) and GPUs coexist, each with its specialized architecture. This heterogeneity allows the system to leverage the strengths of both processors and accelerators, enhancing overall computational performance.

Definition 6. Consider a system with a set of processors and accelerators denoted by H. The system is heterogeneous if there exist distinct architectures $A_1, A_2, ..., A_m$ such that each processor or accelerator $h_i \in H$ corresponds to a specific architecture.

$$H = \{h_1, h_2, ..., h_k\}, \quad \exists A_j : \text{architecture}(h_i) = A_j, \forall h_i \in H$$

Such a system can also be modeled with the LSLA model, although some research suggests the use of non-linear modeling to better accommodate constraints, leading to the proposal of the GPU-oriented System-Level Architecture Model (GSLA) [PPH21]. The GSLA also defines MoA $\Lambda = \langle P, C, L, \hat{c}, \lambda \rangle$ where $\hat{c}$ is defined as a non-linear function. This approach allows for the modeling of cost variations between the parallelism of a multi-core CPU and that of a GPU coprocessor kernel. Consequently, GSLA offers a more nuanced representation of system behavior, particularly in heterogeneous computing environments

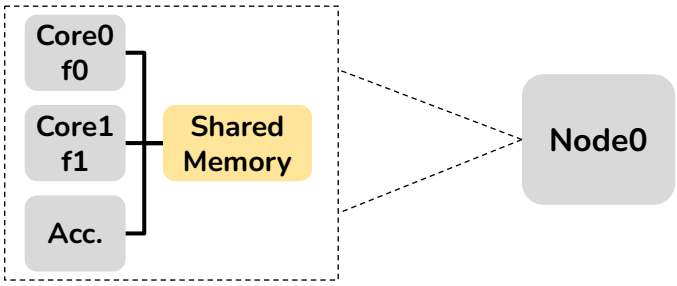

Figure 1.4 – Illustration of a heterogeneous architecture

where different processing units exhibit distinct parallelization characteristics.

1.1.3 Memory architecture

In the domain of architectural models, literature commonly classifies them into two main categories, namely Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC), based on the complexity of their instruction sets [Hea95]. The Flynn taxonomy [Fly72] classifies architectures based on instruction and/or data parallelism, encompassing Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD) as illustrated in Figure 1.5.

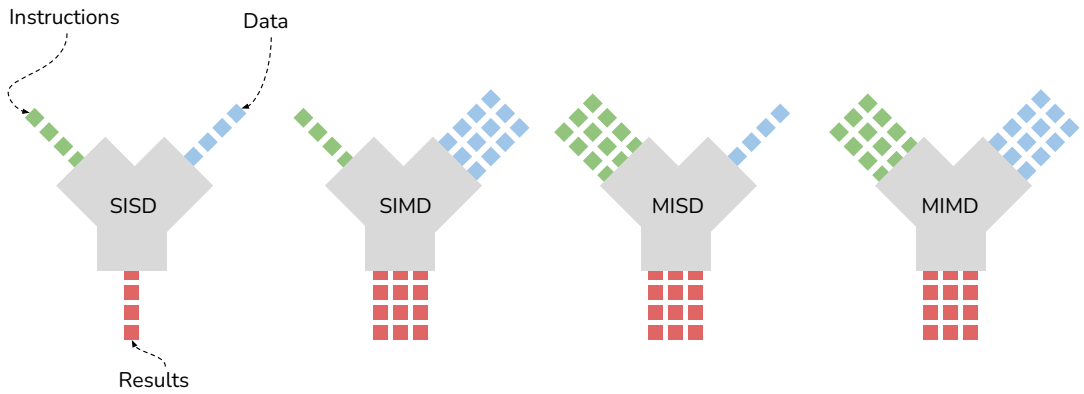

Figure 1.5 – Illustration of the Flynn taxonomy

MIMD, prevalent in CPU-based Multi-core architectures, further categorizes based on memory architecture into Uniform Memory Access (UMA), Non Uniform Memory

Access (NUMA), and NO Remote Memory Access (NORMA) [Baz+00]. These categories delineate the interaction between instructions and memory:

— **UMA**: These architectures adhere to the shared memory model, where all cores share a single physical global address space. Commonly used in Symmetric Multi-Processor (SMP), this parallel programming model allows multiple cores to access a unified memory, often equipped with a full cross-bar to handle memory transactions efficiently. However, as the number of cores increases, UMA architectures may suffer from performance degradation due to shared memory and memory arbiters, leading to memory access conflicts. Instructions and data are fetched from and stored back to this shared memory space, facilitating seamless communication between the processing units.

— **NUMA**: These architectures feature distributed memory with transparent memory accesses. Transparent memory access enables cores to access the global address space through their cache memory hierarchy. NUMA architectures are easier to program than distributed memory architectures, as data communication between cores is concealed by various levels of data caches, automating the communication process. However, scaling poorly onto massively parallel architectures, int [Has18] the author mentions the NUMA effect, where processors experience varying memory latency access times. Additionally, NUMA systems usually implement cache coherence, known as Cache Coherent NUMA (CC-NUMA) architectures, resulting in significant coherence traffic during data updates (data sharing). Efficient utilization of NUMA computer systems necessitates placing accessed memory buffers as close as possible to the computing resources, the cores. Instructions are fetched from memory and executed by the cores, while data is transferred between memory and caches to facilitate computation.

— **NORMA**: This architecture lacks shared memory, necessitating communication between operators through messages. Here, the instructions communicate with data via message-passing mechanisms, marking a departure from centralized memory access.

This thesis specifically discusses Distributed Computing System (DCS) that incorporate HPC capabilities. These systems are characterized by multiple interconnected nodes, where each node may consist of multi-core processors or specialized accelerators. Within a node, a UMA architecture is typically employed, facilitating equal access times to memory

for all processors within the node. Conversely, between nodes, the communication model often adheres to a NORMA architecture paradigm, wherein nodes do not have direct access to the memory of remote nodes.

1.1.4 Network Topologies

Table 1.1 – Advantages and Limitations of HPC Network Topologies

Topology	Advantages	Limitations
Cluster with Crossbar	Full-speed communication between any pair of hosts. No contention on the switch. Suitable for homogeneous clusters.	High cost due to crossbar switches. Scalability challenges beyond a certain number of hosts.
Cluster with Shared Backbone	Efficient use of shared communication medium (backbone). Suitable for moderate-sized clusters.	Contention on the shared backbone. Limited bandwidth capacity.
Torus Cluster	Regular, structured interconnection. Minimal latency for nearest-neighbor communication.	Complex wiring and routing. Limited scalability beyond a certain dimension.
Fat Tree Cluster	High bisection bandwidth. Scalable and fault-tolerant.	Requires additional switches and cabling. Initial setup complexity.
Dragonfly Cluster	Scalable and fault-tolerant. Low global latency.	Complex routing algorithms. High initial cost.

The interconnection network topology in HPC systems impacts the overall system performance including latency, bandwidth, and fault tolerance. Understanding and exploiting these topologies are essential for optimizing communication between nodes and achieving efficient parallelization. Here, we delve into five prominent network topologies [1]: Cluster with Crossbar, Cluster with Shared Backbone, Torus Cluster, Fat Tree Cluster, and Dragonfly Cluster illustrated in Figure 1.6 and summarized in Table 1.1.

Cluster with Crossbar

A cluster with a crossbar topology features a central crossbar switch that directly connects all processors as illustrated in Figure 1.6 (a). Originating from the earliest supercomputer designs by Seymour Cray, notably the Cray-1 and Cray-2, this topology

1. `https://simgrid.org/doc/latest/Platform_examples.html`

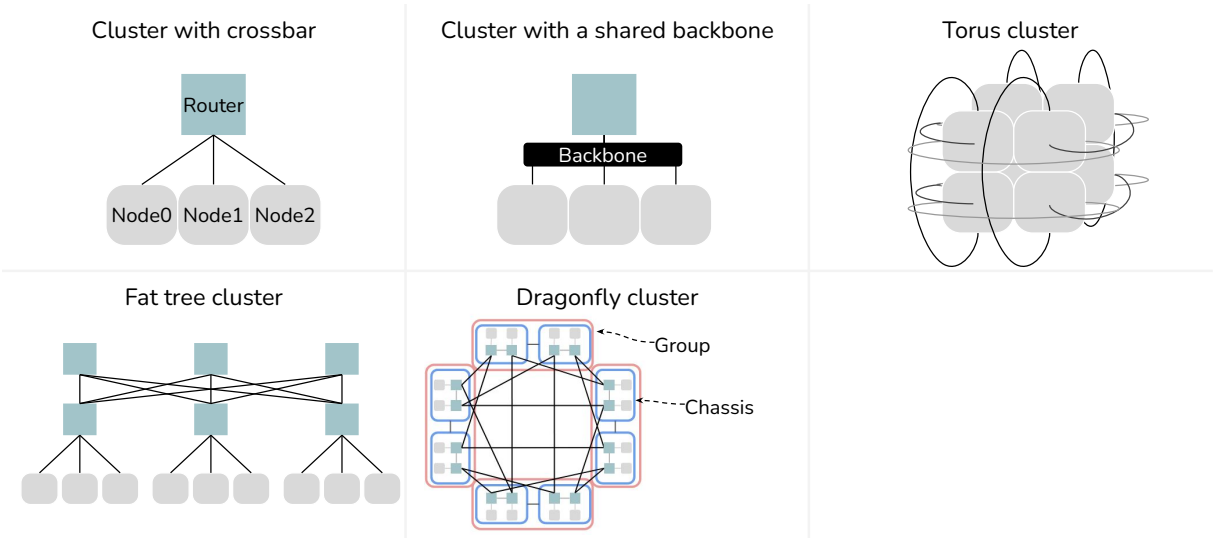

Figure 1.6 – Illustration of the five main network topology

ensures efficient and non-blocking communication between nodes, making it well-suited for bandwidth-intensive and low-latency applications. Although scalability has been improved in some works [BJ15], the number of nodes that can be connected remains limited by the switch ports. This topology is particularly relevant for applications requiring frequent and unhindered communication between processors, such as scientific simulations demanding high inter-node communication rates.

Cluster with Shared Backbone

In this topology, illustrated in Figure 1.6 (b), processors are interconnected through a shared backbone, where multiple nodes share the same communication path. It provides a centralized communication medium, promoting simplicity and ease of management. A cluster with a shared backbone is demonstrated in traditional Ethernet-based HPC clusters. In this setup, nodes connect to a common switch or switches, forming a shared communication backbone for inter-node communication. This topology is relevant in scenarios where simplicity and cost-effectiveness are prioritized over the demand for extremely high bandwidth. It is suitable for applications with moderate communication requirements.

Torus Cluster

A torus cluster represents a network topology devoid of switches, resembling a mesh interconnect with nodes organized in a rectilinear array across N dimensions. Each node establishes connections with its neighbors in a specific pattern, facilitating multiple communication paths and minimizing latency, as illustrated in Figure 1.6 (c). One notable torus family is the k-ary n-cube [GW99] topologies, where n represents the dimension, and each dimension consists of k nodes. A variant known as the 3-D Asymmetric Torus Network deviates from traditional torus networks by featuring varying dimensions along its axes, thereby enabling efficient communication between nodes while optimizing interconnectivity within the system. Routing algorithms have been developed for such asymmetric systems [KA11]. For instance, the IBM Blue Gene supercomputer employs a torus cluster topology, connecting nodes in a three-dimensional toroidal arrangement. Torus clusters are particularly well-suited for applications characterized by regular communication patterns, such as computational fluid dynamics simulations, where neighboring nodes need to exchange data frequently. Torus cluster can be represented with the following canonical forms.

Definition 7. For a Torus network, let $T(x, y, z)$ be defined as follows:

x	is the number of nodes in the first dimension,
y	is the number of nodes in the second dimension,
z	is the number of nodes in the third dimension,.

Example 1. As an example, let's consider the 2-ary 3-cube Torus cluster in Figure 1.6. The canonical form of the torus is: $T(2, 2, 2)$

Fat Tree Cluster

Introduced in [Lei85], fat-tree cluster topology follows a tree structure, where switches are arranged in multiple layers. This design provides redundant paths and high bandwidth, ensuring fault tolerance and efficient communication. Numerous studies have focused on fat trees, with the k-ary n-tree serving as the fundamental type of tree constructed from switches with an equal number of connections going both up and down the tree, as outlined by Petrini et al. [PV97]. This foundational concept was later expanded upon by Ohring [Ohr+95] through the introduction of Generalized Fat Trees (GFT), allowing for a vary-

ing number of upward and downward connections. The Extended Generalized Fat Trees (XGFT) further broadened the scope of possible topologies by accommodating a different number of connections at each level. Parallel ports Generalized Fat Trees (PGFT), as introduced by Jacobs et al. [Jac10], represents an extension of XGFT, incorporating parallel links to preserve cross-bisectional bandwidth. Moreover, Jacobs et al. [Jac10] present the canonical form of PGFT, which will be subsequently used in this paper.

Definition 8. For a Fat Tree network, let $PGFT(h; m_1, ..., m_h; w_1, ..., w_h; p_1, ..., p_h)$ be defined as follows:

- h is the number of levels in the tree,
- m_1 is the number of different lower-level nodes connected to nodes on level noted l,
- w_1 is the number of different upper-level nodes connected to nodes on level $l-1$,
- p_1 is the number of parallel links connecting between nodes in level l and $l-1$.

Example 2. As an example, let's consider the Fat tree cluster in Figure 1.6. The canonical form of the tree is: $PGFT(2; 3, 3; 1, 3; 1, 1)$

Dragonfly Cluster

The Dragonfly topology, a prominent alternative to the Fat Tree cluster, was introduced by Kim et al. [Kim+08] to reduce network costs and diameter while enhancing scalability. This architecture is characterized by interconnected groups of compute nodes. The Dragonfly+ model, as proposed by Shpiner et al. [Shp+17], extends the Dragonfly topology by incorporating additional links between node groups, thereby improving both bandwidth and latency. For instance, the Cray XC30 supercomputer employs a Dragonfly topology in which compute nodes are organized into groups, and these groups are interconnected through high-speed links. This configuration facilitates scalable and fault-tolerant communication. The Dragonfly topology implemented in the Cray XC series is based on the Cray Cascade system [Faa+12], with the canonical form instantiated as follows.

Definition 9. For a Dragonfly network, let $D(G, G_{\text{link}}; C, C_{\text{link}}; R, R_{\text{link}}; N_{\text{pr}})$ be defined

as follows:

G	is the number of groups,
G_{link}	is the number of links between groups,
C	is the number of chassis per group,
C_{link}	is the number of links between chassis,
R	is the number of routers per chassis,
R_{link}	is the number of links between routers,
N_{pr}	is the number of nodes per router.

Example 3. As an example, let's consider the Dragonfly cluster in Figure 1.6. The canonical form of the dragonfly is: $D(4; 4, 2; 1, 2; 1, 1)$

Understanding the nuances of each network topology is crucial in selecting the most appropriate configuration for specific HPC applications. This entails considering various factors such as communication patterns, fault tolerance, and scalability requirements. Table 1.1 provides a concise overview of the advantages and limitations associated with the five prominent network topology families.

1.2 Programming MoC

MoCs are crucial in theoretical computer science for evaluating the performance of algorithms and computing machines [Sav97]. They allow developers to understand the limits of computation, analyze the complexity of problems, design more efficient algorithms, and explore parallelism. By studying different models of computation, we can optimize memory usage, study the power of parallel computing, and develop practical applications in hardware design, programming language compilation, and performance optimization.

1.2.1 Multithreading and Synchronization in Shared-Memory System Using MoC

Task-based Programming MoC are programming paradigms that organize code around autonomous and independent units of work known as tasks. In these models, programs are structured into tasks that represent portions of the computation or activity to be executed.

Tasks can be explicitly defined by the programmer and can be executed concurrently, providing a flexible approach to exploit multi-core and distributed architectures. *Task-based Programming MoCs* aims to simplify parallelism by allowing developers to specify dependencies between tasks and letting the runtime system handle task scheduling and distribution across available resources. This section explores two prominent aspects of *Task-based Programming MoC* on shared-memory systems: Processes & Threads, with a focus on POSIX Threads (Pthreads) and OpenMP.

Processes & Threads constitute fundamental building blocks in parallel programming, allowing concurrent execution of tasks and operations. A *process* represents an instance of a program in execution, encapsulating its code, memory, registers, and other resources. Each *process* operates independently, providing isolation and stability to the system. *Threads*, on the other hand, are the smallest units of execution within a *process*, capable of being scheduled independently. *Threads* share the same memory space within a *process*, enabling efficient communication and resource sharing. In a single-threaded process, as illustrated in Figure 1.7, computations are executed sequentially by a single thread, while multi-threaded processes allow for concurrent execution of computations by multiple threads. Multi-threading can enhance performance by leveraging parallelism, although careful synchronization is required to avoid issues like data races. Overall, processes and threads provide a flexible and powerful framework for managing and executing programs in modern computer systems. However Multi-threaded application development is dependent on both Operating System (OS) and programming languages [Mio22]. Lower-level languages like C use OS-specific threading, while higher-level languages like Java abstract threading. Thus, libraries like Pthreads and OpenMP simplify multi-threaded development across languages.

Pthreads is a standardized library, or Application Programming Interface (API), for thread creation and management [LB98]. It is widely used in Unix-like operating systems to harness parallelism through multithreading. Offering developers the ability to finely control multi-threaded execution and thread synchronization, Pthreads facilitates parallel execution and optimal resource utilization. Noteworthy features of Pthreads include its capabilities for:

— Thread Creation and Management: Pthreads provide functions for creating, managing, and synchronizing threads. Developers can spawn threads, assign tasks, and control their lifecycle through the Pthreads library.

— Synchronization Mechanisms: Pthreads offer synchronization mechanisms, including

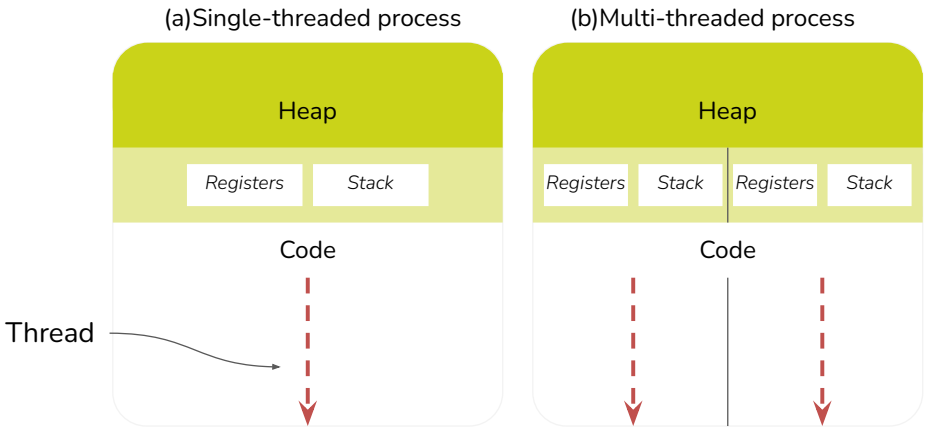

Figure 1.7 – Illustration of processes and threads

(a) Simple example using Pthread

```c
#include <pthread.h>

#define NUM_THREADS 4

void* task_func(void* arg) {
 // Perform task-specific computations here
}

int main() {
 pthread_t threads[NUM_THREADS];

 for (long i = 0; i < NUM_THREADS; ++i) {
  pthread_create(&threads[i], NULL, task_func, (void*)i);
 }

 for (long i = 0; i < NUM_THREADS; ++i) {
  pthread_join(threads[i], NULL);
 }

 return 0;
}
```

(b) Simple Example using OMP

```c
#include <omp.h>

#define NUM_THREADS 4

void task_func(int thread_id) {
 // Perform task-specific computations here
}

int main() {
 #pragma omp parallel num_threads(NUM_THREADS)
 {
  task_func(omp_get_thread_num());
 }

 return 0;
}
```

Figure 1.8 – Simple example creating 4 threads, each executing a task in parallel using Pthreads and OpenMP multithreading

mutexes, condition variables, and semaphores, to ensure proper coordination and avoid data race conditions among threads.

— Thread Safety: The API is designed with thread safety in mind, allowing safe concurrent execution of functions and data manipulation within a multi-threaded environment.

Figure 1.8 (a) shows a simplified code snippet employing Pthreads to parallelize computational workload across 4 threads. Pthreads-specific functions are called to appropriately create and terminate threads, while the workload is encapsulated within a function for distribution.

OpenMP is an API that supports multi-platform shared-memory multiprocessing programming in C, C++, and mathematical FORmula TRANslating system (FORTRAN). OpenMP simplifies parallel programming by employing compiler directives to indicate parallel regions, making it particularly well-suited for task-level parallelism. Notable features of OpenMP include:

— Compiler Directives: OpenMP uses pragmas (compiler directives) to annotate code, indicating regions suitable for parallel execution. These directives guide the compiler in generating parallel code.

— Worksharing Constructs: OpenMP provides worksharing constructs such as parallel loops, sections, and tasks, allowing developers to express parallelism explicitly.

— Automatic Parallelization: Some compilers support automatic parallelization of code marked with OpenMP directives, making it accessible for developers without extensive parallel programming expertise.

— Scalability and Portability: OpenMP facilitates scalability across various architectures and is known for its portability, enabling developers to write parallel code that can run efficiently on different platforms.

— Recent versions (OpenMP 4.0 and beyond): In more recent versions, significant enhancements have been introduced such as supporting SIMD operations, improving dependency management, and supporting GPU offloading.

Figure 1.8 (b) shows a simplified code snippet employing OpenMP to parallelize computational workload across 4 threads.

1.2.2 Distributed-Memory Multiprocessing Programming

Distributed systems programming involves designing and developing software that runs on several computers linked by a network. The most prominent techniques are task-based programming and message passing.

Task-Based and Data-Parallel MoC on Distributed-Memory System

In distributed memory systems, where each processor has its own private memory, task-based computing models have to tackle additional challenges related to task communication and synchronization. It is necessary to efficiently manage data transfer between processors and to guarantee consistency in the state of shared tasks.

OmpSs, introduced by Barcelona Supercomputing Center (BSC) in [Dur+11], is an API designed to extend OpenMP to efficiently exploit asynchronous parallelism and heterogeneous architectures. Originally developed for SMP architectures, it was later expanded into Open Multi-Processing Super scalar (OmpSs)2@Cluster in [Agu+22] to support DCS architectures. Notable features of OmpSs include:

— Compilation directives: OmpSs uses pragmatic directives to annotate code, indicating regions suitable for parallel execution. These directives guide the compiler in generating efficient parallel code.

— Task Management: OmpSs provides advanced mechanisms for managing parallel tasks, enabling developers to express parallelism flexibly across different processors, including CPUs and GPUs.

— Support for Heterogeneous Architectures: By optimizing the use of available resources across diverse architectures, OmpSs offers increased flexibility to exploit the performance of systems composed of different types of processors.

— Performance and Portability: Like OpenMP, OmpSs aims to offer high scalability and portability, enabling parallel applications to run efficiently on a variety of hardware platforms.

Although OpenMP or OmpSs offers powerful features for parallel programming on various architectures, the correctness of the code depends only on the programmers. Some work has been investigated in [Roy18] to tackle compiler analysis, but there is still a need to analyze code specifications.

Message passing

Common HPC programming models use the Single Program, Multiple Data (SPMD) paradigm, a subset of the MIMD category in Flynn's taxonomy [Fly72]. In the SPMD model, a single program runs concurrently across multiple data elements. Each machine or node in the architecture executes the same program on different data, enabling parallel processing and improved efficiency. As shown in Figure 1.9, the program is distributed among processing nodes, with one process per node, each assigned a unique rank. Most distributed systems use message-passing techniques to communicate between machines. Message passing refers to a programming paradigm in which computation units, often called processes or actors, communicate by sending and receiving messages. In this paradigm, each process operates independently and interacts with other processes solely through message exchange. When a process needs to communicate with another, it sends a message containing data or instructions to the recipient process, which then processes the message and may respond accordingly.

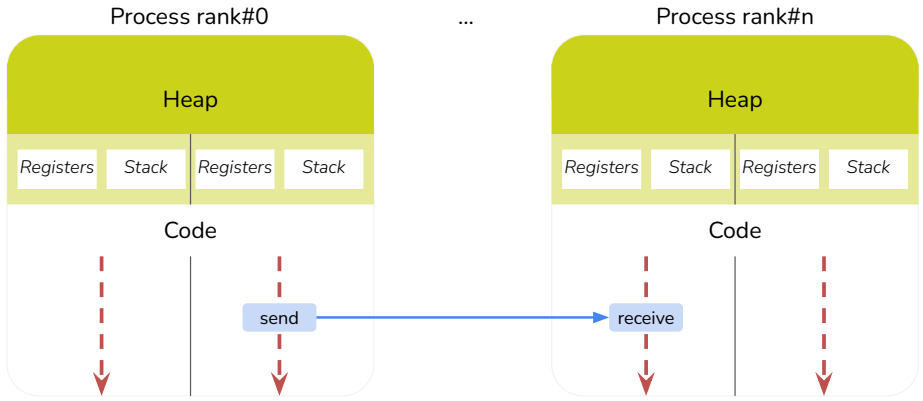

Figure 1.9 – Illustration of the message-passing procedure on a SPMD architecture

MPI introduced in [For94] is a standardized and widely used communication protocol and programming model for parallel and distributed computing based on message passing. Here are its key features regarding DCS systems:

— Memory Management: MPI allows each process in a distributed system to have its own address space, known as distributed memory. Memory management in MPI is explicit, with each process managing its own memory independently. Processes can access only their local memory directly, but they can exchange data with other processes through message passing.

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
 int rank, size;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 int data;
 if (rank == 0) {
  data = 100;
  MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send data to process1
 } else if (rank == 1) {
  MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Receive data from process0
 }

 MPI_Finalize();
 return 0;
}
```

Figure 1.10 – Simple example distributing computation over two machines using MPI

— Communication: MPI provides a set of communication primitives for sending and receiving messages between processes. It supports both point-to-point communication (sending messages between pairs of processes) and collective communication (broadcasting, scattering, gathering, etc., involving multiple processes). Communication operations can be synchronous or asynchronous, offering flexibility in managing communication overhead and latency.

— Synchronization: MPI supports various synchronization mechanisms to coordinate the execution of processes in a distributed system. Synchronization primitives such as barriers allow processes to synchronize their execution, ensuring that certain operations are complete before others begin. MPI also provides mechanisms for handling dependencies between processes, enabling synchronization based on data availability or computation progress.

— Scalability: MPI is designed to scale efficiently to large-scale parallel and distributed computing systems. It supports dynamic process management, allowing processes to be created, terminated, and dynamically assigned to resources at runtime. MPI implementations are optimized for scalability, with efficient support for communication and synchronization across thousands or even millions of processes.

Figure 1.10 shows a simplified code snippet employing MPI to assign a communication rank to the available process and to communicate through message-passing between two

processes.

1.2.3 Acceleration Programming frameworks and libraries

Acceleration Programming frameworks and libraries have emerged to leverage the computational capabilities of specialized hardware accelerators. Acceleration Programming involves preparing and transferring computation workloads to specific hardware resources. Such accelerators can include GPUs, FPGAs, Data Processing Units (DPUs), or any other kind of Application-Specific Integrated Circuits (ASICs). The typical accelerator processing flow relies on a host/device role, with the host preparing and offloading the workload to the device, and retrieving the result at the end, as illustrated in Figure 1.11. This section explores the prominent *Acceleration Programming frameworks and libraries*: Compute Unified Device Architecture (CUDA), OpenCL and SYstem-wide Compute Language (SYCL).

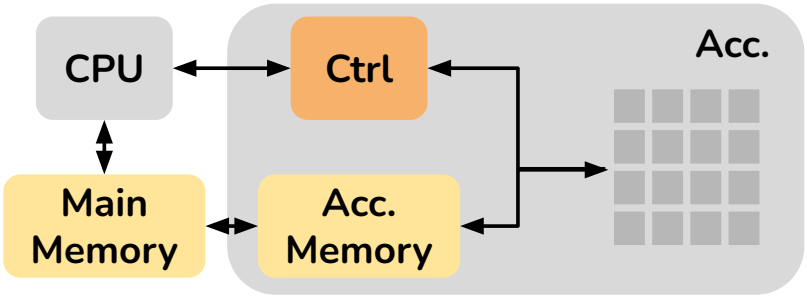

Figure 1.11 – Illustration of the accelerator processing flow

CUDA is a parallel API specifically developed by Nvidia [2] for their General-Purpose Computing on Graphics Processing Unit (GPGPU). Widely adopted in scientific and data-intensive applications, CUDA provides a framework for developers to offload data parallel tasks to the GPU, unlocking massive parallelism restricted to Single Instruction Multiple Threads (SIMT) compatible computations. Key features of CUDA include:

— GPU Offloading: CUDA allows developers to offload parallel portions of their code to the GPU, harnessing the massively parallel architecture for accelerated computation.

— Thread Hierarchy: CUDA introduces a hierarchical thread structure with grids,

2. `https://developer.nvidia.com/cuda-toolkit`

blocks, and threads, providing fine-grained control over parallel execution and resource utilization.

— Unified Memory: CUDA supports Unified Memory, allowing seamless memory sharing between the CPU and GPU, simplifying data management and enhancing programming flexibility.

As mentioned in [Gac20], in 2007, a High-Performance Image Reconstruction (HPIR) workshop was organized to compare the performance of different acceleration architectures. Over the years, Nvidia GPUs have dominated this area of research. However, a common criticism of the enthusiasm for Nvidia GPUs is that CUDA API is designed exclusively for Nvidia architectures. In response, the Khronos consortium proposed the OpenCL open programming standard [Khr11], also based on C syntax, to target all types of accelerator architectures.

OpenCL is an open standard for cross-platform, parallel programming of heterogeneous systems, encompassing CPUs, GPUs, FPGAs, and other accelerators. Developed by the Khronos Group [3], OpenCL provides a flexible and vendor-neutral framework for achieving parallelism on diverse hardware architectures. Key features of OpenCL include:

— Heterogeneous Computing: OpenCL enables developers to create programs that execute across a variety of heterogeneous devices, promoting versatility and adaptability.

— Abstraction of Hardware Architecture: OpenCL abstracts the underlying hardware architecture, allowing developers to write code that can run efficiently on different accelerators without extensive modification.

— Memory Management: OpenCL provides explicit memory management, allowing developers to allocate, transfer, and manage memory across different computing devices.

While flexible has historically been less portable and more architecture-specificas noted in [Nez09], OpenCL 3.0 introduces several improvements that enhance portability across platforms. OpenCL 3.0 allows selective adoption of features, facilitating more flexible development across different architectures. However, it still operates at a lower level of abstraction compared to OpenMP, and challenges remain in achieving seamless cross-platform compatibility, particularly when compared to more high-level frameworks like OpenMP.

3. `https://www.khronos.org/opencl/`

In addition to these acceleration programming libraries, there are also directives like Open Accelerators (OpenACC) [Wie+12] and OpenMP [CJV07], as well as Domain-Specific Language (DSL) such as Matlab or TensorFlow [Dev22], which utilize these low-level libraries. OpenACC is aimed at being integrated into OpenMP, a process that has already begun with OpenMP 4.0, and continues to evolve in subsequent versions, as mentioned in [Mio22]. Its goal is to enable the offloading of computational workloads to accelerators with minimal modification to the source code. However, the ease of use of OpenACC often sacrifices architecture-specific optimizations, resulting in a performance gap. On the other hand, SYCL provides a higher-level programming model built on top of OpenCL, offering a unified, cross-platform abstraction for parallel programming. As mentioned in [Gho12], the future of parallel processing is heavily influenced by the Nvidia CUDA platform. While facing challenges due to its limited compatibility with Nvidia GPUs, CUDA remains a prominent force. Competitors like AMD and Intel are entering the market, intensifying competition. Nonetheless, recent advancements suggest a promising outlook for CUDA, as GPUs gain popularity in scientific computing due to their robust processing capabilities and accessibility. Among developers, CUDA is emerging as the preferred choice for programming language.

All these APIs can offer hardware-specific optimizations but require manual intervention to specify these optimizations accurately. This human intervention can introduce errors and block process automation. In response to these limitations, the dataflow programming model, detailed in the next section, offers a promising solution.

1.2.4 Dataflow MoC for the SKA application specification

The SKA [4] project will comprise numerous antennas and dishes, with data routed to the on-site Central Signal Processor (CSP) for initial front-end processing. Subsequently, the data will be transmitted via dedicated long-distance optical fiber links to the SDPs, where it will be converted into data and images suitable for radio astronomers. Given the nature of the SKA project, which involves a massive flow of data requiring efficient, it is perfectly suited to being modeled as a dataflow. Moreover, the dataflow MoC has already been employed for modeling the SDP in [Mio+20] and a generic version of the imaging pipeline has been proposed in [Wan+24], demonstrating its applicability to complex data processing tasks within the SKA project.

4. `https://www.skatelescope.org/`

Kahn Process Network and Dataflow Process Network: The Dataflow MoC represents a paradigm where computation is modeled as a directed graph of actors, each capable of executing independently. Kahn Process Network (KPN) [Kah74] and Dataflow Process Network (DPN) [LP95] are prominent manifestations of this model. In KPN, illustrated in Figure 1.12, actors communicate through unbounded First In First Out (FIFO) channels, allowing for dynamic scheduling of tasks. DPN extends this concept by introducing bounded FIFO channels, providing a more realistic representation of physical communication channels. KPN ensures deterministic behavior because the same inputs will always yield the same outputs regardless of the execution order. In contrast, DPN can exhibit non-deterministic behavior due to the bounded nature of channels, where the timing and order of execution can affect the output. Both KPN and DPN serve as foundational frameworks for expressing and analyzing concurrency in dataflow systems.

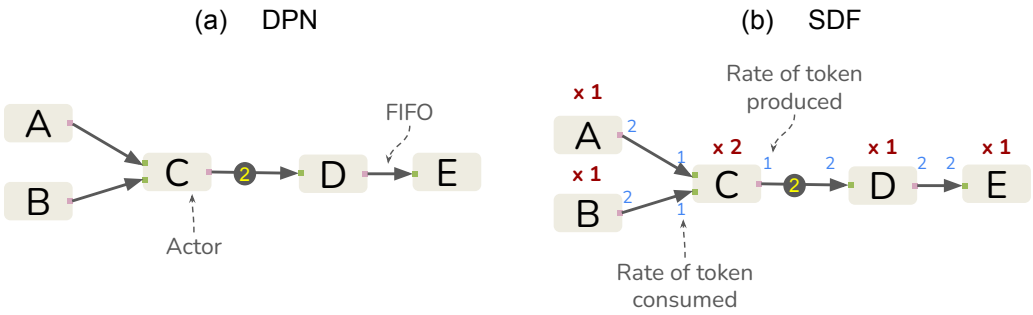

Figure 1.12 – Illustration of the (a) DPN and (b) Synchronous Dataflow (SDF) MoCs

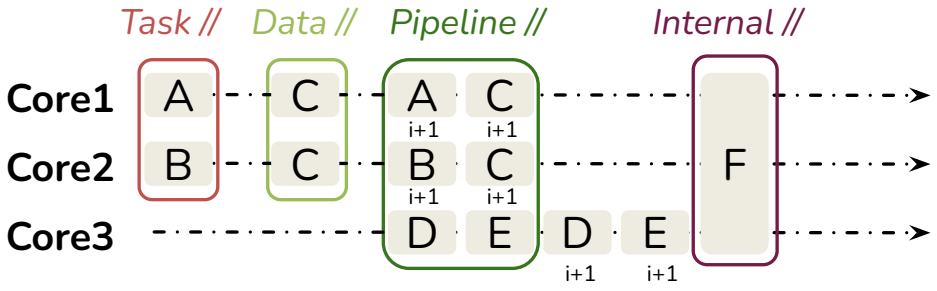

Figure 1.13 – Illustration of the different types of parallelism from the SDF graph in Figure 1.12

SDF MoC is a specialized subset of the dataflow MoC that introduces a regular and

deterministic structure [LM87b]. In SDF, actors have statically defined firing rates, and communication occurs synchronously. This deterministic nature simplifies the analysis of system behavior, enabling optimizations such as static scheduling. While SDF imposes constraints, it provides advantages in terms of predictability and ease of analysis, making it applicable in real-time and HPC systems.

One of the advantages of SDF MoCs is their expressiveness of parallelism which is particularly appealing for parallel and distributed computing, aligning with the scalable and parallel nature of modern HPC architectures. Computational tasks execute concurrently as data becomes available, enabling efficient utilization of resources. Parallelism in SDF can manifest in various forms, including task-level parallelism, data parallelism, pipeline parallelism, and internal parallelism [Zho+13].

— **Task** parallelism is expressed by two actors belonging to parallel data-paths like actors A and B from the dataflow graph in Figure 1.12 and the mapping representation in Figure 1.13. As there is no data-path between these actors, they can be fired at the same time.

— **Data** parallelism is expressed when several firings of a single actor are independent of each other. If enough data tokens are present in the input FIFO, then several firings can be executed concurrently. An example is the actor C in Figure 1.13 which can be executed 2 times when A and B are executed 1 time.

— **Pipelining** a dataflow graph consists of dividing a graph into pipeline stages, with an explicit delay used to separate stages. A delay, in dataflow graph, correspond to a number of data tokens present in the FIFO when the described application is initialized. During the first iteration of the graph, the application runs the first stage and retains the final token value, which is then used during the second iteration as the initial token for the second stage, and so on.

— **Internal** parallelism occurs when an actor utilizes parallelism within itself, often employing multiple PEs. This internal parallelism can be facilitated directly by the host language or through a hierarchical actor whose behavior is represented as a dataflow graph [Mio22]. In the example graph depicted in Figure 1.13, actor F is presumed to be a parallel actor executed on three cores.

In [Des14], a classification of properties of the dataflow MoC is proposed. These properties are split into two sets: objectively measurable properties and those that cannot be objectively measured. Objectively measurable properties include:

— *Decidability*: This refers to the ability to prove statically, at compile time, that an application described with the MoC will execute within bounded memory and without deadlock.

— *Determinism*: Applications modeled with this MoC solely depend on the data flowing within the application and not on external factors [LP95].

— *Compositionality*: It allows the MoC to be analyzable independently of the internal specifications of the actors composing an application graph [Tri+13].

— *Reconfigurability*: In a reconfigurable dataflow MoC, the firing rules of actors can change during the execution, but it should only happen at quiescent points to maintain predictability or partial determinism [NL04].

Properties not objectively measurable, yet useful, include:

— *Predictability*: It encompasses all behaviors related to changes in the firing rules of the MoC. Predictability can vary depending on how often or how the firing rules change.

— *Expressiveness*: This reflects the range of applications that can be represented with the MoC. The ability to model conditional structures, loops, and dynamic behavior determines its expressiveness. Dynamic MoCs are generally more expressive than static MoCs due to their ability to handle dynamic behavior.

In SDF, two critical properties, consistency, and liveness, ensure the efficient execution and prevention of deadlocks.

— *Consistency*: This property guarantees that no data token will indefinitely accumulate in any FIFO of the graph. Consistency is determined through the analysis of the topology matrix Γ associated with an SDF graph. The graph is consistent if and only if rank(Γ) = |A| - 1, with |A| being the number of connected actors in the graph. The Repetition Vector (RV), denoted $\mathbf{q}$, is the smallest non-zero integer vector verifying $\Gamma \times \mathbf{q} = 0$, computed through algorithms [BML96].

— *Liveness*: Liveness ensures that the graph can run indefinitely without any deadlocks. A live graph has sufficient initial data tokens to execute a full iteration, and each graph iteration starts with the same number of initial data tokens. Approaches like Symbolic Execution (SE) exist for verifying liveness [Bil+96], but they are not efficient for large graphs. Research efforts, explore mathematical approaches based on the analysis of strongly connected components to make liveness analysis faster and more efficient [Gha+06].

Generalization of the SDF MoC

This section introduces three generalizations of the SDF MoC that are later explored in this thesis. These extensions, namely the Interface Based Synchronous Dataflow (IB-SDF) MoC [PBR09], Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC [Des+13], and State-Aware Parameterized and Interfaced DataFlow (SPiDF) MoC, aim to provide more faithful modeling of applications by addressing specific aspects of complexity and flexibility. Illustrated in Figure 1.14, each model is defined as follows:

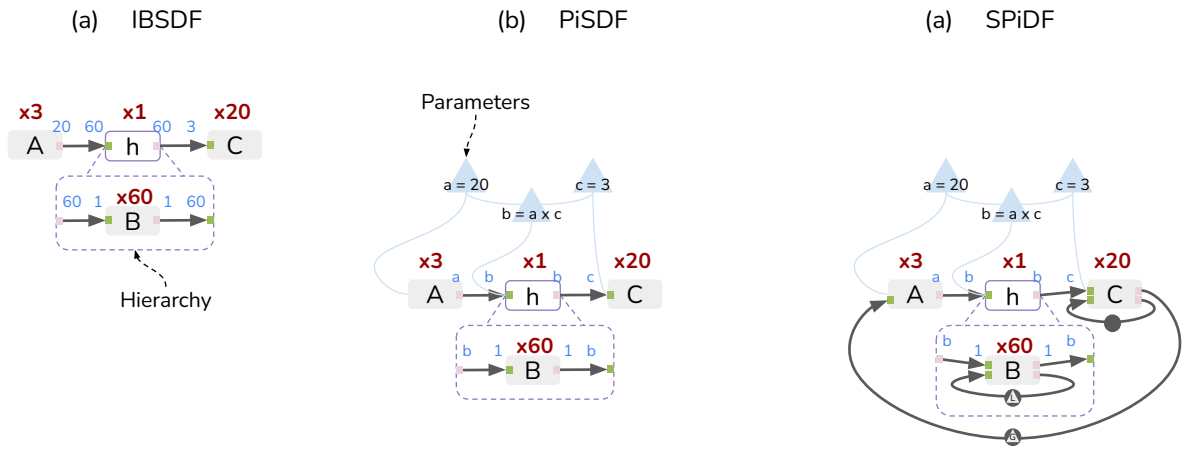

Figure 1.14 – Illustration of some generalizations of an SDF graph

— **IBSDF** MoC extends the SDF MoC, as described in [PBR09]. IBSDF, illustrated in Figure 1.14(a), allows for the definition of actor behavior using hierarchical subgraphs, introducing compositionality through the incorporation of data interfaces and specific execution rules for subgraphs. In an IBSDF graph $G = (A, F, I)$, where A represents the set of actors, F represents the set of Fifos and I represents the set of interfaces, each interface ($I = (I_{\text{in}}, I_{\text{out}})$) separates the internal definition of a subgraph G from the external behavior of the associated hierarchical actor HG. Source interfaces (I_{in}) receive data tokens from the corresponding input ports of the hierarchical actor, functioning as circular buffers if the subgraph consumes more tokens than provided. Sink interfaces (I_{out}) transmit only the last produced tokens, discarding any excess. This ensures the compositional properties of IBSDF. Moreover, interfaces in a subgraph are write-locked and read-locked for source and sink interfaces, respectively, throughout subgraph execution. Notably, delays in hierar-

chical IBSDF graphs have no side effects. These formal aspects collectively establish the compositional nature of the IBSDF MoC.

— **PiSDF**, introduced in [Des+13], extends the SDF and IBSDF MoCs with hierarchical and dynamically reconfigurable features. Similar to IBSDF, in a PiSDF graph, a hierarchical actor is defined by a subgraph. Formally, a PiSDF graph $G = (A, F, I, \Pi, \Delta)$ contains actors A, FIFO F, hierarchical interfaces I, parameters Π, and parameter dependencies Δ. The hierarchical interfaces of the PiSDF MoC inherit directly from the IBSDF MoC, ensuring compositional behavior where the internal specification of actors does not affect graph analysis. Parameters $\pi \in \Pi$ are associated with values $v \in \mathbb{N}$. Parameters can be statically defined, derived from others, or set dynamically by configure actors at runtime. A dynamically set parameter, termed a configurable parameter, is set by a configured actor once per graph iteration. Dependencies $\delta \in \Delta$ propagate parameter values between configuration input/output ports of actors and parameters. In hierarchical actors, configuration ports, also known as configuration interfaces, are static parameters within the associated subgraph. The combination of parameters Π and dependencies Δ forms a parameter dependency tree $T = (\Pi, \Delta)$. Unlike dataflow graph FIFO, parameter dependencies in PiSDF provide instantaneous access to connected parameter dependencies once their values are set.

— **SPiDF**, also presented as the State-Aware Parameterized and Interfaced Synchronous Dataflow (SA-PiSDF) and introduced in [Arr+18], extends the PiSDF with state-awareness. The model introduces the persistence of a graph state with three types of delays: Local Delay (LD), Locally Persistent Delay (LPD), Globally Persistent Delay (GPD).

  — The LD delayed data tokens are preserved within the scope of a unique graph iteration. Indeed, LD are initialized by an optional *Setter* actor at the beginning of each iteration and the final token is retrieved by a *Getter* actor at the end of the iteration. As illustrated in Figure 1.14 (c) an LD initializes the FIFO unrolling actor C at each graph iteration.

  — The LPD tokens exhibit persistence specifically within the scope of a hierarchical actor. This persistence significantly influences the firing sequence of actors located at the same hierarchical level. It establishes a precedence relationship, determining the order in which hierarchical actors fire, directly impacted by the presence of LPD tokens. As illustrated in Figure 1.14 (c) an LPD initializes

the FIFO unrolling actor B at each subgraph execution.

— The GPD are initialized at the beginning of the program and the value of the delay at the end of a graph iteration is used as the initial value for the next iteration. As illustrated in Figure 1.14 (c) a GPD initialize the FIFO unrolling actor A at the beginning of the program.

These properties allow greater model flexibility and more reliable application representation.

Specialization of the SDF MoC

The purpose of specializing dataflow graphs, as discussed in [Der19], is to introduce constraints or restrictions that tailor the behavior of the graphs to specific requirements or characteristics. In particular, the author presents three specializations of the SDF MoC: Single rate Synchronous Dataflow (SrSDF), Homogeneous Synchronous Dataflow (HSDF), and Directed Acyclic Graph (DAG). Each specialization imposes unique constraints on the consumption and production rates of SDF actors or on the structure of the SDF graph itself. For instance, the HSDF and SrSDF models restrict consumption and production rates, while the DAG model restricts the graph structure to be acyclic. These specializations, illustrated in Figure 1.15, facilitate the analysis and design of dataflow systems by refining the model to match specific application requirements or analysis techniques.

— **SrSDF** model: An SDF graph is classified as a SrSDF model if, for each Fifo queue (f_i), the production rate ($\mathrm{r}_{\mathrm{prod}}$) of its source actor ($A_{\mathrm{source}}$) is equal to the consumption rate ($\mathrm{r}_{\mathrm{cons}}$) of its target actor ($A_{\mathrm{target}}$), i.e., $\mathrm{r}_{\mathrm{prod}}(A_{\mathrm{source}}) = \mathrm{r}_{\mathrm{cons}}(A_{\mathrm{target}})$. Similar to the HSDF transformation, the conversion of an SDF graph to an equivalent SrSDF graph involves duplicating actors based on their repetition factor. However, the SrSDF graph conversion utilizes fewer FIFO queues to connect the duplicated actors. Specifically, if multiple FIFO queues connect the same pair of actors in the HSDF graph, they are merged into a single FIFO queue. The production and consumption rates of this merged queue are equal to the number of merged FIFO queues. Mathematically, let $f_1, f_2, ..., f_n$ denote the merged FIFO queues connecting actors a_i and a_j in the HSDF graph, where n is the number of merged FIFO queues. Then, in the resulting SrSDF graph, a single FIFO queue f_{merged} connects a_i and a_j, with production rate $\mathrm{r}_{\mathrm{prod}}(f_{\mathrm{merged}}) = \mathrm{r}_{\mathrm{cons}}(f_{\mathrm{merged}}) = n$. This process minimizes the number of FIFO queues required while preserving the behavior of the original SDF graph. Figure 1.15 (c) illustrates this process.

— **HSDF** model: An SDF graph is categorized as a HSDF model if all actors within the graph have consumption and production rates equal to 1. The HSDF model can also be viewed as a special case of the SrSDF graph, i.e., $\mathrm{r}_{\mathrm{prod}}(A_{\mathrm{source}}) = \mathrm{r}_{\mathrm{cons}}(A_{\mathrm{target}}) = 1$. The conversion of an SDF graph to an HSDF graph, known as expansion, is a common transformation in dataflow analysis. It involves duplicating each SDF actor based on its repetition factor and token ordering. If an SDF actor a produces p data tokens on a FIFO queue f, f is duplicated p times for each instance of a in the resulting HSDF graph. Consequently, every actor in the HSDF graph has a repetition factor of 1, with consumption and production rates set to 1 on all FIFO queues. The conversion algorithm is described in [BML96], and Figure 1.15 (b) illustrates this process.

— **DAG** model: The DAG MoC is a specialized form of the SDF MoC, wherein cyclic data paths are prohibited. Given a consistent and schedulable SDF graph, its transformation into an equivalent DAG entails replacing FIFOs containing delays with a pair of special actors: Save and Restore. To ensure schedulability, every cyclic data path must include at least one initial data token, inherently breaking all cycles when replacing FIFOs with delays. The Save actor is tasked with backing up several data tokens equivalent to the delay count of the FIFO. Executed before the end of the iteration, it preserves tokens for consumption in the subsequent iteration. Conversely, the Restore actor retrieves the backed-up tokens and forwards them to the appropriate consumer. A Save actor must always precede its corresponding Restore actor in scheduling. This transformation imbues the SDF graph with a hidden causality property, as the transformation process preserves data flow consistency. Mathematically, let D denote the set of FIFOs containing delays in the SDF graph, and n_i represent the number of delays in FIFO $i \in D$. Then, the Save actor S_i saves n_i data tokens, and the Restore actor R_i retrieves and forwards these tokens. The transformation effectively ensures acyclic behavior while preserving data flow consistency.

These specializations serve various purposes, one of which is to provide a foundation for resource allocation based on dataflow principles detailed next chapter.

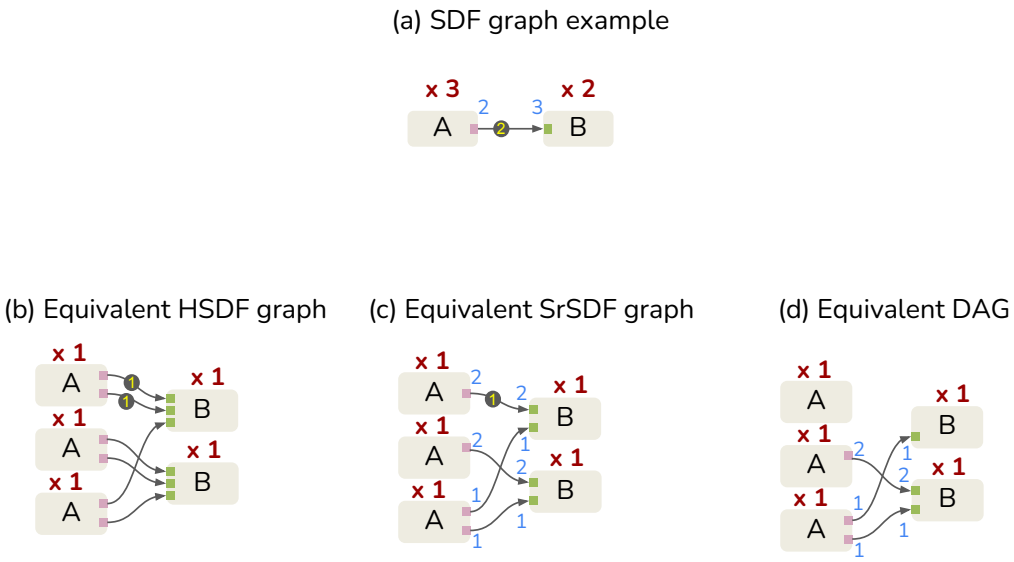

Figure 1.15 – Illustration of some of the specializations of an SDF graph

Conclusion

In this chapter, we have explored hardware/software co-design requirements for the SKA using the AAA methodology. This approach enables automatic architectural exploration through graph representation, separating application and architecture design into independent models (MoC and MoA). We also detail how these models can represent both embedded and HPC systems, and provide an overview of different programming MoCs, including shared memory systems, distributed multiprocessing, acceleration programming frameworks, and SKA application specification through dataflow MoC.

All of these programming models, including data flow models, are designed to abstract away the details of the underlying architecture, enabling portability across different systems. However, this independence can lead to performance trade-offs, as it may limit the ability to optimize for specific hardware features, such as cache hierarchies or vectorization capabilities, which are crucial for maximizing performance in HPC environments. In contrast, APIs can offer finely tuned optimizations, but require manual intervention to specify these optimizations accurately. The contribution of this thesis aims to bridge this gap by providing a structured approach that leverages the strengths of both portability and optimization potential.

The next chapter will address the challenges of resource allocation, an essential aspect for rapid prototyping of HPC architectures. It will present existing methods and tools,

focusing on dataflow optimization to improve the execution of complex programs on HPC architectures.

Chapter 2

Resource Allocation Challenge

Introduction

AAA is a rapid prototyping methodology. By creating early versions of not yet built systems, developers can explore design choices, evaluate ideas, and facilitate the decision-making process. As depicted in [Has00], the motivations behind rapid prototyping in parallel programming are multifaceted, some of which are as follows. Firstly, reducing execution time is a primary goal. Parallelism allows dividing tasks into smaller units that can be executed concurrently. By identifying efficient parallelization strategies early on, we can achieve faster execution and improved performance. Secondly, increasing fault tolerance is essential in distributed and parallel systems. Prototyping enables us to experiment with fault-tolerant mechanisms, assess their effectiveness, and make informed decisions about error handling and recovery strategies. Thirdly, explicitly exploiting inherent parallelism is a key motivation. Some applications inherently exhibit parallelism, such as scientific simulations or data processing tasks. Rapid prototyping allows us to explore how to harness this inherent parallelism effectively. Lastly, rapid prototyping facilitates early evaluation and correct design choices. Using different approaches, we can evaluate alternative design decisions, evaluate trade-offs, and ensure that our chosen parallelization approach is in line with the requirements of the system.

In this chapter, Section 2.1 outlines the typical design flow for rapid prototyping and exposes the resource allocation challenges. An overview of existing research in the field of rapid prototyping is presented in Section 2.2. Finally, Section 2.3 presents the state-of-the-art methods for simplifying and optimizing the dataflow scheduling.

2.1 Rapid Prototyping Design Flow

In [Des14], the author states that the typical rapid prototyping design flow can indeed be segmented into three distinct parts: *Developer Inputs*, *Rapid Prototyping,* and *Legacy*

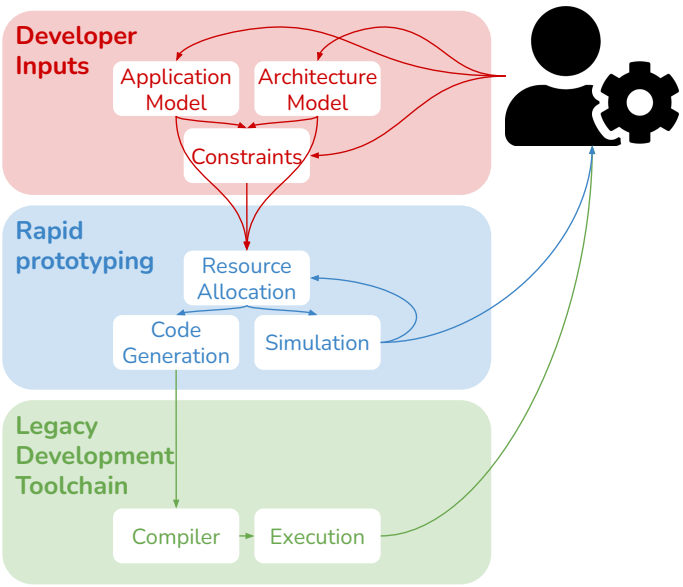

Figure 2.1 – Overview of a rapid prototyping design flow

Development Toolchain as illustrated in Figure 2.1. Each phase plays a crucial role in the iterative process of designing, refining, and deploying applications on any type of system: embedded or even HPC.

1. **Developer Inputs**: In the context of co-design, where systems comprise both hardware and software components, developer inputs encompass high-level models that specify crucial properties of the system. These inputs typically include a model of the application (as detailed in Section 1.2), a model of the targeted architecture (as discussed in Section 1.1), and a set of constraints for deploying the application on the architecture. Following the AAA methodology, the segregation of these inputs within the design flow ensures their independence, thereby facilitating the deployment of an application on multiple architectures or the use of a single architecture to deploy multiple applications.
2. **Rapid Prototyping**: Once the developer inputs are defined, rapid prototyping begins with the creation of initial prototypes of the parallel algorithm or system. This phase involves quickly implementing different parallelization strategies and architectural choices to explore the design space. Developers may utilize high-level parallel programming frameworks, domain-specific languages, or prototyping tools to expedite the development process. The focus is on generating functional prototypes

that can be evaluated for correctness, performance, and scalability. Through rapid iteration, developers assess various design alternatives, experiment with different parallelization techniques, and gather feedback to inform further refinements.

3. **Legacy Development Toolchain**: The legacy development toolchain plays a crucial role in supporting rapid prototyping efforts by providing essential infrastructure, tools, and resources. This includes compilers, debuggers, profilers, and simulation environments tailored for the target architecture. Developers leverage these tools to streamline the development process, identify performance bottlenecks, and validate their prototypes against performance targets.

In Figure 2.1, we present an outline of the typical rapid prototyping design flow, wherein developers or tools utilize simulation and execution feedback to enhance the quality of prototypes. The primary objective of this thesis is to enhance the rapid prototyping process to align more effectively with HPC constraints. The subsequent sections delve into the complexities associated with various aspects of this phase and examine existing solutions documented in the literature.

2.1.1 Resource allocation

To generate a multi-core, or more broadly a multi-PE implementation, it is necessary to allocate resources. This process comes in two forms: static and dynamic. Static resource allocation involves predefining and allocating resources based on fixed criteria, during compile time, while dynamic allocation adjusts resource distribution in real-time, or execution time, according to current demand and workload. Static allocation offers simplicity but may lead to inefficient resource utilization, whereas dynamic allocation improves efficiency by allocating resources based on demand, albeit introducing complexity and potential overhead. The contribution of this thesis concentrates on static allocation, aiming for reliable prediction in resource allocation processes. The typical static resource allocation process involves four primary tasks, namely *extraction*, *mapping*, *scheduling*, and *timing* as exposed in [Arr20], detailed in [LH89] and illustrated in Figure 2.2.

1. **Extraction**: The initial step in the multi-PE scheduling process involves extracting the various parallel and sequential executable tasks from a given application. This phase identifies and isolates tasks to be executed in parallel or sequentially.
2. **Mapping**: Following task extraction, the mapping phase assigns each extracted task to a specific PE. This assignment is typically based on heuristics considering

factors such as communication costs and energy efficiency. Communication costs are often calculated using task predecessors and successors.

3. **Scheduling**: Once all tasks are mapped, the scheduling phase entails determining the execution order of tasks based on their dependencies. In applications represented by DAGs, data dependencies dictate a strict ordering of tasks within the same data path.
4. **Timing**: The final phase involves assigning start times for each task. This step is typically handled by the runtime manager of the application or the underlying host operating system. Start times are determined based on various factors, including the current processing load of the platform and synchronization points such as data availability.

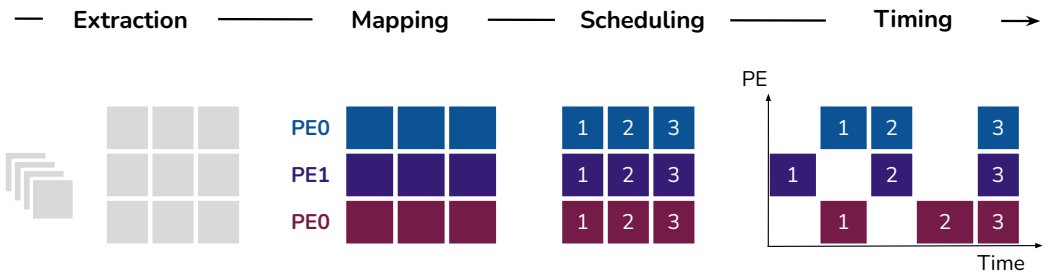

Figure 2.2 – Overview of the typical multi-PE resource allocation flow

In static dataflow resource allocation, the extraction step is decomposed into two steps, namely flattening and Single rate Directed Acyclic Graphs (SrDAG) transformation which are subsequently detailed:

1. **Flattening** transformation: Flattening the hierarchy of an IBSDF, a PiSDF or a SPiDF graph entails replacing hierarchical actors with their subgraphs and, if necessary, substituting data interfaces with actors that implement specialized behavior. In accordance with [PBR09], Broadcast (Brd) actors replace the functionality of data input interfaces that generate multiple identical data tokens. Conversely, Roundbuffer (Rnd) actors replace the functionality of data output interfaces that selectively transmit only the latest data tokens received to their respective output FIFO. Formally, let $G = (A, F, I)$ represent the hierarchical SDF graph, where A is the set of actors, F is the set of FIFOs, and I is the set of interfaces. Upon flattening, hierarchical actors H are replaced by their corresponding subgraphs, and data

interfaces requiring special behavior are substituted by Brd or Rnd actors. This process ensures a flattened representation of the original SDF graph while preserving the special functionalities of the interfaces.

2. **SrDAG** transformation: The SrDAG MoC is a specialization form of the SDF MoC wherein the production and consumption values on each FIFO are equal and cyclic data paths are prohibited. Given a consistent, schedulable, and flattened graph, its transformation into an equivalent SrDAG entails SrSDF conversion followed by the DAG conversion. The SrDAG transformation is used to reveal parallelism and emphasizes the interdependencies between the actors later used to compute the Critical Path (CP) as an example.

   Example 4. Given the flattened SDF graph depicted in Figure 2.3 (b), the computed RV is $\mathbf{q} = (1, 4, 2)^T$. This RV indicate the number of execution associated to each actor at each graph iteration. For example, actor A is executed once, actor B is executed four times, and actor C is executed twice. To derive the equivalent SrDAG shown in Figure 2.3 (a), actor A is duplicated once, B is duplicated four times, and C is duplicated twice. Additionally, a pair of special actors, *Fork* and *Join*, are introduced. These actors are responsible for distributing and gathering tokens to and from the respective receiving actors, ensuring a single rate on each FIFO.

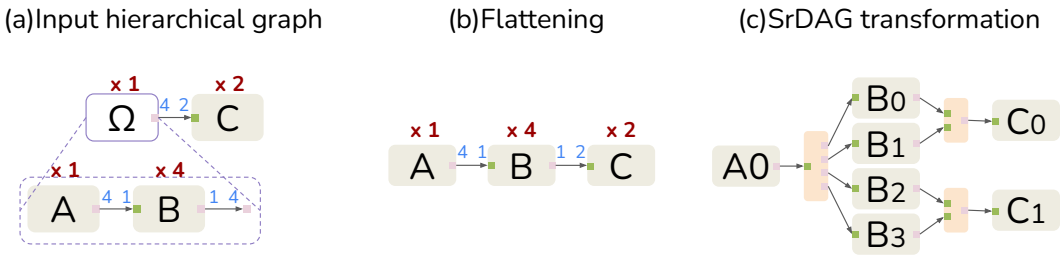

Figure 2.3 – Typical flattening process: 3 SDF actors turn into 10 SrDAG actors

One significant challenge lies in efficiently allocating resources on a target architecture within a decent compile time, irrespective of their complexity. The complexity of scheduling algorithms is largely influenced by the number of vertices in the graph. For instance, a basic list scheduling algorithm, such as the one described in [Sin07], exhibits a complexity of $O(P(V + E))$, where V and E represent the vertex and edge sets, respectively, of the SrDAG, and P denotes the number of PEs. Regarding the SDF, the complexity amounts

to $O(A\log(A) + P(A + E))$, as highlighted in [PBL95]. This complexity can lead to excessively long scheduling times. Given that mapping opportunities are constrained by the number of PEs available in the architecture, exposing more parallelism than the number of PEs is unnecessary and time-consuming.

Example 5. Let's consider a machine learning application, specifically the Squeezenet neural network. The SDF model of this network comprises 70 actors. Upon transformation into a SrDAG, the network expands to 5452 actors. When mapping this application onto an architecture consisting of 8 PEs using greedy algorithms, we need to evaluate 8 mapping choices. However, given that the degree of parallelism is up to 1000, it results in a needlessly fine granularity.

2.1.2 Code generation

Code generation is the process by which high-level specifications or domain-specific languages are translated into executable code that is optimized for specific target architectures. Leveraging sophisticated code generation techniques, including template-based approaches, domain-specific optimizations, and runtime code synthesis, these approaches leverage the translation process while accommodating hardware-specific constraints and performance considerations. In [Pel+13], the author highlights two primary options: generating static execution when an OS is not required, or utilizing an OS capable of dynamic scheduling.

— **Static Code Generation and Execution**: Static code generation is characterized by the generation of code whose structure and computations remain relatively constant throughout program execution. In this paradigm, the algorithms, data structures, and computational logic are predefined and do not vary significantly based on runtime conditions or input data. Static code generation is often well-suited for scenarios where the computational requirements are known beforehand or where the algorithms exhibit stable behavior across different input datasets or problem instances. Examples of static code generation include the compilation of standard libraries, code templates, and pre-optimized routines commonly employed in HPC applications.

— **Dynamic Code Generation and Execution**: Dynamic code generation might refer to adapting code or algorithms based on changing conditions or requirements over an extended period. In SKA context, dynamic code generation could involve adapt-

ing communication protocols, signal processing, or resource allocation strategies based on varying network conditions, traffic loads, or evolving technology standards. Dynamic code generation allows for adaptability to changing scenarios, improved resource utilization, and the ability to incorporate advancements in communication technologies over time.

2.1.3 Simulation

In the rapid prototyping context, the simulation process, as highlighted by Bernreuther et al. [Ber+05], serves as a powerful bridge between theoretical models and real-world phenomena. Its adoption is motivated by the complexity of physical processes and technical systems, making simulation a necessary tool for exploring hardware and software behavior, and more. Additionally, practical constraints like cost and safety often make physical experiments impractical, leading researchers to rely on simulation for studying various phenomena. Simulation of parallel systems is crucial for understanding and optimizing the performance of multi-core processors, distributed networks, and other complex systems that operate concurrently. By simulating these parallel systems, researchers can identify bottlenecks, improve efficiency, and ensure reliable performance under various conditions. Overall, the two most important characteristics of HPC simulators are their accuracy and their scalability. Accuracy refers to the fidelity of the simulator predictions compared to real-world observations. As mentioned in [RK08], the timing accuracy of simulators can be categorized into three main types:

— **Functional**: These simulators only predict the results produced by the simulated system by executing the same computation. Functional simulation is useful for quickly verifying the correct behavior of designed systems.

— **Approximately timed**: Approximately-timed simulators predict system performance based on simplified models of the architecture [Pel+09a]. While they can be generated quickly, the actual performance may slightly differ from the predicted performance.

— **Cycle accurate**: Cycle-accurate simulators predict the exact performance of the system. They are based on low-level models, making them slower but more precise. These simulators are often used when hardware is not available, and simulation results are crucial for proving real-time constraints compliance in safety-critical systems.

Similarly, these levels of accuracy extend to other simulated metrics such as energy consumption or memory footprint, among others, where achieving higher accuracy may result in longer simulation times. The following define key metrics and the importance of finding a tradeoff between at least latency, throughput, energy, memory, and cost.

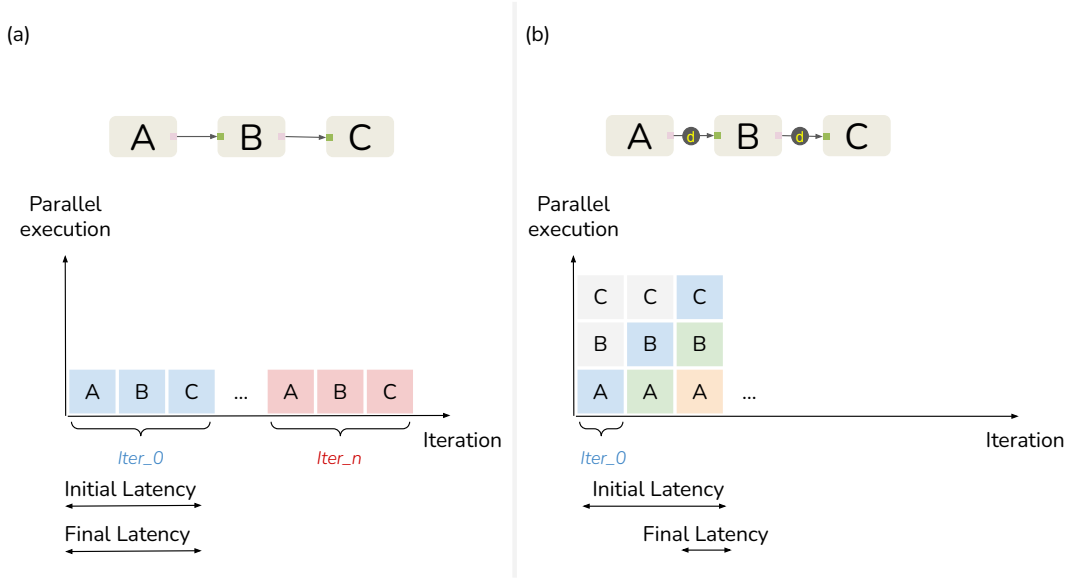

Figure 2.4 – Illustration of the (a) Fixed latency during iteration, and (b) Asymptotic latency during iteration. Square with the same color represent computation from the same graph iteration. Gray squares represent phases without useful computation that is when computation does not produce any output.

— **Latency**: refers to the elapsed time between the initiation of the first task and the completion of the last task.

Definition 10. Let T_{init} denote the initiation time of the first task, and T_{last} represent the completion time of the last task. The latency L is defined as the difference between T_{init} and T_{last} at the beginning of system execution.

$$L = T_{\text{last}} - T_{\text{init}}$$

Example 6. In a parallel processing system, if the first task begins at time $T_{\text{init}} = 0$ and the last task is completed at $T_{\text{last}} = 100$ seconds when the system achieves its maximum throughput, then the final latency L_{final} is 100 seconds.

— **Initial and Final Latency**: For applications designed to run indefinitely, the latency during the graph iteration of the DAG modeling the application can initially decrease up to a certain threshold, starting from an initial latency L_{init}, also referred to as *makespan* in [Hon20], and gradually decreasing to a final latency L_{final}, also referred to as Initiation Interval (II) duration in [Hon20]. This variation is attributed to the implementation of pipelines, which enhance system performance at the expense of an elevated initial latency. The concept of DAG pipelining, explained in Section 2, involves breaking dependencies in the DAG and dividing it into multiple concurrently executable stages. This process begins by setting an initial value for breaking dependencies during the first graph iteration, and this value is then propagated from the previous iteration. Graph iteration denotes a predetermined sequence of task execution that can be endlessly repeated without encountering a deadlock.

Example 7. Let's consider the DAG composed of tasks A, B, and C as illustrated in Figure 2.4 (a). Throughout each graph iteration, the initial and final latency remains constant and equals the sum of the elapsed time for each sequential task, from the initiation to completion, expressed as $L_{init} = L_{final} = L_A + L_B + L_C$. In the scenario where all task executions are equivalent, this results in $L_{init} = L_{final} = 3$ time units. Conversely, let's consider a pipelined graph consisting of actors A, B, and C, where dependency breaks are denoted with a d on the edge, as illustrated in Figure 2.4 (b). The first pipelined stage comprises actor A, the second with B, and the third with C. The initial latency represents the completion of all graph iterations until reaching the pipeline depth, calculated as $L_{init} = L_{iter_0} + L_{iter_1} + L_{iter_2} = 6$ time units. The final latency occurs once the process has reached the pipeline depth, and it is expressed as $L_{final} = 1$ time unit.

— **Throughput**: Refers to the amount of data processed or transferred per unit of time. It is commonly defined as follows:

Definition 11. Throughput Let D denote the amount of data processed or transferred, and T represents the time taken to process or transfer the data. The throughput R is calculated as the ratio of data D to time T.

$$R = \frac{D}{T}$$

Example 8. If a system processes 100 GB of data in 10 seconds, then the throughput

R is 10GB/s

Maximum throughput refers to the highest achievable data transfer rate under optimal conditions. It represents the peak performance of the system in terms of data movement. In a non-pipelined application graph, as illustrated in Figure 2.4 (a), the maximum throughput is reached from the first graph iteration. However, in a pipelined application graph, as illustrated in Figure 2.4 (b), maximum throughput is reached when the final latency is obtained at the expense of the initial latency.

— **Memory and Storage Requirements**: Memory requirements refer to the amount of Random Access Memory (RAM) required to run the application efficiently without problems. If an application is defined as a graph then the simulated memory can be defined as follows:

Definition 12. Let S_i represent the size of the i-th FIFO buffer in the application graph. The memory requirements M are the sum of the sizes of all individual FIFO buffers:

$$M = \sum S_i$$

Example 9. If a application graph presents 3 FIFO buffers with sizes $S_1 = 2$ GB, $S_2 = 1$ GB, and $S_3 = 3$ GB, then the memory requirements M are 2+1+3=6 GB.

Parallelizing an application typically requires more memory than its sequential counterpart. This is because data is often shared or distributed between different processing units. While memory optimization techniques such as Memory Exclusion Graphs (MEGs) [Des+15] help to minimize the number of buffer copies to just what is needed, memory limitations often act as a barrier to extensive parallelization. Storage, in contrast, refers to the non-volatile space needed to store intermediate and final data outputs, which is essential for applications with large-scale data processing requirements, such as SKA. The SKA project, dealing with petabytes or even exabytes of data, requires significant storage infrastructure, both for temporary buffer storage during computation and for long-term archival purposes. Efficient storage management is critical in ensuring that the massive data inflows do not overwhelm the system and that the necessary datasets are readily accessible for subsequent analysis. Data-intensive applications such as SKA must account for both read/write throughput and capacity when planning storage resources, as I/O bottlenecks can drastically affect overall performance.

— **Energy Consumption**: refers to the total amount of electrical energy required by the hardware components to perform computational task. In a multi-node multicore architecture context, energy consumption can be divided into two main components: node-level energy and link-level energy. Node-level energy is calculated by combining dynamic power (related to the computation) and static power (associated with idle power consumption). Link-level energy is determined by the transmission power, link distance, and link bandwidth.

Definition 13. Let E_{node} represent the node-level energy, and E_{link} represent the link-level energy. The total energy consumption E is the sum of E_{node} and E_{link}.

$$E = E_{\text{node}} + E_{\text{link}}$$

Example 10. If the node-level energy is 50J and the link-level energy is 20J, then the total energy consumption E is 50+20=70J

— **System Cost**: The system cost is calculated by summing the individual cost associated with nodes, cores (processing units within nodes), routers (which direct communication between nodes), and links (connexion for data transfer)These components are considered during the modeling of architectures and topologies. It gives a rough estimation the financial investment required to acquire, deploy, and maintain the HPC system.

2.2 Existing work on rapid prototyping

In this section, we present the related work on rapid prototyping and simulation tools.

2.2.1 Rapid Prototyping Tools

Models and prototyping tools based on AAA have demonstrated their effectiveness in accelerating the development and deployment of signal-processing applications on complex architectures. Notably, a survey on co-design tools for real-time systems by Torngren et al. [Tör+06] highlights key tools such as SynDEx [GLS99; Sor04] and Ptolemy [Eke+03]. SynDEx excels in matching algorithms with architectures, particularly focusing on memory optimization, enabling the design of efficient embedded systems. On the other hand, Ptolemy stands out as one of the earliest and most comprehensive open-source

frameworks for modeling and simulating applications. Its unique feature lies in the ability to graphically edit graph composition that combines different MoC. While Ptolemy primarily focuses on theoretical studies of real-time applications and their simulation on Multi-core CPUs, it cannot generate code directly. DSPCAD Lightweight Dataflow Environment (LIDE) [She+11] is a command-line tool supporting modeling, simulation, and implementation of Digital Signal Processor (DSP) systems. Similarly, Open RVC-CAL Compiler (ORCC) [Yvi+13] generates various types of hardware and software codes from a single DPN-based language named RVC-CAL. However, both LIDE and ORCC face challenges in guaranteeing deadlock-freeness and memory boundedness for applications due to their particular graph-based nature. Simulink [KKM16] is a popular tool for dynamic system modeling and simulation. However, Simulink lacks specialization for dedicated architectures whether for multi-node HPC or embedded systems. Alternatively, Python libraries such as Dask [Roc15] offer multi-node parallelization capabilities by partitioning tasks and exploiting DAG to reveal parallelism. However, Dask is not typically classified as a rapid prototyping tool but rather as a parallel computing framework. Although it effectively handles the data duplication inefficiencies of Python, scalability issues can arise when managing large datasets or computationally complex workflows. Finally, PREESM [Pel+14] is an open-source rapid prototyping tool providing robust analysis for heterogeneous multi and many-core single-node targets.

2.2.2 HPC simulation tools

In [Cas+14], the authors introduce SimGrid and provide an overview of popular simulators aimed at achieving fast simulation for distributed computer systems, where fast refers to the speed of simulation compared to real-time. The primary offline simulators discussed include PMaC's Open Source Interconnect and Network Simulator (PSINS) [Tik+09], LogGOPSim [HSL10], BigSim [ZKK04], and MPI-SIM [PB98]. PSINS employs a monolithic model of collective communications, relying on coarse approximations or extensive calibration experiments for each type of collective operation, with a focus on communication performance and synchronization in parallel environments. LogGOPSim, based on the LogGOPS network model, simulates communication performance, task synchronization, and execution times in LogGOP environments, although its restricted model may limit its versatility compared to other simulators. BigSim integrates a parallel emulator, a Charm++-based programming language, an adaptive MPI environment, and a parallel post-mortem mode simulator for performance prediction, including network simulation.

Designed for large-scale simulations, the ability of BigSim to handle detailed simulations can be complex and require significant computing resources. MPI-SIM, a library for execution-driven parallel simulation of MPI programs, enables prediction of existing MPI program performance based on architectural characteristics. Notably, SimGrid stands out for its versatile framework, accurately capturing a broader range of metrics, including communication performance, resource consumption, latency, and execution times, setting it apart from other simulators.

This thesis aims to leverage PREESM and SimGrid to build Simulator of the Science Data Processor (SimSDP) and meet SKA requirements to simulate astronomical applications in HPC systems with precision and efficiency in various key metrics, including latency, throughput, memory usage, energy consumption, and system cost.

2.3 Existing method for simplifying and optimizing dataflow scheduling

As highlighted in [Hon20], scheduling poses significant challenges, particularly in minimizing latency. While many scheduling problems are known to be NP-complete, certain cases may exhibit polynomial-time solutions, as noted in [KA99]. To address this complexity, various heuristic approaches are employed. List scheduling, for instance, generates a priority list of tasks for execution but does not guarantee optimal scheduling outcomes. Research has shown that this method can lead to suboptimal results and may even increase latency, particularly as the number of PEs increases [Gra69]. These heuristics can be implemented offline or online, with tools like StarPU, a runtime system designed for heterogeneous computing environments that facilitates task scheduling and resource management [Aug+11], and PREESM utilizing list scheduling [KA99]. Alternatively, Integer Linear Programming (ILP) [LM69] has been employed for decades to compute optimal schedules but is limited to small systems due to its computational demands. Constraint programming [DDP18] offers greater flexibility in expressing constraints but requires significant computational time. Other methods such as local search, Evolutionary Algorithm (EA) [ECP06], Satisfiability (SAT) [KS92] techniques are also utilized. For multi-objective scheduling problems, various heuristics like the maximum diversity approach are available. Linear Programming (LP) shows promise in efficiently scaling for similar problems [MR14], while EA has also been utilized for multi-objective scheduling [KC02]. Despite the effectiveness of these techniques, they all have limitations concerning the complexity of

the problem, both algorithmic and architectural. This is particularly problematic for the SKA radio telescope due to the intricate nature of astronomical applications, which often involve vast datasets and complex algorithms for signal processing and the complexity of HPC architectures. This section introduces techniques for simplifying and optimizing dataflow scheduling: clustering and retiming.

2.3.1 Clustering of Dataflow Actor

One approach to simplify the complexity associated with mapping and scheduling algorithms involves reducing the number of actors to be mapped within the SrDAG, while preserving the behavior of the application. This reduction can be accomplished through clustering techniques, which modify the input graph by grouping actors with a particular behavior. Since grouping two or more actors into a single equivalent hierarchical actor may change the behavior of the application, or even create deadlocks, clustering rules have been introduced in [PBL95].

Table 2.1 – Summary of Key Notations

Notation	Description
κ_α	The number of samples consumed on the SDF arc α, per sink invocation.
ρ_α	The number of samples produced on SDF arc α, per source invocation.
δ_α	The number of initial samples ("delay") on the SDF arc α.
src_α	The node that produces the tokens on arc α.
snk_α	The node that consumes the tokens produced on arc α.

If x and y are adjacent nodes in an SDF graph, then:

$$Q(x, y) = \frac{q(x)}{\gcd(q(x),q(y))}$$

We can view $Q(x, y)$ as the number of times that node x is invoked in a single invocation of the cluster x, y .

Definition 14. Considering a consistent SDF graph $G =< A, F >$, and $(x, y) \in A$ is an ordered pair of distinct adjacent actors in G. Then $cluster(x, y, G)$, the graph that results from clustering x,y into a single actor Ω is consistent if the following four conditions hold true.

1. Cycle introduction condition: There is no simple path from x to y that contains more than one arc. A simple path is one which does not visit any actor along the path more than once.
2. Hidden delay condition: If x and y are in the same strongly connected component, then both condition **a** and **b** must hold true.
   a) at least one arc from x to y has zero delay.
   b) for some positive integer k , $\mathbf{q}(\mathrm{x}) = \mathrm{k}\mathbf{q}(\mathrm{y})$ or $\mathbf{q}(\mathrm{y}) = \mathrm{k}\mathbf{q}(\mathrm{x})$.
3. First precedence shift condition: If x is in a nontrivial strongly connected component C, then either condition **a** or **b** must hold true.
   a)
   $$\text{for each } \alpha \in \left\{ \alpha' || \begin{pmatrix} (snk(\alpha') = x) \\ and \\ (src(\alpha') \in C) \\ and \\ (src(\alpha' \notin x, y) \end{pmatrix} \right\} \text{ with snk}(\alpha') \text{ and src}(\alpha')$$
   respectively being the sink and the source of the arc α', there exists integers $k1 > 0$ and $k2 \geq 0$ such that $\rho_\alpha = k1Q(x,y)\kappa_\alpha$, and $\delta_\alpha = k2Q(x,y)\kappa_\alpha$.
   b)
   $$\text{for each } \alpha \in \left\{ \alpha' || \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (src(\alpha' \notin x, y) \end{pmatrix} \right\} \text{ , there exists integers}$$
   $k1 > 0$ and $k2 \geq 0$ such that $\kappa_\alpha = k1Q(x,Y)\rho_\alpha$, and $\delta_\alpha = k2Q(x,y)\rho_\alpha$.
4. Second precedence shift condition: If y is in a nontrivial strongly connected component C, then either condition **a** or **b** must hold true.
   a)
   $$\text{for each } \alpha \in \left\{ \alpha' || \begin{pmatrix} (snk(\alpha') = x) \\ and \\ (src(\alpha') \in C) \\ and \\ (src(\alpha' \notin x, y) \end{pmatrix} \right\} \text{, there exists integers}$$
   $k1 > 0$ and $k2 \geq 0$ such that $\rho_\alpha = k1Q(x,y)\kappa_\alpha$, and $\delta_\alpha = k2Q(x,y)\kappa_\alpha$.

b)

$$\text{for each } \alpha \in \left\{ \alpha' || \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (src(\alpha' \notin x, y) \end{pmatrix} \right\}, \text{ there exists integers}$$

$k1 > 0$ and $k2 \geq 0$ such that $\kappa_\alpha = k1Q(x, Y)\rho_\alpha$, and $\delta_\alpha = k2Q(x, y)\rho_\alpha$.

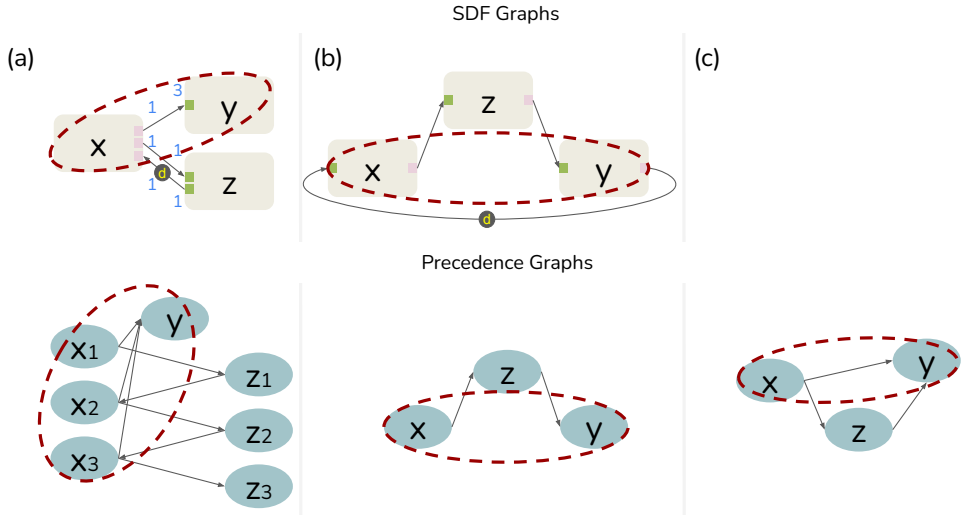

Figure 2.5 – (a) illustrate the violation of the first precedence shift condition, (b) illustrate the violation of the hidden delay condition, and (c) illustrate the violation of the cycle introduction condition

Clustering SDF actors is a powerful approach to customize computation behavior for a specific target. Clustering involves grouping SDF actors based on specific criteria into a single equivalent actor. The widely employed technique involves a hierarchical dataflow-based clustering method, which comprises two branches: agglomerative and divisive.

Agglomerative Clustering Techniques consist in a bottom-up approach that iteratively merges actors based on their similarities. The authors in [PBL95] also present four clustering techniques. The first method is a manual clustering technique. This approach relies on senior developer analysis to understand the application and enable effective optimizations. However, the involvement of senior developers is time-consuming and error-prone. This task is especially challenging when the application size increases. Automatic strategies like the one presented in this thesis enable the safe and efficient handling of

complex application programs. The second method consists of clustering SDF subgraphs as much as possible. Similar to the previous method, it also depends on the developer's judgment to stop the iteration of subgraph clustering which, if uncontrolled, leads to a sequential application. The third method is the Unique Repetition Count (URC) clustering technique. The method consists of creating an SDF subgraph of at least two sequential actors with identical RV coefficient and no internal state as defined in Definition 15. The last method, a dynamic clustering one, falls beyond the scope of this thesis, which primarily addresses static allocation. This process was originally introduced in [Sar87].

Definition 15. Let $G = (V, E)$ be a directed acyclic graph representing a well-ordered, URC SDF subgraph, where V is the set of vertices and E is the set of edges. The graph G satisfies the following properties:

1. **Well-Ordered:** G is well-ordered if it admits only one topological sort, ensuring a unique ordering of vertices.
2. **URC:** G is a URC SDF subgraph if all vertices in V have identical repetition counts, denoted as **q**-values.

Another clustering technique depicted in [BL93], the Pairwise Grouping of Adjacent Nodes (PGAN) method involves iteratively clustering two neighboring actors, resulting in nested looped schedules. This iterative process is applied to an entire SDF graph, pairing actors until only a single cluster remains. The resulting schedule variability is influenced by the selection of coupling heuristics. Consequently, the technique offers a multitude of potential configurations, making comprehensive evaluation a complex task. Its extension, the Pairwise Grouping of Adjacent Nodes for Acyclic graph (APGAN) clustering technique introduced in [BML97] shows that first clustering couples that form a cluster with the highest repetition count ρ lead to a minimum memory requirement schedule and minimize the possible configuration see Definition 16.

Definition 16. If Z is a subset of actors in a connected, consistent SDF graph:

$$\rho(Z) \equiv \gcd(\{\mathbf{q}(A) \mid A \in Z\})$$

In essence, agglomerative clustering coarsen application, and uncontrolled clustering results in sequential application.

Divisive Clustering Techniques, as a top-down approach, involve recursively dividing or partitioning a graph into smaller subgraphs [Gen91]. This method is particularly efficient for partitioning a graph on a multi-node architecture. The advantage lies in each

node being able to independently process its assigned subset of the graph, making use of available resources like CPU cores and memory on that node. Various multi-node techniques have been explored in the literature. For instance, a task scheduling algorithm based on task duplication is proposed for multi-core-cluster systems [YGD13]. This algorithm comprises two steps: first, processes are assigned to processor nodes, and then threads within processes are assigned to core nodes. However, this algorithm does not handle core frequency variability within nodes or transfer cost, and the quality of the final partitioning depends on the initial partitioning relevance. Another approach, outlined in [Jeo+21], employs an EA to optimize task mapping onto processors. The authors use an existing Worst-Case Response Time (WCRT) analysis tool to ensure real-time requirements are met for all applications. To reduce analysis time, a clustering technique is introduced, though the process can still be time-consuming. An additional system, Tofu, presented in [WHL19], addresses partitioning very large Deep Neural Network (DNN) models across multiple GPU devices to reduce per-GPU memory usage. Tofu is specifically designed for fine-grained tensor operator dataflow graph partitioning and seamlessly integrates with general-purpose deep learning platforms like MXNet. However, it should be noted that Tofu applicability is limited to a specific kind of application and target.

In summary, existing methods for graph clustering are efficient in reducing graph complexity but have limitations, such as neglecting core frequency variability, having limited applicability, and lacking efficient solutions for heterogeneous multi-node and multi-core architectures.

2.3.2 Partitioning by retiming

Pipelining serves as an alternative approach to partitioning a graph, facilitating the efficient distribution of its actors among available resources, akin to divisive clustering. This concept, along with the broader notion of retiming, has been extensively investigated within the realm of Very Large Scale Integration (VLSI) circuit design [LS91; Par07]. Parhi [Par07] formally defined the legality of pipelining specifically for SrSDF graphs. Furthermore, pipelining has been explored in the context of software pipelining [Lam04; All+95], where retiming methods have been applied [CDR98]. However, these studies are limited to SrSDF graphs. The idea of pipelining SDF graphs was initially proposed by Lee and Messerschmitt [LM87c] as an optimization strategy. Subsequently, Gordon et al. [GTA06] and Kudlur et al. [KM08] introduced heuristic approaches to pipeline partially unfolded SDF graphs. Gordon et al.'s heuristic involves an initial transformation of the original ac-

Table 2.2 – Summary of Dataflow Clustering Techniques

Single-Node Clustering Methods	**Advantages**	**Limitations**
Manual [PBL95]	User control over grouping	Tedious and prone to deadlocks
SDF [PBL95]	Handles subgraphs efficiently	May alter overall application behavior
URC [PBL95]	Efficient handling of sequential actors	Limited to actors with identical RV coefficient
PGAN [BL93]	Provides many configuration options	Evaluation can be tedious
APGAN [BML97]	lead to a minimum memory requirement schedule	Sequential
Multi-Node Clustering Methods	**Advantages**	**Limitations**
Task Scheduling [YGD13]	Independent processing on each node	Quality depends on initial partitioning
EA [Jeo+21]	Optimizes task mapping	Time-consuming process
Tofu System [WHL19]	Reduces per-GPU memory usage	Limited applicability

tors to balance execution times and adjust parallelism levels. On the other hand, Kudlur et al. employed an ILP formulation to determine the unfolding limit, necessitating the latency as an input. Udupa et al. [UGT09] proposed a comprehensive unfolding of SDF graphs into their SrSDF equivalents, allowing fine-tuning of added delays. Their approach utilizes an ILP formulation to compute the firing stage of each actor. Various studies have explored scheduling in both optimal and heuristic forms [KA99; MG13]. Some works have investigated combining pipelining with scheduling, focusing on SrSDF graphs [YH09] or DAG [CZ12]. Additionally, efforts have been made to find optimal retiming solutions to reduce the makespan of a graph [Zhu+15; Liv+07]. Zhu et al. [Zhu+15] and Liv et al. [Liv+07] utilize symbolic execution of partially unfolded SDF graphs to determine retiming solutions. [Hon+20] focuses on pipelining SDF graphs in their original reduced form

to provide a rapid heuristic without execution, symbolic or otherwise. Another technique involves retiming SDF graphs by introducing initial data into buffers, enabling parallel execution of an actor's firings [KLE17]. This approach is particularly suited for executing the same algorithm on multiple data concurrently, akin to a GPU. In summary, existing methods for graph retiming are efficient methods to enhance the distribution of actors on the available resources, but they are limited by the increasing complexity of graphs.

Conclusion

In this chapter, we have identified several scientific challenges related to resource allocation in the context of rapid prototyping for parallel applications. Specifically, major challenges include efficient prototyping flow design, resource allocation management, code generation, and simulation. We also explored existing work in the field of rapid prototyping, focusing on tools based on AAA methodology and HPC simulation tools. Finally, we discussed current methods for simplifying and optimizing dataflow scheduling, including dataflow clustering and partitioning by retiming. The main scientific challenge identified is the complexity of efficient resource allocation in heterogeneous multi-node and multi-core environments, and the need to simplify and optimize dataflow scheduling to improve the overall performance of parallel systems.

The next chapter, Chapter 3, will detail our specific contribution to these challenges. Specifically, we will focus on the static distribution of dataflow applications across homogeneous single-node and multi-core architectures. Chapter 4 extends the dataflow distribution across heterogeneous multi-node and multi-core architectures. Chapter 5 will introduce a method for the deployment and performance evaluation of applications across various multi-node networks. Finally, Chapter 6 will conclude this work and propose potential research directions for future work.

Part II

Contributions

Chapter 3

Static Dataflow Granularity Optimization for Homogeneous Multicore Architectures

3.1 Introduction & Motivation

This chapter introduces a fast method to generate high-performance parallelized code from a dataflow specification of an application. Dataflow MoCs are efficient programming paradigms for expressing the parallelism of an application [LH89]. Traditionally, mapping and scheduling methods for dataflow MoCs rely on complex graph transformations to explicit their parallelism which can result in complex graphs for embarrassingly parallel applications. For such applications, state-of-the-art mapping and scheduling techniques are prohibitively complex, while the exposed parallelism often exceeds the parallel processing capabilities of the target architecture. This chapter describes three clustering methods, known as the Scaling up of Clusters of Actors on Processing Element (SCAPE) method, which assert their ability to adjust the granularity of an application to match the available PEs in an architecture. They achieve this by employing clustering techniques to optimize mapping and scheduling opportunities.

The SCAPE method is described as an agglomerative clustering approach. The method operates as a bottom-up method, systematically merging actors based on their similarities. This technique involves the following steps:

1. Identification of Patterns: Particular patterns of actors are identified.
2. Subgraph generation: These identified actors are grouped to form subgraphs.
3. Schedule: The scheduling of the newly created subgraph is computed.
4. Code Generation: The associated code for the subgraph is generated.

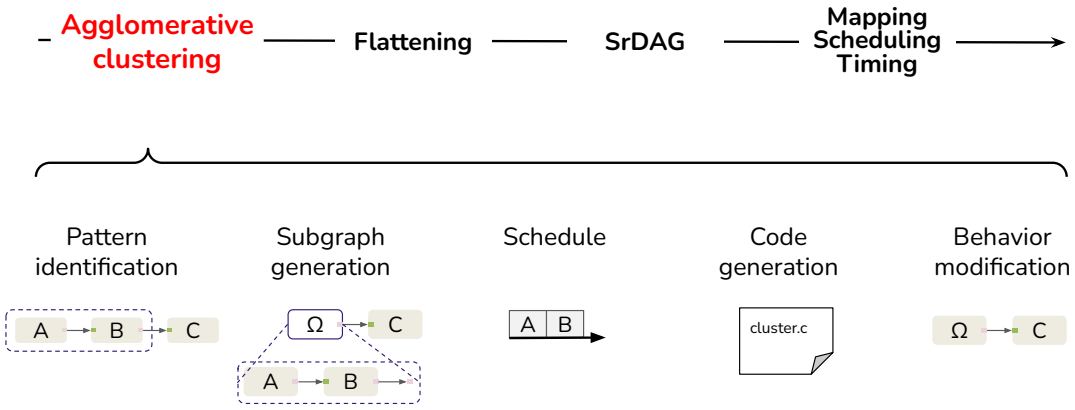

Figure 3.1 – Agglomerative clustering principle

5. Behavior Replacement: The behavior of the hierarchical actor is replaced by the newly generated code.

The process is visually depicted in Figure 3.1 and detailed in the following section.

The rest of this chapter is organized as follows: Section 3.2.1 presents the first version of SCAPE that clusters the data parallelism sequence of actors to match their parallelism to the target architecture. Section 3.2.2 proposes a first extension adding pipeline parallelism on sequential actors and matching pipeline parallelism to the target architecture. Lastly, Section 3.3 extends the two previous ones, considering the hierarchical structure of the PiSDF MoC.

3.2 Fine-tuning Parallelism

3.2.1 Fine-tuning Data Parallelism

Identification of particular pattern

The first version of the SCAPE method, also called $SCAPE1$, proceeds with the agglomeration clustering step and considers two patterns focusing on data parallelism:

URC pattern: Introduced in [PBL95] and presented in Section 2.2, is a sequence of at least two sequential actors with the same RV coefficient **q** without the internal state.

Example 11. Considering the graph $G1$ shown in Figure 3.2 and 4 CPU cores, actors B and C are URC because there are at least 2 sequential actors without any delay on their

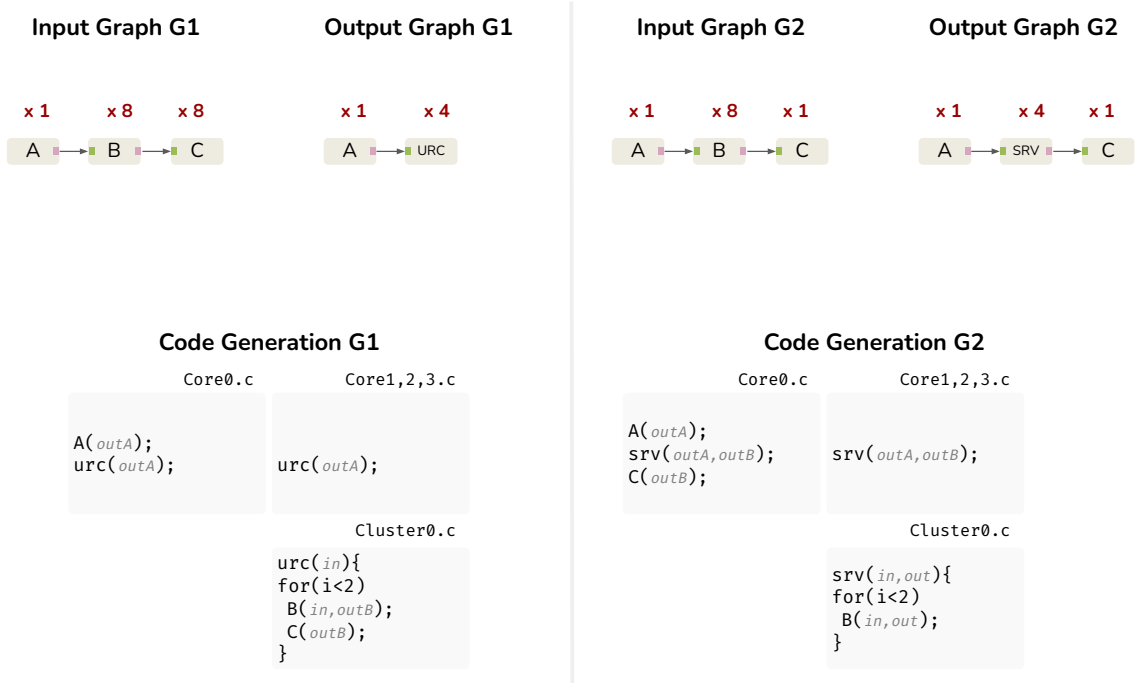

Figure 3.2 – Illustration of the SCAPE method to fine-tune data parallelism on 4 PEs explained in Example 11

connected FIFO and sharing the same RV coefficient $\mathbf{q(B)} = \mathbf{q(C)} = 8$.

Single Repetition Vector (SRV) pattern: Introduced in [Ren+23a], is a single actor that does not belong to an URC candidate, with a RV $\mathbf{q}$ greater than or equal to the number of PEs of the target architecture.

Example 12. We consider the graph $G2$ shown in 3.2 and 4 CPU cores, actor B is an SRV candidate because it is an isolated actor with a repetition greater than the number of cores; here, 8 is greater than 4.

Subgraph transformation

The two identified patterns are processed as two clusters of actors. A subgraph is generated containing the actors B and C from graph $G1$, and another subgraph containing actor B from graph $G2$. Once the actors are isolated in a subgraph, the next step, called the *scaling*, consists of matching the RV coefficient of the clusters of actors to the target architecture. According to [LM87c] to preserve the consistency of a graph G, on each FIFO f the rates of consumed and produced tokens *cons* and *prod* and the RV $\mathbf{q}$ of the source and sink actors *src* and *snk* are linked by the equation:

$$\mathbf{q}(src(f)) \times prod(f) = \mathbf{q}(snk(f)) \times cons(f) \tag{3.1}$$

To calculate the *scaling*, the RV of the hierarchical actor $\mathbf{q}(h_a)$ shall be equal to the greatest common divisor of the RVs of the actors of the subgraph C flattened just above the number of PE n_{PE}.

$$\mathbf{q}(h_a) = gcd(\mathbf{q}(a \in C) \,|\, \mathbf{q}(h_a) \geq n_{PE} \tag{3.2}$$

In case the hierarchical actor contains a FIFO with a number of delays D, special care must be taken when *scaling* the actor. In particular, if the hierarchical actor is directly connected to a delayed FIFO or indirectly via a special actor or an interface connected to a delayed FIFO. If one condition holds true then the calculation of the scaling is indexed on the delay value such as the rates of consumed tokens on the delayed FIFO $cons(f_{h_{a_d}})$ has to be less than or equal to the delay value D.

$$\mathbf{q}(h_a) = gcd(\mathbf{q}(a \in C)) \,|\, cons(f_{h_{a_d}}) \leq D \tag{3.3}$$

To preserve the consistency of the graph, the rates of tokens consumed and produced on the input and output ports by the hierarchical actor $in(h_a)$ and $out(h_a)$, $_f$ for final and $_i$ for the initial value, are scaled as follow:

$$\begin{cases} in(h_a)_f = in(h_a)_i \times \mathbf{q}(h_a)_i / \mathbf{q}(h_a)_f \\ out(h_a)_f = out(h_a)_i \times \mathbf{q}(h_a)_i / \mathbf{q}(h_a)_f \end{cases} \tag{3.4}$$

Thus the actors from the subgraph are executed $\mathbf{q}(a \in C)/\mathbf{q}(h_a)$ times.

Example 13. We consider the graph $G1$ depicted in 3.2 and a target architecture with 4 CPU cores. As the RV $\mathbf{q}$ of the URC cluster is $\mathbf{q}(a \in URC) = 8$, then the *scaling* will be $gcd(8, 4) = 4$. Respectively, considering the graph $G2$ shown in 3.2 and a target architecture with 4 CPU cores. As the RV $\mathbf{q}$ of the SRV cluster is $\mathbf{q}(a \in URC) = 8$, then the *scaling* will be $gcd(8, 4) = 4$.

Cluster code generation

The code generated by the standard flattening approach in the PREESM rapid prototyping framework [Heu+12] consists of translating a dataflow graph into parallel code and takes the form of a specific C file for each target CPU core. Every file contains a part dedicated to the initialization of the application, including the definition of allocated buffers, actors, and FIFOs initialization functions, such as delay initialization. The second part of these files is a loop representing the thread that contains the scheduled firing of actors. It is a function call that implements the behavior of the actor.

The next step of the method is to generate the implementation of the cluster function to be integrated into the existing code generation. The SCAPE approach takes the form of C files where a function that contains the scheduled firing of actors of the subgraph is defined. The schedule is computed with the APGAN algorithm [BML97]. The firing of actors is translated into function calls implementing the behavior of the actors. The functions contain arguments referring to the ports of the actors and exchange data via FIFO buffers.

Example 14. Considering the graph $G1$ and $G2$ depicted in 3.2 the APGAN scheduler yields the following results:

$$S_{URC_{G1}} = 2(BC) \text{ and } S_{SRV_{G2}} = 2(B).$$

This translates to a for loop with 2 iterations.

Graph transformation

The last step is to replace the behavior of the hierarchical actor with the code generated beforehand. The method reduces the size of the SDF and consequently the SrDAG graph later processed by the standard code generation.

Example 15. Considering the graph $G1$ depicted in 3.2 and a target architecture with 4 CPU cores, the size of the graph is reduced from 17 SDF actors to 5 SDF actors. Respectively, considering the graph $G2$ depicted in 3.2, the size of the graph is reduced from 10 SDF actors to 6 SDF actors.

The reduced complexity of the graph significantly decreases the mapping and scheduling time without compromising parallelism.

In summary, the version of the SCAPE method focusing on fine-tuning data parallelism reduces mapping and scheduling time while preserving the parallelism of SDF graphs. It consists of reducing the size of the graph by clustering actors reproducing particular patterns and then reducing the firing instances of these clusters on the target architecture.

3.2.2 Fine-tuning Pipeline Parallelism

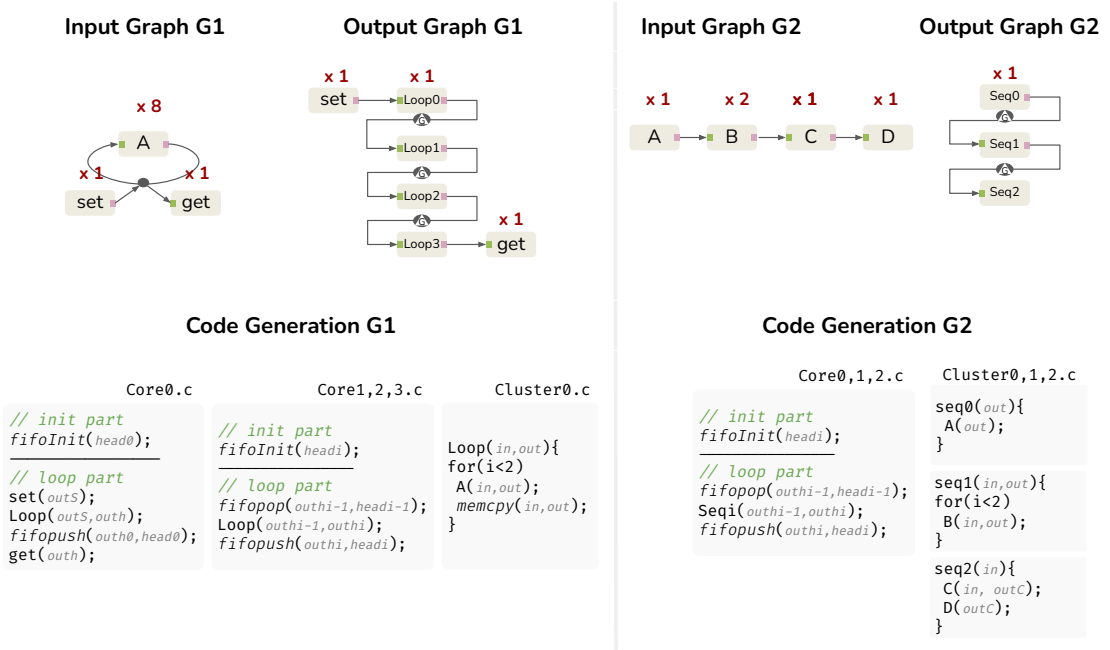

Figure 3.3 – Illustration of the *SCAPE2* method to fine-tune pipeline parallelism on a 4 PEs architecture

Identification of particular pattern

The second version of SCAPE, called $SCAPE2$, extends the first one and considers the two patterns of the first version of SCAPE, URC and SRV, and adds two additional ones:

The **Loop** pattern allows the creation of parallelism on cyclic graphs. Illustrated in Figure 3.3, a cyclic part is a sequence of actors where the last is connected to the first by one or more FIFOs with a LD.

Example 16. Taking into account the input graph $G1$ depicted in Figure 3.3 where the input of the actor A depends on its output but is initialized by the actor *set*, the output of the 8^{th} firing of A is stored by the actor *get* at each iteration of the graph. Given an architecture of 4 CPU cores. The identified actor is A.

The **Sequential** pattern allows the creation of parallelism on sequential graphs. Figure 3.3 is a sequential part of the graph or a part with a degree of parallelism lower than the number of CPU cores. The method is about grouping actors in topological order so that the sum of the execution times of the actors contained in a group tends to be equally distributed. The number of groups must be equal to the number of CPU cores.

Example 17. Considering the input graph $G2$ depicted in Figure 3.3 with a sequence of actors with a degree of parallelism of 1 and 2 and an architecture of 3 CPU cores. We assume that the sum of the execution time of actor A is equivalent to two execution times of B also equivalent to the sum of the execution times of C and D. The method initially identifies a first set composed of actors A, a second set composed of two instances of B, and a final set composed of actors C and D.

Subgraph transformation

The second step of the method consists in generating subgraphs composed of identified actors. The LDs are extracted from the subgraph and connected to the corresponding hierarchical actor. The operation consists of adding interfaces in the subgraph and linking the corresponding ports on the hierarchical actor to the delay(s). Extracting the delay(s) is necessary to compute the internal execution order of the subgraph via the APGAN method.

In the case of the **loop** pattern, the transformation, called semi-unrolling, consists of duplicating n times the hierarchical actor, where n_{loop} is the greatest common divisor of the RVs of the actors of the subgraph C flattened just above the number of CPU cores,

fixing its RV **q** to 1. The RV of the content is also scaled so that it is the result of division.

$$n_{loop} = gcd(\mathbf{q}(a \in C)) \,|\, n_{loop} \geq n_{PE} \tag{3.5}$$

To preserve the consistency of the graph, the FIFO buffers subsequently connected to the actors from the **Loop** are duplicated, distributed, or gathered on the different **Loop** instances.

Example 18. Considering the input graph G shown in Figure 3.3 where the identified actor B has a RV $\mathbf{q} = 8$ and the architecture consists of 4 CPU cores. We obtain: $gcd(8, 4) = 2$. Consequently, the method duplicates the subgraph that contains 2 instances of B 4 times. Thus the SrDAG transformation of this graph, which would have resulted in $8 + 2 = 10$ actors, now consists of 6 actors.

In the case of **Sequential** pattern, the SrDAG transformation of graph G shown in Figure 3.3 which would have resulted in 5 actors is presently 3 actors.

Cluster code generation

As the previous SCAPE version, the third step is to generate a code for the cluster. The particularity of the code resulting from the subgraph that contains looped actors lies in the copy of delayed output(s) on delayed input(s) using *memcpy* function. Considering the input graph G shown in Figure 3.3, the Set function initializes a buffer on its output argument. The first looped function $Loop_0$ copies this buffer on the input of the first B function call. Then the B output buffer is copied on the B input buffer, and the B function call output buffer is copied on the looped function output buffer.

Graph transformation

The last step is to replace the behavior of the hierarchical actor with the code generated beforehand. Then the method integrates pipelining by adding a Globally Persistent Delay (GPD) between the **Loop** and **Sequential** clustering stages. [Hon+20] describe how to automatically create a pipeline on an SDF graph.

The clustered transformed graph is then employed by the rest of the static scheduling method able to generate code. PREESM translates GPD by a *fifoInit* function in the initialization part of the CPU cores C File. This function is used to reset a global buffer at the start of the application. In the second part of the CPU cores C File, the loop part, a

fifopop precedes firing of the **Loop** and **Sequential** clustering pipeline stages loading the previously initialized GPD. The stages are succeeded by a *fifopush* function that stores the last delayed tokens of the actor for the next use. Thus, at each graph iteration, the first firing of clustering pipeline stages receives the delayed FIFO from the previous iteration.

Example 19. Considering the input graph $G1$ shown in Figure 3.3 the method transmits an output graph to the rest of the static scheduling process composed of 4 pipeline stages, the first one containing the setter actor and the last one the getter actor. Stages are linked by GPDs. *fifoInit* function initialized global buffers *head*0, *head*1, and *head*2 in the initial part of the program. *fifoPop* function copies the *head*0 buffer in another buffer *outh* before its reading by *Loop*1 function call. *fifoPush* function copy the *outh* buffer in *head*0 buffer after the *Loop*1 function process.

In summary, the version of the SCAPE method adding fine-tuning pipeline parallelism reduces the mapping and scheduling processing time while increasing the parallelism of SDF graph. The method consists of reducing the size of the graph by clustering actors detecting particular patterns and increasing parallelism by adding pipelines considering the target architecture.

3.2.3 Experiments

The proposed methods noted $SCAPE1$ for the first version of SCAPE and $SCAPE2$ for the second version of SCAPE are applied to deploy the Stereo image processing application on several multicore target architectures. These two proposed methods are compared to the standard dataflow resource allocation method regarding the resource allocation time and the final latency speedup obtained and illustrated in Figure 3.4. The figure illustrates the standard resource allocation without clustering using a red dotted line, the first version of SCAPE using a blue solid line, and the second version of SCAPE using a green dashed line. The results regarding resource allocation demonstrate that both clustering methods expedite the process, particularly the second version of SCAPE, which enhances clustering opportunities. Regarding final latency speedup, the first version of SCAPE matches the performance of the standard resource allocation without clustering. In contrast, the second version of SCAPE consistently delivers a greater speedup.

This superior performance in *SCAPE2* can be attributed to its pipelining mechanism, which provides an "unfair" advantage by allowing multiple processes to execute concurrently. This pipelining not only reduces idle times but also improves overall system

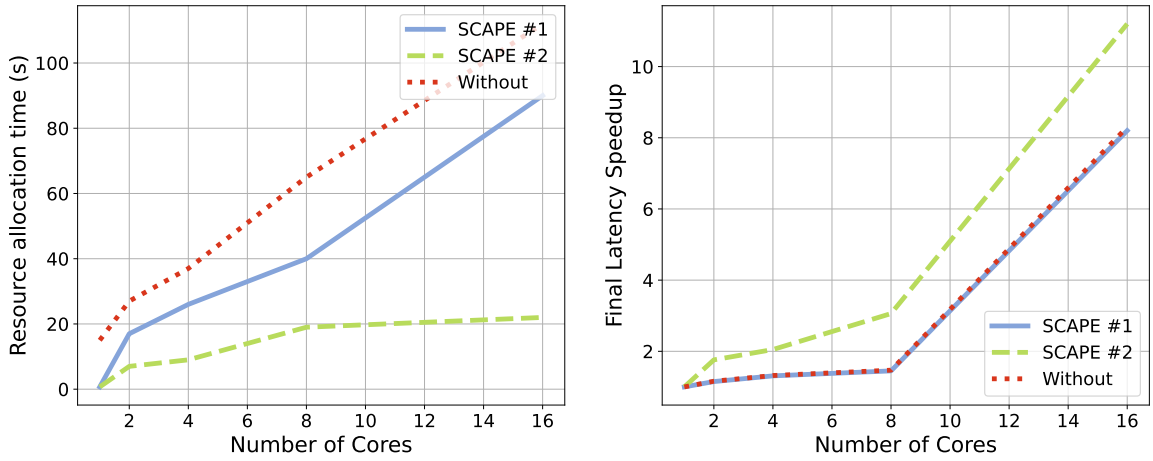

Figure 3.4 – Comparison of analysis time and final latency speedup between the standard resource allocation without clustering, with the first version of SCAPE and with the second version of SCAPE deploying the Stereo application on several multicore architectures

efficiency by better utilizing available resources. Consequently, *SCAPE2* achieves faster resource allocation and improved latency speedup compared to the first version and the standard allocation method. This advantage underscores the importance of advanced techniques such as pipelining in optimizing resource allocation and performance in complex systems.

3.3 Considering the Hierarchical Context State in Granularity Adaptation

This section proposes several strategies for managing dataflow graph hierarchy and clustering. Section 3.3.1 presents a parameterized management used in the two first versions of SCAPE, and Section 3.3.2 presents a new extension of both SCAPE methods by considering the hierarchical context.

3.3.1 Granularity Configuration Method for Input Parameter

The two first versions of the SCAPE method take as input the PiSDF graph of N hierarchy levels that models the application and an integer value corresponding to the number n_c of hierarchy levels that the user wants to group coarsely. The output of the

new method is a transformed graph with the RV **q** associated with the actors located in the subgraphs on $N-n_c$ level reduced to the number of PE that compose the architecture. A graph on N levels will have $N+2$ possible configurations of $N-n_c$ levels, as illustrated in Figure 3.5.

— Level 0 configuration: it is the state-of-the-art configuration where the entire graph is flattened before producing the SrDAG for scheduling.

— level $N+1$ clustering configuration: it groups the entire graph into a single actor, thus resulting in a mono-core schedule.

— Level 1 clustering configuration: It corresponds to generating groups on the bottom levels and reduces the RV **q** associated with the actors located in the subgraphs on this level to the number of PE.

— Level $l \,|\, l \in [2, N+1[$ clustering configuration: It corresponds to coarsely grouping bottom levels, generate groups on the just upper levels, and reduce the RV **q** of actors on this level to the number of PE.

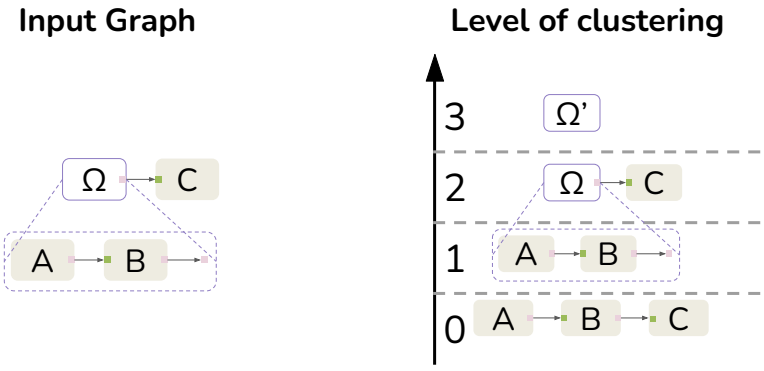

Figure 3.5 – Clustering configurations based on hierarchical dataflow levels

In summary, the method enables the developer to select the optimal granularity to enhance latency, throughput, and memory requirements for a target architecture. However, it necessitates human expertise to assess the suitability of each clustering configuration. Specifically, choosing a small number of hierarchy levels that the user wants to coarsely group n_c leads to fine granularity, which improves parallelism but incurs a large scheduling overhead. Conversely, a large number n_c results in coarser granularity, reducing scheduling overhead but risking loss of parallelism. The next section proposes an automated strategy to maximize clustering at each hierarchical level.

3.3.2 Hierarchical Context State Aware method

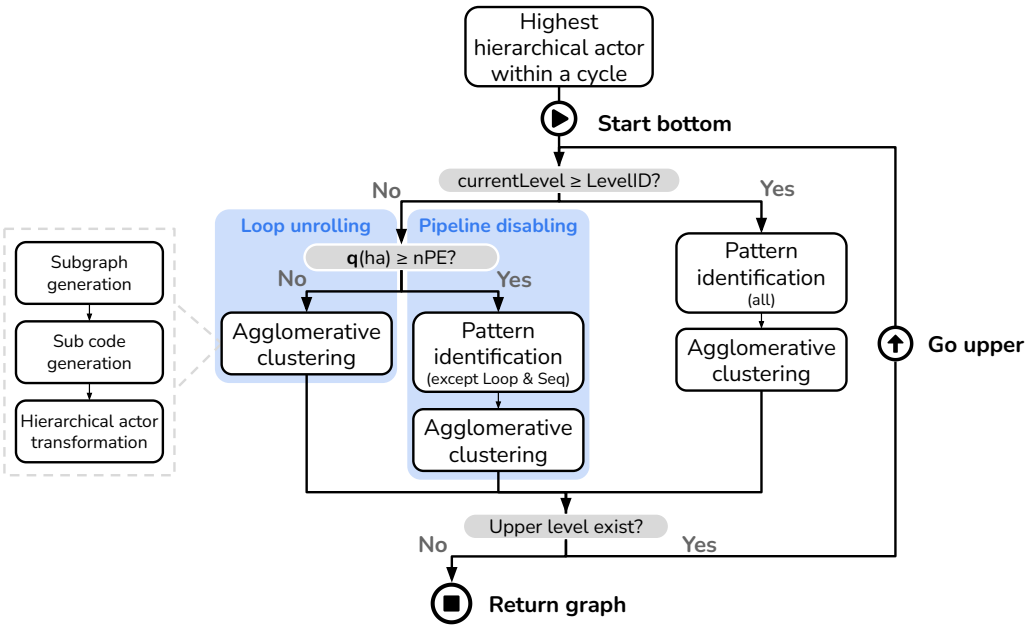

Figure 3.6 – Algorithm of the third version of the SCAPE method

The proposed clustering method extends the previous SCAPE methods and as $SCAPE1$ and $SCAPE2$ is inserted upstream of the standard resource allocation process. Unlike $SCAPE1$ and $SCAPE2$, the proposed method returns a single well-suited clustering configuration to the architecture. The algorithm of the method is illustrated in Figure 3.6.

The different parts of the algorithm are detailed in the following sections. The method involves identifying hierarchical data flow levels that impede pipeline utilization. It also encompasses the recognition of four actor patterns within authorized levels. These patterns are then replaced by hierarchical actors that encapsulate the identified actors. Additionally, the algorithm involves scheduling computation for the newly formed subgraph, and the subsequent replacement of the behavior of the hierarchical actor with the generated code.

Identification of the highest hierarchical actor within a cycle

The method takes into account the limitations of prior work, where pipeline insertion in sequential portions was performed without considering the hierarchical context. This previous approach could inadvertently introduce pipelines within a cycle, potentially

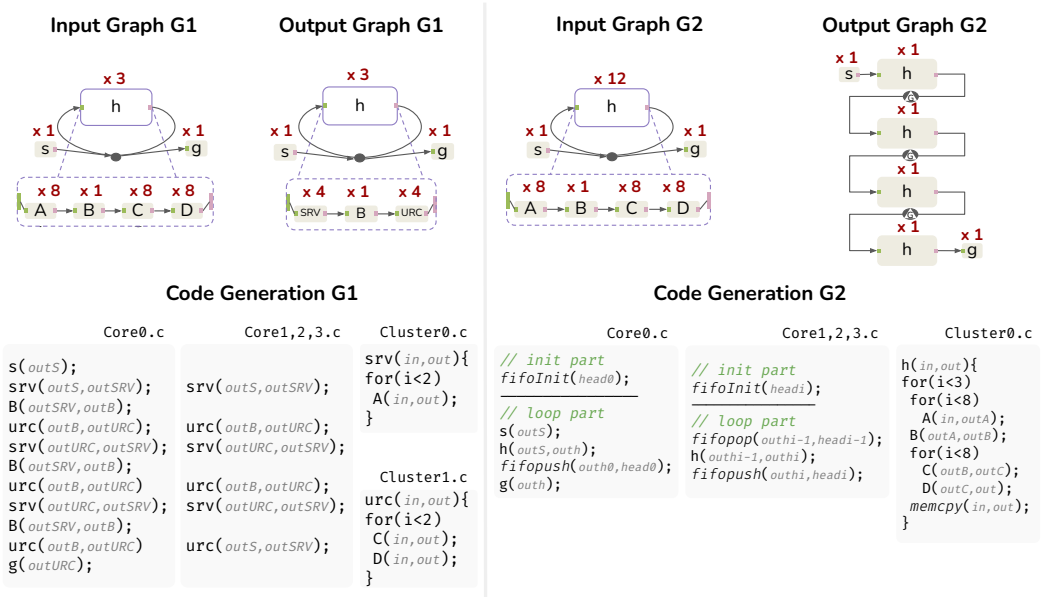

Figure 3.7 – Illustration of cycle management on a 4-core CPU architecture

causing a misalignment of token orders. To address this issue, the new method starts by assessing the feasibility of generating pipelines at a specific level, considering that cycles should not include pipelines. This assessment is carried out by initially traversing the graph from top to bottom and identifying the highest level where a hierarchical actor is present within a cycle. The clustering approach chosen depends on the repetition of the identified cycle and the available PEs:

— If the common repetition among all actors within the cycle, including the hierarchical actor, is less than the number of PEs, the method allows all specific clustering patterns from SCAPE, that are unrelated to the pipeline on the lower levels of the hierarchical actor. In other words, only data parallelism is reduced on the child levels of the hierarchical actor.

— Conversely, if the common repetition among all actors within the cycle, with the hierarchical actor, is greater than the number of PEs, the method coarsely groups the hierarchical levels below the hierarchical level of the actor and proceeds with semi-unrolling.

Example 20. Considering the graphs $G1$ and $G2$ illustrated in Figure 3.7 on 2 levels. On the top level, a hierarchical actor h is self-looped and its content is a sequence of actor A, B, C, D. h_{G1} is fired three times per graph iteration and h_{G2} is fired twelve times. On both graphs, the highest hierarchical actor within the cycle is h located on the top-level

graph.

Identification of particular pattern

In this section, we detail the specific patterns of actors that will be identified for clustering. The choice of clustering patterns is contingent on the repetition **q** of the highest hierarchical actor within the graph ha, as solved in the preceding section, and whether we examine the lower or upper levels concerning the hierarchical actor. The method follows a bottom-up approach, starting the recognition process at the lowest hierarchical level of the graph. Once all specific actors at the current level are clustered, the method proceeds to seek patterns at higher hierarchical levels.

Here are the patterns and corresponding scenarios:

— When the collective repetition of all actors within the cycle, including the hierarchical actor, is below the number of available PEs and the current hierarchical level is lower than the level where the highest hierarchical actor is located, the method will pinpoint two patterns from the first method, SCAPE. These patterns aim to reduce data parallelism based on the number of PEs. These two patterns are URC and SRV, as introduced in Section 3.2.1. Additionally, in pursuit of further graph complexity reduction, the method will identify another pattern: the **Low latency impact (LLI)** pattern. This process consists of clustering actors with a negligible execution time compared to other actors on the same hierarchical level. This cluster respects the topological order of the graph and has no internal state. This case corresponds to the *Pipeline disabling* stage in Figure 3.6.

— Conversely, when the repetition of all actors within the cycle, including the hierarchical actor, exceeds the number of available PEs, the method coarsely groups levels lower than the one containing the highest hierarchical actor. This scenario is represented by the *Loop unrolling* stage in Figure 3.6.

— As for the upper levels, the method takes into account all patterns, as it no longer faces the potential insertion of pipelines within cycles.

Example 21. Considering the graphs $G1$ and $G2$ in Figure 3.7 and a 4-core CPU architecture. As h_{G1} is fired three times per graph iteration that is less than 4 then its content is the object of clustering URC, SRV and LLI patterns of actors. As h_{G2} is fired twelve times per graph iteration that is more than 4 then its content is the object of coarse clustering.

Subgraph transformation

Sections 3.2.1 and 3.2.2 delved into the concept of agglomerative clustering techniques and their role in streamlining dataflow actors. Now, the focus shifts to the next phase in the proposed method, termed *Subgraph Transformation* which is a pivotal step in the hierarchical clustering of dataflow actors. This phase operates subsequent to pattern identification and entails the generation of subgraphs containing the identified actors. Within these subgraphs, the delays of all persistency are systematically extracted and interconnected with the corresponding hierarchical actors. The subgraph transformation process encompasses the addition of interfaces to the subgraph and the establishment of links between the ports on the hierarchical actor and the delay(s). Extracting these delay(s) serves a crucial purpose, enabling the computation of the internal execution order of the subgraph through the APGAN method. Moreover, it is important to highlight that at this step, the proposed method scales the URC and SRV subgraphs to align them with the target architecture. This scaling process ensures that these subgraphs iteratively match the number of PE of the target. The subsequent example provides further clarity on this scaling operation and its implications on different graphs within the context of the execution of the proposed method.

Example 22. Considering the graphs $G1$ and $G2$ in Figure 3.7 and a 4-core CPU architecture. URC_{G1} and SRV_{G1} are scaled to iterate 4 times and their content twice per graph iteration. h_{G2} is duplicated 4 times and its content is rolled up in a loop that iterates 3 times per graph iteration so that $3 \times 4 = 12$.

Cluster code generation

Following the agglomerative clustering process, the subsequent phase entails the generation of code for the cluster. The method is implemented in the open source PREESM framework [Pel+14]. The tool takes as input a dataflow graph where standard actors are specified in C code. Consequently, the cluster code also takes the form of C files, where a specialized function is defined. This function encapsulates the scheduled firing of actors within the subgraph, and the schedule itself is computed with the APGAN algorithm. The translation of actor firings into function calls plays a pivotal role, effectively implementing the behavior of these actors.

In the generated code, the *for loops* symbolize the RV values of the actors, allowing for the precise execution of their behaviors. Moreover, the functions include arguments that

reference the ports of the actors, and data exchange is facilitated through FIFO buffers.

In scenarios where the *loop* pattern is applied and semi-unrolling is performed, feedback mechanisms are transformed into a *memcpy* function. This function is responsible for copying data from an output buffer to an input buffer, aligning the code with the specific behavior of the actors.

Example 23. Considering the graphs $G1$ and $G2$ Figure 3.7 and a 4-core CPU architecture. The schedules of the clusters are calculated using the APGAN algorithm and are as follows:

— $S_{SRV_{G1}} = 2(A)$

— $S_{URC_{G1}} = 2(C\ D)$

— $S_{h_{G2}} = 3(8A\ B8(C\ D))$

The corresponding code for these clusters is illustrated in Figure 3.7.

Graph transformation

The behavior of the hierarchical actor parent of the subgraph is then replaced by the newly created cluster code. Depending on whether the current search level is higher than the identified level and the identified actor is the candidate of *Loop* or *Sequential* patterns then GPDs are inserted between them creating pipeline stages.

Example 24. One schedule of the output graph of G1 illustrated in Figure 3.7 can be:
$S_{G1} = s\ 4(SRV)\ B\ 4(URC)\ 4(SRV)\ B\ 4(URC)\ 4(SRV)\ B\ 4(URC)\ g$
A delay is inserted between two instances of h_{G2} creating four pipeline stages. One schedule of the output graph of G2 can be: $S_{G2} = s\ 4(h)\ g$

3.3.3 Experiments

Experimental setup

The proposed method is applied to three image processing applications: OpenVVC(1), Squeezenet(2), and Stereo(3) on a SPiDF description. The OpenVVC dataflow model has been introduced in [Hag+22], the author particularly explores parallelism between tiles in the VVC decoder. The Squeezenet application is a CNN for computer vision introduced in [Ian+16]. The experimented Stereo matching algorithm has been introduced in [Zha+13] and aims at reconstructing the disparity maps with a pair of images. OpenVVC and Stereo SPiDF models present 1 hierarchical level whose hierarchical actor is self-looped

forbidding internal pipelining. Squeezenet SPiDF model presents 3 hierarchical levels and clustering opportunities on each of them. All of these dataflow models are available in GitHub [1].

Table 3.1 compares graph complexity based on the size of SrDAG in terms of the number of Actors, noted A, and the number of FIFOs, noted f, the resource allocation time and the theoretical speedup deploying Stereo application on 4-Core architecture with the state-of-the-art techniques and the SCAPE methods. Then the third version of SCAPE, noted $SCAPE3$ is compared to the standard resource allocation without clustering, noted NC, and to the Best Clustering configuration of the first two SCAPE methods, noted BC. Owing the fact that the previous SCAPE method provides a set of clustering configurations, the one that offers the best tradeoff in terms of the number of SrDAG actors, resource allocation process time, also called analysis time, and throughput speedup is chosen. Table 3.2 extends the comparison of the graph complexity based on the size of SrDAG on three image processing applications on several multicore target architectures. Figure 3.8 illustrates analysis time, that is the time the tool takes to map schedule and generate code to an application, and throughput speedup. Figure 3.9 compares two critical performance metrics: the average data transfer time per thread, and the average synchronization time per thread. The average data transfer time per thread represents the time spent by the *memcpy* function to efficiently distribute tokens among several instances of actors. This time duration exhibits variability depending on the specific thread in which these functions are executed, and the metric presented here captures the average data transfer time per individual thread. Concurrently, the average synchronization time per thread reflects the delays introduced by employing *pthread* barriers to synchronize the various threads involved in the process. Since the barriers necessitate certain threads to await the completion of others, there can be discrepancies in the waiting times across threads, thus resulting in varying synchronization times.

Given that the performance criteria heavily depend on the architecture and that the method leverages architectural details in the graph transformation process, experiments were conducted across a range of architectures, specifically those with 1 to 8 homogeneous cores. To conduct these experiments, the proposed method was implemented within the PREESM [Pel+14] rapid prototyping framework, which is part of open-source projects. The experiments were carried out on a desktop computer featuring an 8-core Intel i7-8665U processor and 31.2 GB of RAM.

1. `https://github.com/preesm/preesm-apps`

Clustering method	A	f	Resource allocation time (ms)	Theoretical Speedup
No clustering	313	1069	61827	2,13
Original URC [PBL95]	226	797	47969	2,40
SDF - 1 level [PBL95]	1	0	1466	1
SDF - 2 level [PBL95]	37	58	7815	1,99
SCAPE1 - 1 level [Ren+23a]	13	17	2618	1,81
SCAPE1 - 2 level [Ren+23a]	84	160	9610	2,36
SCAPE2 - 1 level [Ren+23b]	13	17	2618	1,81
SCAPE2 - 2 level [Ren+23b]	116	190	11987	3,15
SCAPE3 [Ren+24a]	94	152	9630	3,15

Table 3.1 – Comparison of the number of actors A and the number of FIFO f of the SrDAG, the resource allocation time and the theoretical speedup between state-of-the-art clustering method deploying Stereo application on a 4-Core architecture

		Number of CPU cores							
		1		2		4		8	
App.	**Met.**	A	f	A	f	A	f	A	f
(1)	NC	7262	39995	7262	39995	7262	39995	7262	39995
	BC	**1**	**0**	**12**	**14**	**24**	**34**	**99**	**159**
	*SCAPE*3	**1**	**0**	**12**	**14**	**24**	**34**	**99**	**159**
(2)	NC	5452	17262	5452	17262	5452	17262	5452	17262
	BC	**1**	**0**	1364	3589	1376	3916	1400	4197
	*SCAPE*3	**1**	**0**	**124**	**260**	**188**	**480**	**316**	**920**
(3)	NC	313	1069	313	1069	313	1069	313	1069
	BC	**1**	**0**	73	125	76	143	**89**	**313**
	*SCAPE*3	**1**	**0**	**53**	**85**	**61**	**113**	**89**	**313**

Table 3.2 – Comparison of the number of actors A and the number of FIFO f of the SrDAG between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, as well as the clustering configuration of the *SCAPE*3 method on 3 applications: OpenVVC(1), Squeezenet(2), and Stereo(3) on various number of CPU cores

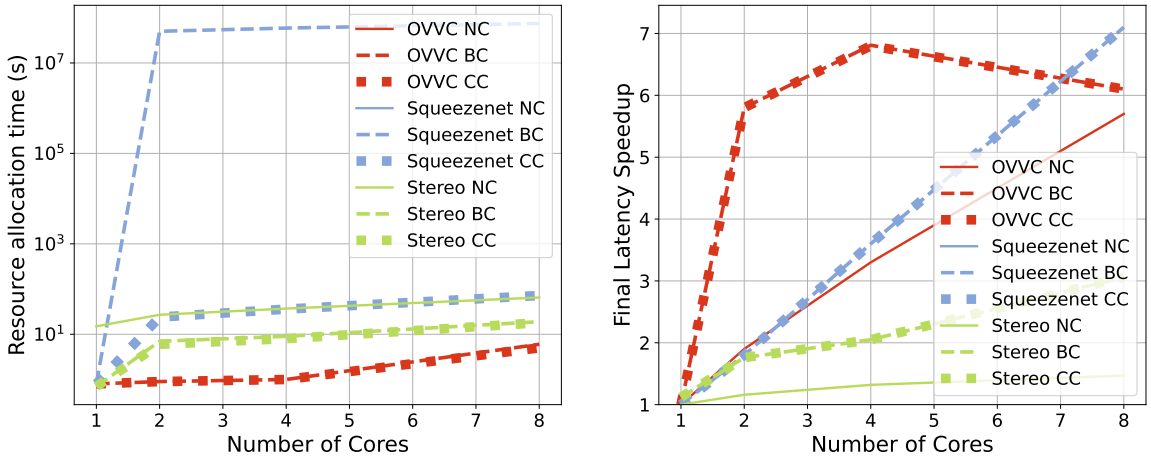

Figure 3.8 – Comparison of analysis time and throughput speedup between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, as well as the clustering configuration of the proposed method (CC), across three applications and varying numbers of CPU cores. Squeezenet NC are not shown because they are too large.

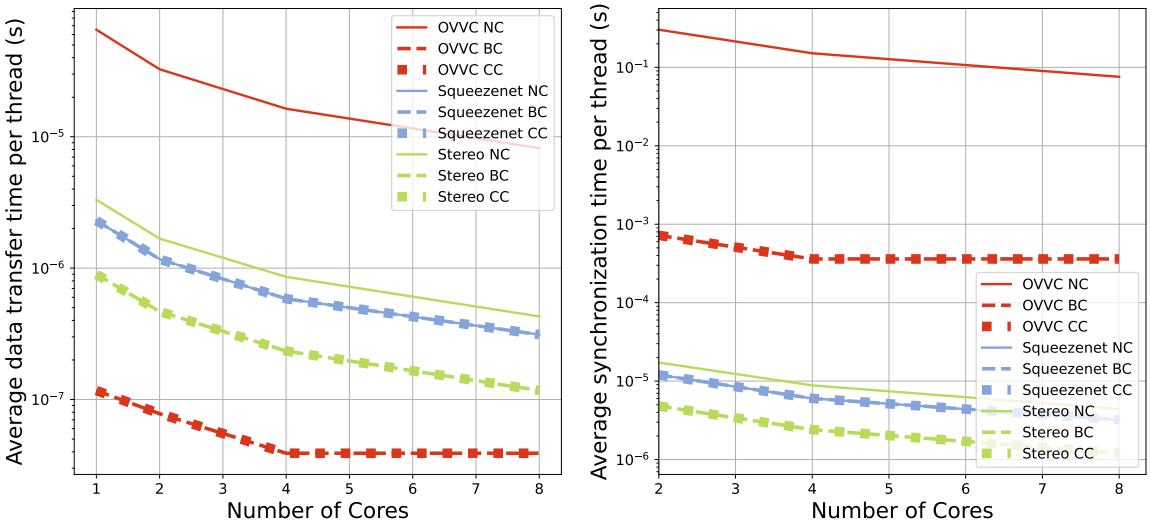

Figure 3.9 – Comparison of average data transfer and synchronization times per thread between the No Clustering (NC) and the Best Clustering (BC) configurations of the previous SCAPE method, along with the clustering configuration of the proposed method, across three different applications and varying numbers of CPU cores

Resource allocation time evaluation

The experimental results in Tables 3.1 and 3.2 unveil the impact of graph complexity on the performance of the proposed method. The size of the SrDAG, as indicated by the number of actors, plays a critical role in influencing the efficiency of the mapping and scheduling process. The tables illustrate a key trend: larger graphs, characterized by a higher number of actors, tend to induce more time-consuming computations during the mapping and scheduling phase, as the method explores various possibilities on the target architecture. In contrast, smaller graphs lead to a simplified process. Moreover, the experimental findings highlight that the complexity of the graph generated by $SCAPE3$ is either lower or equivalent to the best outcome achieved by the first two versions of the SCAPE method. Furthermore, it significantly surpasses the method without clustering, affirming its efficiency in streamlining the dataflow graph transformation process. Specifically, for the OpenVVC dataflow model, the proposed method and the previous SCAPE method both provide the same configuration, the best. The previous SCAPE methods has to evaluate all possible configurations to retrieve the best one, while the proposed method gives the best solution directly. The finding solution leads to a substantial reduction in the number of actors, from 7262 to 24, on a 4-core architecture. This configuration similarity arises particularly on dataflow graphs with a small number of hierarchical levels and few of them admit clustering opportunities. Specifically, in the case of the OpenVVC dataflow model with 1 hierarchical level and clustering opportunities localized on a single hierarchical level, both the proposed method and the previous clustering method achieve the same configuration post-clustering. This is highlighted in the Squeezenet application that has clustering opportunities on each hierarchical level the proposed method provides the least complex graphs better than all other configurations. Concerning the stereo application the method provides the least complex graphs on small architecture and then retrieves the same configuration as the previous SCAPE methods. The method provides a graph complexity adapted to the number of CPU cores of the target architecture, therefore the complexity of the resulting graph increases as the architecture increases until it reaches the original configuration when the degree of parallelism of the description of the origin becomes lower than the number of CPU cores and the pipeline gains reach the system limit.

The experimental results depicted on the right of Figure 3.8 compare the three use cases: in red OpenVVC, in blue Squeezenet, and in green stereo; on the three methods: in dotted line, No clustering, in dashed line, the previous SCAPE method, and in full line,

the proposed method. The results in terms of analysis time testify to the link between the complexity of SrDAG and the cost of the analysis because the method provides a faster analysis if not equal to the previous method. The proposed method always provides on these curves an analysis time of a few seconds. The curves for Squeezenet and OpenVVC without clustering are not visible due to their high complexity, causing the analysis to reach the computational limits of the test computer after a full day of processing. This indicates that the computational capabilities of the tool have been pushed to their limits without reaching the code generation stage.

Final latency evaluation

This section investigates the performance of the method by analyzing the final latency speedup results displayed in Table 3.1 and on the left side of Figure 3.8. These results demonstrate that the method outperforms the configuration without clustering, which is otherwise identical to the previous SCAPE configuration. Notably, the complexity of the OpenVVC and Squeezenet applications is substantial, making direct output generation impossible without the aid of the method. In the case of OpenVVC, manual modifications have been applied to the configuration without clustering to enable performance comparison. However, such manual modifications are not replicable for Squeezenet, resulting in the absence of a corresponding final latency curve with the standard resource allocation method, hence the need for optimization.

Stereo and OpenVVC, both containing cycles, benefit from the proposed method by leveraging the semi-unrolling feature. This approach brings a two-fold advantage. Firstly, it introduces pipeline stages, thereby increasing parallelism, resulting in improved speedup compared to the configuration without clustering. Secondly, as loop pipeline stages are created based on the greatest common divisor between the number of CPU cores and the number of repetitions of the original loop, the performance curve of the method exhibits a step-like pattern. The points where the curve reaches a ceiling correspond to the initiation of each new step in the performance improvement.

The equal performance in final latency obtained with the proposed method and the previous SCAPE methods concerning Stereo and OpenVVC is because the methods retrieve the same graph configuration. Concerning Squeezenet it is because both configurations as similar parallelism setups which is detailed in Section 3.3.3. The reason why the method outperforms the OpenVVC dataflow model is a generic model independent of any target architecture. The goal of the developer is to model the application once and

deploy it on any available architecture, which is useful in the first phases of a project to quickly evaluate the potential of a solution. This generic model greatly limits any potential hardware-specific optimizations, which are automatically provided by the proposed method, thus achieving both the enhancement of rapid prototyping to quickly deploy an architecture-independent model of the application and the provision of hardware-specific optimizations.

Thread-level data transfer and synchronization evaluation

The results shown in Figure 3.9 reveal significant time savings in implementing parallelism through specific clustering. This implementation encompasses parallelism in the description, inter-thread data transfer, and synchronization processes, as detailed in [Pel+13].

Applications with a lot of communication between computations such as OpenVVC with 39995 FIFOs in Table 3.2 can suffer from the synchronization of dependent computations on several threads more than the data transfer time. In fact, the average data transfer time is negligible by a few nanoseconds on the CPU architectures compared to the average synchronization time of a few milliseconds.

There are two types of synchronization used in PREESM: the synchronization of the beginning and end of the thread loops with the use of barriers and the synchronization between dependent computations with the use of *send* and *receive* functions on each thread between each dependent instance. The reduction of complexity thanks to clustering reduces the number of communications, from 39995 f to 14 f on 2 core architecture concerning OpenVVC, which reduces the number of synchronization and thus the average synchronization time per thread, from $302ms$ to $722us$. As the number of cores increases, the dependent computations and synchronization are distributed, reducing the gain obtained through clustering. Hence the clear acceleration in final latency on 2 cores, fades with the complexity of the architecture.

3.4 Summary conclusion

This chapter proposed a new clustering method, called SCAPE for optimizing the execution of a program on a homogeneous single-node target architecture. The method consists of matching the application parallelism to the parallelism of the target architecture by clustering a particular pattern of dataflow actors. The SCAPE method has three versions summarized in Table 3.3, the first one focuses on data parallelism match-

	v1	**v2**	**v3**
Match data parallelism	✓	✓	✓
Match pipeline parallelism	✗	✓	✓
Set of clustering configurations	✓	✓	✗
Hierarchical context-aware	✗	✗	✓

Table 3.3 – Summary conclusion of the three version of SCAPE

ing [Ren+23a], the second one adds pipeline parallelism matching [Ren+23b] and the last one considers hierarchical context state [Ren+24a].

Chapter 4

Static Dataflow Distribution on Heterogeneous Multi-Node & Multi-Core Architectures

4.1 Introduction & Motivation

This chapter introduces SimSDP a dataflow-based resource allocation method for heterogeneous multi-node and multi-core HPC target architectures. Our approach addresses both computation partitioning on processing elements and automated code generation. The proposed method offers an effective solution for intelligently partitioning large processes, such as astronomical backends, on available resources. Our method significantly speeds up the resource allocation process to within a few minutes, regardless of the complexity of the algorithm, in contrast to the exponential allocation time displayed by state-of-the-art methods. Experimental results show that the method accelerates the resource allocation process by a factor of 10 to 72 for deploying a Radio Frequency Interference (RFI) filter with a small data set on three multi-node and multi-core architectures compared to the standard methods. This improvement is achieved by optimizing task allocation and leveraging automated code generation. The findings contribute to the efficient utilization of multi-node architectures and accelerate the development of complex applications.

The rest of this chapter is organized as follows: Section 4.2 describes the proposed method. Section 4.3 outlines the experimental evaluation of the partitioning method regarding resource allocation process time, partitioning effectiveness, and performances. Finally, Section 4.4 concludes this chapter.

		Workload - link load	Workload - final latency	link load - final latency
Pearson	linear	0.99	0.99	0.98
	sinus	0.68	0.31	0.26
	meth.	-0.74	0.76	-0.67
Spearman	linear	0.99	0.98	0.98
	sinus	0.61	0.33	0.2
	meth.	-0.59	0.51	-0.63
Kendal	linear	0.91	0.89	0.89
	sinus	0.48	0.24	0.1
	meth.	-0.50	0.37	-0.50

Table 4.1 – Correlation coefficient analysis between workload link load and final latency, comparing dummy linear correlation, dummy sinus correlation, and a method that distributes loads randomly to deploy RFI filter on a 3-nodes 3-cores homogeneous architecture across 50 simulations

4.2 Workload-balanced Iterative Distribution

Table 4.1 illustrates the correlation between workload standard deviation, link load standard deviation, and final latency. In this study, we use a method that randomly distributes the same workload on a 3-node 3-core homogeneous architecture and compares the result with linear and sinusoidal simulation using the three main coefficient correlation methods: Pearson, Spearman, and Kendal. Correlation analyses show that increasing workload standard deviation significantly reduces link load standard deviation and notably increases final latency while increasing link load standard deviation moderately decreases final latency. Note that a small deviation implies an equitably distributed load. To maximize final latency, the proposed method in this chapter focuses on distributing the workload equitably.

The proposed method is organized into four main stages as illustrated in Figure 4.1. To begin, it performs node-level partitioning, dividing the graph into subgraphs, each associated with a specific node (Section 4.2.1). These subgraphs are carefully constructed to achieve a balanced workload distribution, taking into account the heterogeneity of the nodes. Subsequently, the method proceeds to thread-level partitioning, optimizing the allocation and scheduling of threads within each node (Section 4.2.2). Following this, the method undergoes simulation, covering executions within individual nodes as well as interactions between nodes (Section 4.2.3). This step is essential to confirm that previously

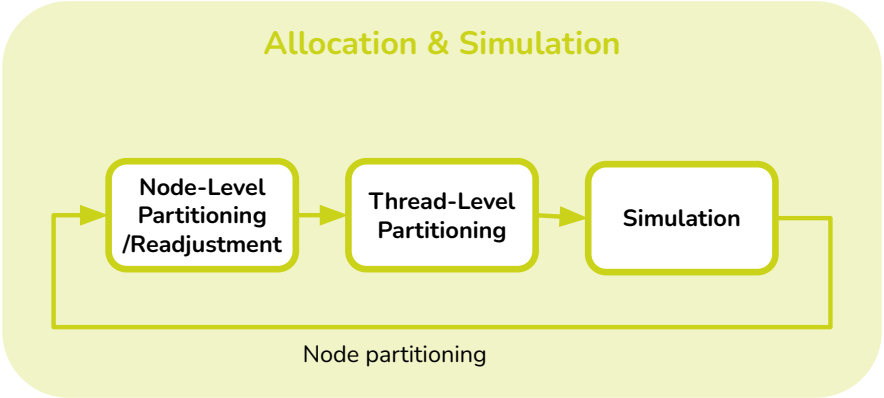

Figure 4.1 – Visualization of the SimSDP Procedure

overlooked communication factors do not negatively impact the final allocation and system performance. Lastly, the method allows for the potential readjustment of subgraph construction as needed (Section 4.2.4), making adjustments based on the simulation results to optimize the allocation and scheduling further. Upon successful mapping and scheduling validation, the method addresses the challenge of synchronizing and transferring data between system components, as discussed in Section 4.2.5. An overview of the method is illustrated in Figure 4.1. The main notations used in this thesis are listed in Table 4.2.

4.2.1 Node-Level Partitioning

This step aims to generate time-balanced subgraphs, each associated with one node outlined in Algorithm 1. To achieve this, the method relies on a metric representing the relative computational capacity offered by each node. This metric, called number of equivalent cores takes into account the core speed as a determining factor. The equivalent multi-node architecture is determined based on the slowest core type present within the multi-node architecture. The computation is represented by the following equations:

$$\begin{aligned} C_{eq} &= \textstyle\sum_{x=1}^{n} \mathrm{C}_x \times k_x \\ C_{eq_i} &= \textstyle\sum_{x=1}^{n} \mathrm{C}_{i_x} \times k_x \end{aligned} \tag{4.1}$$

Where C_{eq} represents the number of equivalent cores across all nodes, C_{eq_i} denotes the number of equivalent cores on the node with index i, C_x represents the number of cores

Table 4.2 – Summary of Key Notations

Notation	Description
C_{eq}	Number of equivalent cores on the overall architecture
C_{eq_i}	Number of equivalent cores on the node with index i
C_x	Number of cores running at a given operating frequency on the overall architecture
C_{i_x}	Number of cores running at a given operating frequency on the node with index i
k_x	Scaling factor for core (type) x
TC_{eq}	Cumulative sum of the execution of all graph actors if all run on the slowest core of the overall architecture
TC_{eq_i}	Cumulative sum of the execution of all actors in index subgraph i if all run on the slowest core of the overall architecture
$\mathbf{q}(a)$	Coefficient of the RV associated with actor a
$T_{coreSlow}(a)$	Execution time of one instance of actor a on the slowest core of the architecture
σ_i	Deviation of node i's workload from total workloads per node
L_i	Workload of node i
$Node_i$	architecture node of rank i of the set of nodes organized in descending order of complexity
Sub_i	Subgraph associated with architecture node of rank i of the set of nodes organized in descending order of complexity

running at a given operating frequency across all nodes, C_{i_x} the number of cores running at a given operating frequency on the node with index i, k_x is the scaling factor which is simply the ratio of the core operating frequency to the slowest core operating frequency of the overall architecture, and n is the number of operating frequencies of system cores.

Example 25. For example, consider an architecture comprising 3 nodes. The first node consists of 4 cores, with 2 cores operating at twice the frequency of the slowest core of the overall architecture. The second node has 6 cores, with 2 cores operating at 2 times the frequency of the slowest core of the overall architecture. The last node is composed of 6 cores, with 4 cores operating at 2 times the frequency of the slowest core of the overall architecture. In this case, the equivalent architecture is represented as (Ceq_0, Ceq_1, Ceq_2) = (6, 8, 10) instead of (4, 6, 6), and Ceq is 24. The equivalent architecture accounts for the varying core speeds across nodes, resulting in a modified configuration with different numbers of equivalent cores for each node.

The number of equivalent cores metric is used to compute a second metric called the cumulative equivalent time. This second metric estimates the theoretical execution time of the subgraph on the associated node where all cores are 100% utilized to execute the application. At this stage, the subgraphs are not yet built. The computation is represented by the following equations:

$$\begin{aligned} TC_{eq} &= \sum \mathbf{q}(a) \times T_{coreSlow}(a) \\ TC_{eq_i} &= TC_{eq} \times C_{eq_i}/C_{eq} \end{aligned} \tag{4.2}$$

Here, TC_{eq} denotes the cumulative sum of the execution of all graph actors if all run on the slowest core of the multi-node architecture. $\mathbf{q}(a)$ represents the coefficient of the RV associated to actor a, and $T_{coreSlow}(a)$ is the execution time of one instance of actor a on the slowest core of the architecture. TC_{eq_i} represents the cumulative equivalent time estimating for later subgraph execution time on the node with index i. This ensures that the actors are distributed equitably among the subgraphs based on their cumulative execution times.

The methods sequentially construct a subgraph for each architecture node of the considered architecture. Hence, the order in which nodes are considered has an impact on resource allocation. There are $n!$ possible architecture node orderings depending on the order in which the nodes are positioned, where n is the number of nodes. We refrain from testing all combinations and, for simplicity, we have set the order of subgraph construction in descending order of node performance. The nodes are ranked according to the number of equivalent cores C_{eq}, the first node being the one with the highest number of core equivalents.

Example 26.

For instance, considering the equivalent architecture (Ceq_0, Ceq_1, Ceq_2) = (6, 8, 10). The method arranges nodes in order of performance as follows (Ceq_0, Ceq_1, Ceq_2) = (10, 8, 6). The method constructs three subgraphs, each associated with a specific node. The equivalent cumulative time associated with $Node0$ for constructing $Sub0$ is calculated as: $TC_{eq_0} = TC_{eq} \times 10/24$. Similarly, the equivalent cumulative time associated with $Node1$ for constructing $Sub1$ is determined as: $TC_{eq_1} = TC_{eq} \times 8/24$. For $Node2$ and the construction of $Sub2$, the remaining cumulative equivalent time is allocated as: $TC_{eq_2} = TC_{eq} \times 6/24$.

These calculations ensure that each subgraph is assigned a proportional amount of cumulative equivalent time based on the number of equivalent cores in the architecture

associated with each node. The next step involves dividing the graph at the actor instance level using three patterns illustrated in Figure 4.2, which offer greater flexibility in actor distribution:

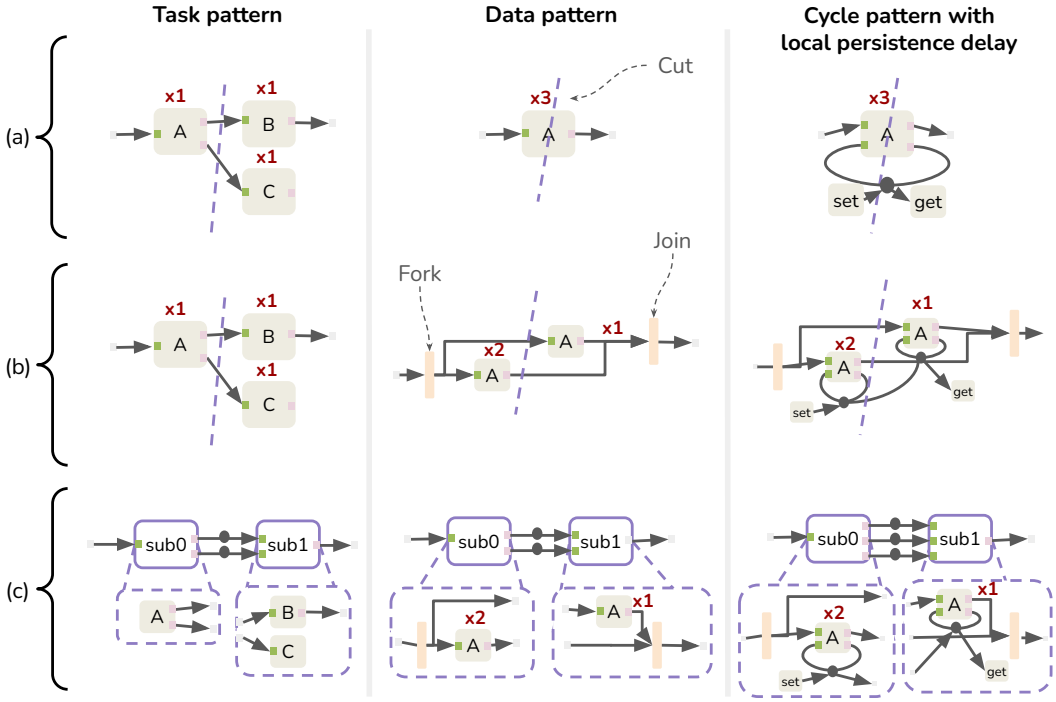

Figure 4.2 – Details of the Node-Level Partitioning Patterns Management: (a) Identification, (b) Task Division, (c) Subgraphs Generation

— Task pattern: An actor sequence with no internal state and a RV **q** of 1.

— Data pattern: An actor with no internal state and a RV **q** greater than 1.

— Cycle with LD: An actor with an internal state, meaning at least one of its outputs is connected to one of its inputs, and a RV **q** greater than 1. To ensure cycle consistency, a delay with initial tokens is added. While different degrees of persistence can describe a delay, our method allows only dividing the cycle with LD, where the delay is reset at each graph iteration. Other cycles cannot be sequenced.

Subsequently, the process constructs the subgraphs by incrementally adding actors in a pseudo-topological As Soon As Possible (ASAP) order, which is a scheduling approach commonly used in dataflow programming. In ASAP order, tasks are scheduled as early as possible without violating their dependencies, ensuring efficient and timely execution

of the dataflow [LM87b]. In our case the method starts by ranking the SDF actors in the topological order, starting with the source actors at rank 0. Subsequently, the rank is incremented, and a secondary list of actors associated with this rank is created. Actors are included only if they have no predecessors outside the previously established lists. The process is repeated until all actors are included. In our context, the actors within each topological rank are arranged in descending order based on their execution duration until the target time per subgraph TC_{eq_i} is achieved. As this target time approaches, the method checks whether the joining actor is part of the divisible pattern list; otherwise, the entire instance is incorporated. If it is a pattern, then the method performs a graph transformation consisting of splitting the instances and distributing the tokens via special actors (fork, join) shown in orange in Figure 4.2. This approach ensures an effective and optimized distribution of actors among subgraphs.

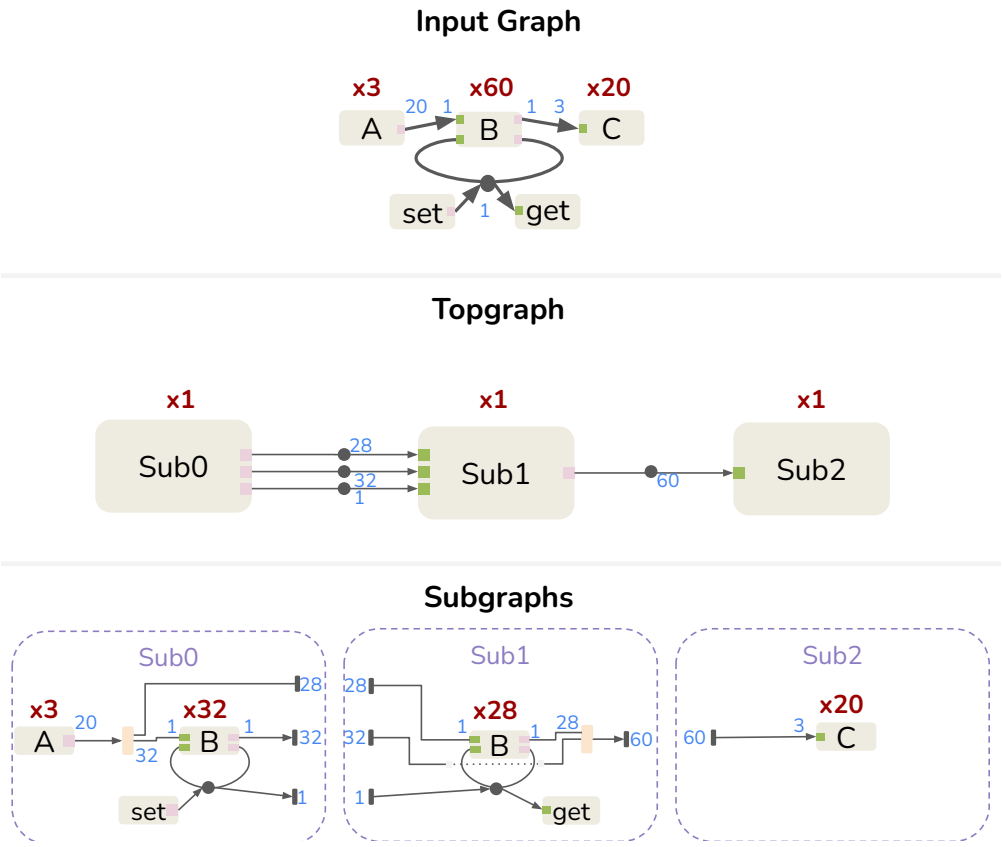

Figure 4.3 – Representation of the Node-Level Partitioning Graph Transformation

Example 27.

For example, with the equivalent architecture (Ceq_0, Ceq_1, Ceq_2) = (10, 8, 6) and the input graph depicted in Figure 4.3, assuming each actor has an equivalent execution time, 100-time units each, we get: $TC_{eq} = 8300$, $TC_{eq_0} = TC_{eq} \times 10/24 \simeq 3458$, $TC_{eq_1} \simeq 2766$, and $TC_{eq_2} = 2075$. After adding the actors in topological order, we get the following subgraphs: $sub0 = 3A + 32B$; $sub1 = 28B$; $sub2 = 20C$. The method then constructs the subgraphs based on these actor assignments.

At this step, the aim is to align the dataflow model to the hardware pipeline. A hardware pipeline is a computational design that divides tasks into sequential stages, with each stage being executed by a distinct hardware unit. It is widely employed in modern computer processors to accelerate the execution of instructions by breaking them down into discrete steps, such as fetching, decoding, executing, and writing back results. This approach allows for concurrent processing of multiple instructions, enhancing computational efficiency. Similarly, the dataflow pipeline is a model that mirrors this hardware philosophy. Our method integrates the work of [Hon+20] to automatically insert pipelines into a dataflow graph. It consists of breaking the dependencies in the graph and dividing it into several simultaneous executable stages. This process is achieved by inserting delay on strategic FIFO. These delays initially hold tokens in the first iteration, which are then carried over from the previous iteration. The method considers the worst-case execution of each actor and leverages estimated execution times, organizing actors in topological order using both ASAP and As Late As Possible (ALAP) schedules. By averaging the results obtained from these two schedules, the method estimates the overall execution time. This introduces controlled delays within the topgraph FIFOs, ensuring seamless and synchronized communication between the subgraphs within the pipeline structure of the architecture. Having completed this initial step, which enabled a coarse-grain separation of the graph across the available nodes, we will now proceed to achieve a finer distribution of each thread within each node.

4.2.2 Thread-Level Partitioning

The process of thread-level partitioning comprises two graph transformation phases, tailored to adjust the granularity of each subgraph to match the multi-core architecture of the associated nodes. The initial step is referred to as the Euclidean [Euc00] transformation, followed by the subsequent SCAPE transformation. The algorithm is outlined in Algorithm 2.

The Euclidean transformation involves the identification of dataflow actor a exhibiting

Algorithm 1 Node-Level Partitioning

Inputs: A Dataflow graph $G = \langle A, F \rangle$ with A a set of actors and F a set of FIFO. And a heterogeneous multi-node & multicore architecture $\Lambda = \langle N, C_x, k_x \rangle$ with N a set of nodes, C_x and k_x defined in Table 4.2
Outputs: Subgraphs $sub_x = \langle A_x, F_x \rangle$ associated with each node N, and a topgraph $top = \langle A_{top}, F_{top} \rangle$
Step 1:
$(C_{eq}, C_{eq_i}) \leftarrow equivalentArchitecture(\Lambda)$ using equations 4.1
Step 2:
$(TC_{eq}, TC_{eq_i}) \leftarrow cumulativeEquivalentTime(C_{eq}, C_{eq_i}, \Lambda)$ using equations 4.2
Step 3:
$N_i \leftarrow ArchitectureNodeOrdering(C_{eq})$
$A_{topologic} \leftarrow actorTopologicalOrdering(A)$ using pseudo-ASAP order
$A_{topologic_{rank}} \leftarrow actorRankOrdering(A)$ using decreasing actor execution time order
Step 4:
*/*Subgraph Construction*/*
for each $N_i \in N$ **do**
    sum of timing: $Tim = 0$
    for $a \in A_{topologic_{rank}}$ **do**
        while $Tim \leq TC_{eq_i}$ **do**
            if $a \in patternList$ 4.2 **then**
                $A_{topologic_{rank}} \leftarrow divide(a)$
            end if
            $sub_i \leftarrow add(a)$
            $Tim+ = Tim(a)$
        end while
    end for
end for
Step 5
*/*Topgraph Construction*/*
$top \leftarrow pipeline(F_{top})$ using the approach described in [Hon+20]

a parallelism degree exceeding the number of equivalent cores of the node on which they are deployed. Furthermore, such an actor undergoes an Euclidean division based on the actor RV, by the equivalent number of cores, resulting in a remainder greater than 0. This transformation divides the actor a into two distinct actors a_0 and a_1. The first actor encompasses a RV, denoted as $\mathbf{q}(a_0)$, calculated as:

$$\mathbf{q}(a_0) = \text{Remainder}(\mathbf{q}(a) \div C_{eq_i}) \tag{4.3}$$

Meanwhile, the second actor RV, represented by $\mathbf{q}(a_1)$, is computed as:

$$\mathbf{q}(a_1) = Quotient(\mathbf{q}(a) \div C_{eq_i}) \times C_{eq_i} \tag{4.4}$$

This division is essential to leverage the *scaling* of the next transformation.

Example 28. Consider an actor C with a repetition vector (RV) coefficient of 20, intended to be deployed on a multi-core architecture with 6 cores. The thread-level partitioning process divides this actor into two actors. Starting from the Euclidean division which gives us: $20 = 6 \times 3 + 2$.

The first created actor, C_0, is assigned an RV of 2, while the second created actor, C_1, is given an RV of 18. Subsequently, the SCAPE transformation is applied to scale the second actor, C_1, to the target by clustering it. The clustering of C_1, denoted as Ω_C, is configured to have an RV of 6.

Each instance of Ω_C is then mapped to a core of the target multi-core architecture. Notably, the cluster Ω_C includes 2 repetitions of actor C in its definition. This streamlined process simplifies the mapping procedure, reducing the initial complexity of mapping 20 instances of the actor to just 8 instances for this specific case.

The SCAPE transformations, introduced in [Ren+23a; Ren+23b; Ren+24a], are clustering techniques aimed at automatically adapting parallel computations granularity on a node. In our case, this method aligns the parallelism of the application description with the number of equivalent cores on the node, denoted as Ceq_i. It addresses excessive data parallelism by grouping data parallel actors and adjusting token production and consumption rates to align the RV coefficient relative to the number of cores in the target architecture, also referred to as *scaling*. Additionally, it enhances parallelism in dataflow actor sequence through pipelining and aligns the number of stages with the number of cores in the target architecture. A specific sequential pattern manifests as a cycle, SCAPE only considers cycles initiated by local delay, wherein the introduction of parallelism in-

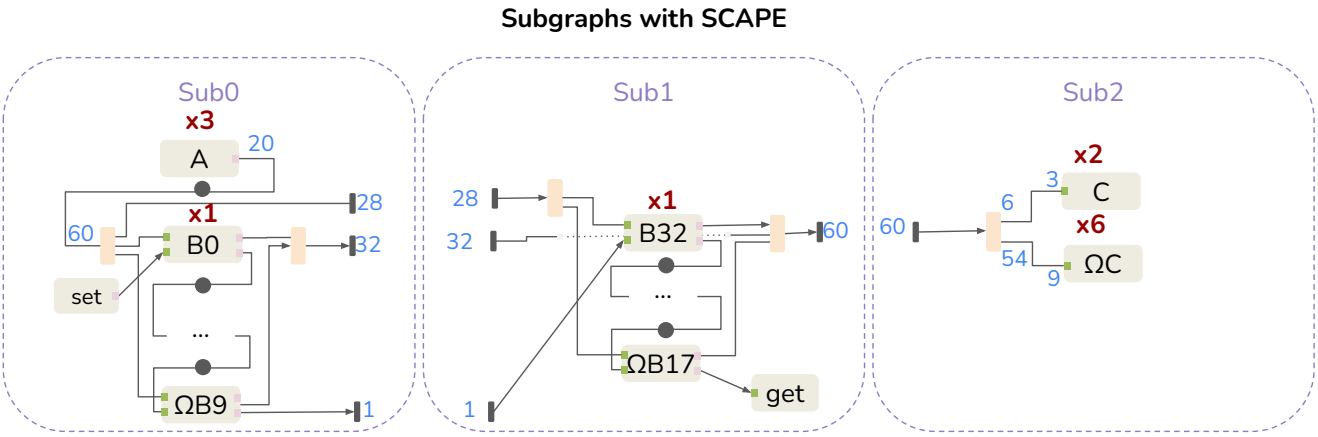

Figure 4.4 – Illustration of the SCAPE transformation applied on the generated subgraphs illustrated in Figure 4.3

volves the process of loop semi-unrolling. This entails dividing the original loop into n smaller loops, forming n pipelined clusters, where n represents the number of equivalent cores. The other cycles are left as is.

Example 29. For example, we consider each generated subgraph from Figure 4.3 along with their corresponding equivalent architectures: $(Ceq_0, Ceq_1, Ceq_2) = (10, 8, 6)$.
To align with our objectives, our method adjusts the data parallelism of actor C, as detailed in Section 4.2.2, by reducing its RV coefficient to match the parallelism of the target. Similarly, we enhance the parallelism of the dataflow actor sequence containing actor A—repeated 3 times—to match 10 equivalent cores. This adjustment involves inserting delays after actor A.
We employ a technique known as semi-unrolling on actor B. For instance, in subgraph 0, where B iterates 32 times on 10 equivalent cores, we first apply the Euclidean transformation, yielding $32 = 3 \times 10 + 2$. This results in 10 clusters of 3 cycles of B and 2 remaining B iterations. Subsequently, we pipeline each cluster and the 2 remaining B iterations by introducing delays between them. This process is facilitated by the original cycle of B, which starts with a local delay.
The resulting transformed subgraphs are depicted in Figure 4.4, with their granularity corresponding to their respective associated nodes.

Once the granularity has been adapted for each node, the process proceeds with a standard resource allocation. For our tests, we use the Critical Path Network (CPN)-dominant list scheduling method [AK97].

Algorithm 2 Thread-Level Partitioning

Inputs: Subgraphs $sub_i = \langle A_i, F_i \rangle$ with A_i a set of actors and F_i a set of FIFO. And heterogeneous associated node architecture $N_i = \langle C_{eq_i} \rangle$ with C_{eq_i} defined in Table 4.2
Outputs: Resource allocation per node γ_i
Step 1:
for each $a \in Sub_i$ **do**
    if $\mathbf{q}(a) > C_{eq_i}$ **then**
        $Sub_i \leftarrow EuclideanDivide(a, C_{eq_i})$
    end if
end for
Step 2:
for each $a \in Sub_i$ **do**
    $Sub_i \leftarrow SCAPETransform(a, Ceq_i)$
end for
Step 3:
$\gamma_i \leftarrow ClassicResourceAllocation(sub_i)$ using CPN-dominant list scheduling [AK97]

4.2.3 Intra/Inter-Node Simulation

To simulate the execution of the deployed SDF graph upon the multi-node architecture, we use two tools: PREESM for intra-node simulations and SimGrid for inter-node simulations. The primary objective of these simulations is to investigate whether the communication load influences the efficiency of the current distribution. To do so we evaluate the communication rate which quantifies the speed at which data is exchanged between different components or nodes [Sha48]. The communication rate is generally faster when exchanging data within a node compared to between two nodes due to factors such as propagation delays, interference, and signal attenuation caused by the physical distance between nodes [MC09]. The intra-node communication rate is determined by the maximal memory bandwidth of the target, calculated as the product of the bus width and the clock frequency of the target. On the other hand, the inter-node communication rate is directly related to the interface data rate and the communication network used.

PREESM: Intra-Node Simulation The current version of the tool [1] serves as an open-source rapid prototyping tool, offering robust analysis for heterogeneous multi and many-core single-node targets. Our method involves partitioning subgraphs associated with a node, enabling the tool to analyze each of these partitions. PREESM simulates the execution of the subgraphs on their respective node, and the resulting latency is treated

1. `https://preesm.github.io/`

as the execution time of the corresponding actor within the topgraph.

SimGrid: Inter-Node Simulation SimGrid [Cas+14] is an open-source framework specifically designed to model and simulate distributed algorithms on distributed IT architecture. Its purpose is to enable researchers to study distributed algorithms without the need for physical deployment, making it a valuable tool for prototyping and evaluation in diverse distributed environments. In this work, SimGrid, whose capabilities have been extended to handle dataflow graphs by IRISA, plays a crucial role in simulating inter-node communication to provide workload information. Using the topgraph containing data volume information for node-to-node transfers within the architecture we build the SimGrid simulation. Each topgraph actor represents a subgraph associated with a specific node. In addition, the mapping and scheduling details and execution times for the topgraph actors on the architecture nodes, provided by PREESM, are used to set up the simulation. Then, SimGrid performs multi-node simulation to analyze and assess the behavior and performance of the distributed system.

4.2.4 Node-Level Readjustment

The previous simulation is used to return performance and workload distribution information and is considered for potential rebalancing. As communication costs are not considered when computing the cumulative equivalent time TC_{eq_i} for constructing the subgraphs, the SimGrid simulation reveals their influences.

The Node-Level Readjustment step follows the same structure as the Node-Level Partitioning introduced in Section 4.2.1. However, during the construction of subgraphs, the definition of the cumulative equivalent time estimate TC_{eq_i} is re-evaluated, considering the workload deviation between nodes in the previous simulation.

$$\begin{aligned} \sigma_i &= L_i - \bar{L} \\ TC_{eq_i}(t) &= (1 - \sigma_i) \times TC_{eq_i}(t-1) \end{aligned} \tag{4.5}$$

Where σ_i represents the deviation in workload on node i, L_i is the workload of node i, and $\bar{L}$ stands for the average workload. Additionally, $TC_{eq_i}(t)$ denotes the current cumulative equivalent time targeted by the subgraph associated with the node indexed by i at time t. Meanwhile, $TC_{eq_i}(t-1)$ refers to the cumulative equivalent time computed at the previous iteration of the method.

Example 30. As an illustration, let's examine SimGrid feedback indicating node workloads

as $100\%; 85\%; 57\%$, considering the equivalent architecture (Ceq_0, Ceq_1, Ceq_2) = (10, 8, 6). The average workload is consequently computed as 80%. Subsequently, we determine an excess load per node, expressed as $20\%; 5\%; -23\%$.

The method proceeds by computing the equivalent cumulative time and subtracting the calculated excess from this time. This process results in the construction of subgraphs with a more precise representation of actor load.

The process then loops back to the thread-level partitioning step 4.2.2, readjusting the granularity and calculating the scheduling placement. The process iteration can be customized by specifying one of the following parameters: a fixed number of iterations, a target load distribution, or a target latency.

4.2.5 Code Generation

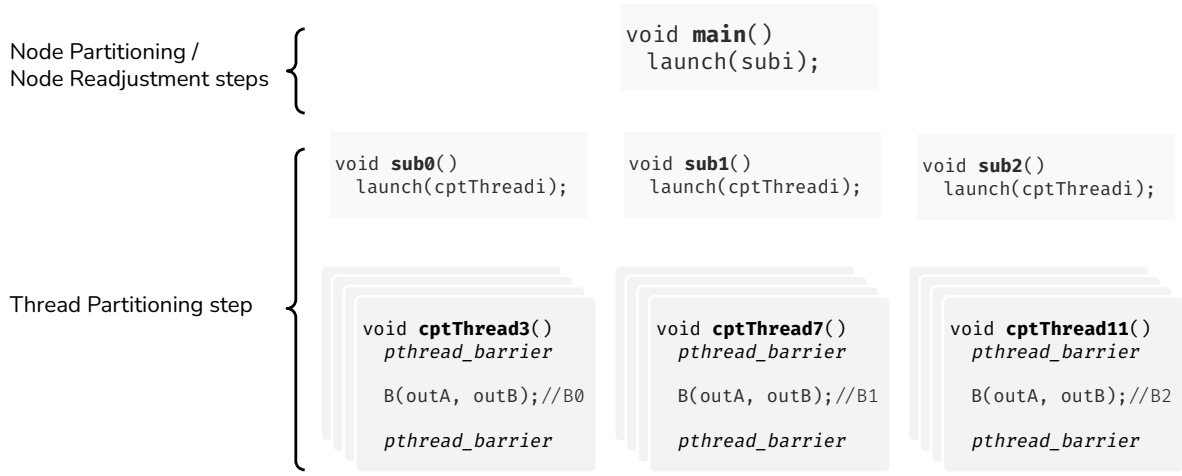

Figure 4.5 – multi-node and multi-core Code Generation Structure

In the subsequent section, once the mapping and scheduling of tasks have been validated, the focus shifts toward the critical aspect of synchronizing and transferring data between all the elements involved. Communication and synchronization between nodes are supported by the MPI library. On the other hand, synchronization between threads residing on the same machine is managed using the Pthread library. Threads within the same device communicate with each other using shared memory, while each node has its own distributed memory. The resulting files generated by SimSDP are organized into three distinct categories of files: the *main* file, the *sub* files, and the *core* files. The *main* file

contains the information required for launching nodes. Each machine within the system receives identical code files. To distinguish between machines, the MPI library assigns a unique identifier to each one. Based on this identifier, a target node function is executed. Moving down to the *sub* files, we retrieved the standard single-node code generation approach from PREESM. These files contain information for launching threads on each node core. Finally, in the third category of files, distinct C files are generated for each target core.

4.3 Experiments

4.3.1 Experimental Setup

Node ID	Node name	PE used	Clock frequency	Intranode max bandwidth	Internode max bandwidth
0	Paravance	3	2 to 4 GHz	472 Gb/s	9.42 Gb/s
1	Paranoia	3 to 6	2 to 4 GHz	477,6 Gb/s	9.42 Gb/s
2	Abacus	3 to 12	2 to 4 GHz	1351,68 Gb/s	9.42 Gb/s

Table 4.3 – Node characteristic

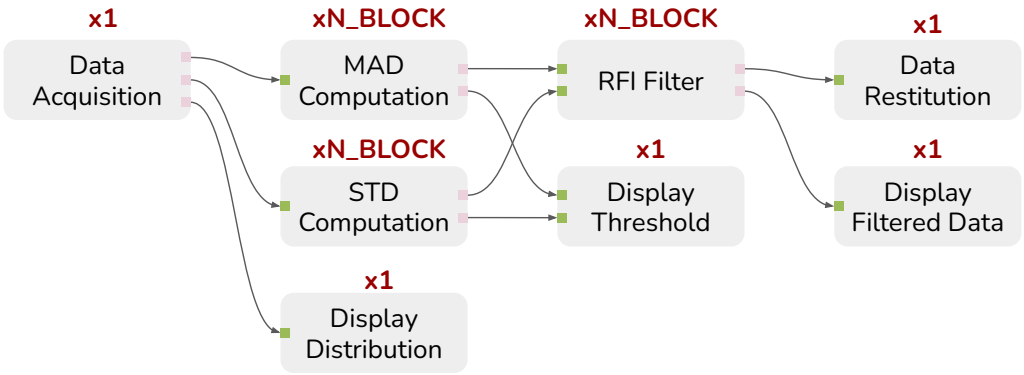

Figure 4.6 – RFI filter simplified dataflow model

The proposed method is applied to deploy the RFI filter dataflow model represented in Figure 4.6. The RFI filter [2] has been represented as a PiSDF graph using PREESM.

2. https://github.com/Ophelie-Renaud/MAD-based-RFI-mitigator

It takes a complex data file as input, extracts its imaginary and real parts, calculates thresholds using the Median Absolute Deviation (MAD) and standard deviation method concurrently, applies the chosen filter, and displays relevant curves during this process before reconstructing the filtered complex data file. The model is parameterized, enhancing its flexibility to handle various data sizes for processing. Two dataset sizes are examined to demonstrate the relevance of the method: one with 1.565 MB and another with 4.78 GB. The PiSDF graph of the RFI filter comprises 613 actors and 30 FIFO buffers. This graph is then flattened into an SrDAG by PREESM for analysis, mapping-scheduling, and code generation. The SrDAG for the smaller dataset consists of 1415 actors and 4016 FIFO buffers, while the SrDAG with the larger dataset has 1.4×10^7 actors and 4.0×10^7 FIFO buffers.

The deployment of the RFI filter takes place on three multinode target architectures, the key features of which are outlined in Table 4.3.

— $Archi_{\#1}$: This architecture consists of three homogeneous nodes. Each node comprises three x86 cores that share the same frequency, $f0 = 2GHz$.

— $Archi_{\#2}$: This architecture consists of three heterogeneous nodes. The first node is composed of three x86 cores, the second node consists of six x86 cores, and the third node comprises twelve x86 cores. All cores in this architecture share the same frequency, $f0 = 2GHz$.

— $Archi_{\#3}$: This architecture also consists of three heterogeneous nodes. The three nodes are composed of three cores, two cores operate at frequency $f0 = 2GHz$, and the other cores operate at frequency $f1 = 4GHz$.

Grid5000, introduced in [Cap+05], is an experimental research platform for large-scale distributed and parallel systems. The platform enables prototypes and new technologies to be validated in controlled, reproducible environments [Bal+13]. The produced prototypes are subsequently deployed on three nodes within the Grid5000 cluster in Rennes [3]. These nodes depicted in the Table 4.3 are Paravance, Paranoia, and Abacus. Our study inter- and intra-node simulations are based on the specific characteristics of these three nodes. The proposed method has been implemented into the PREESM rapid prototyping framework in open-source projects. The resource allocation processes are performed on a desktop computer with an 8-thread Intel i7-8665U processor and 31,2 GB of RAM.

Figure 4.7 provides a visual representation of the cumulative execution time for each step of the methods on a single iteration, evaluated on both the small (Figure a) and large

3. `https://www.grid5000.fr/w/Rennes:Hardware`

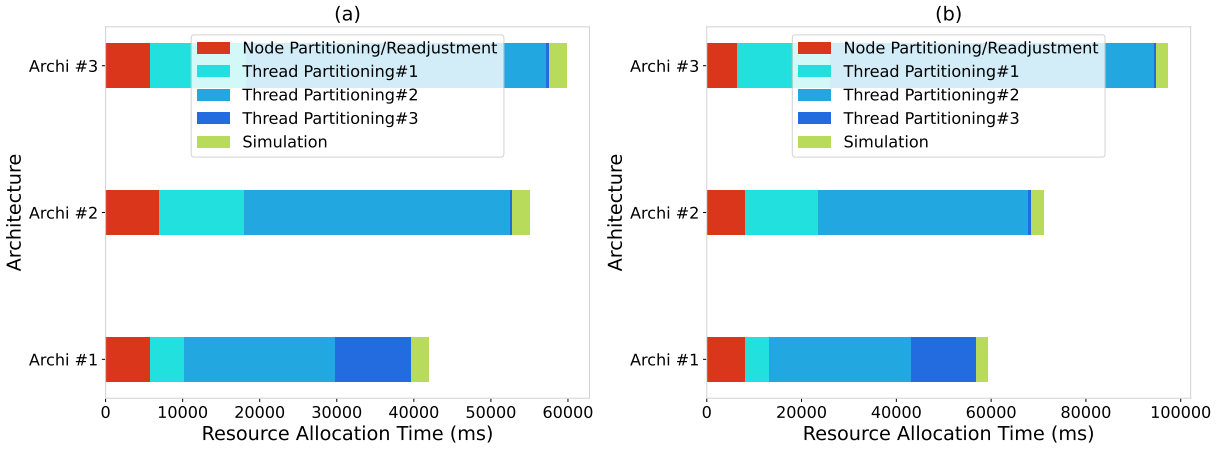

Figure 4.7 – Cumulative resource allocation time obtained (a) on the Small and (b) Large RFI Dataflow Models on the 1st Round

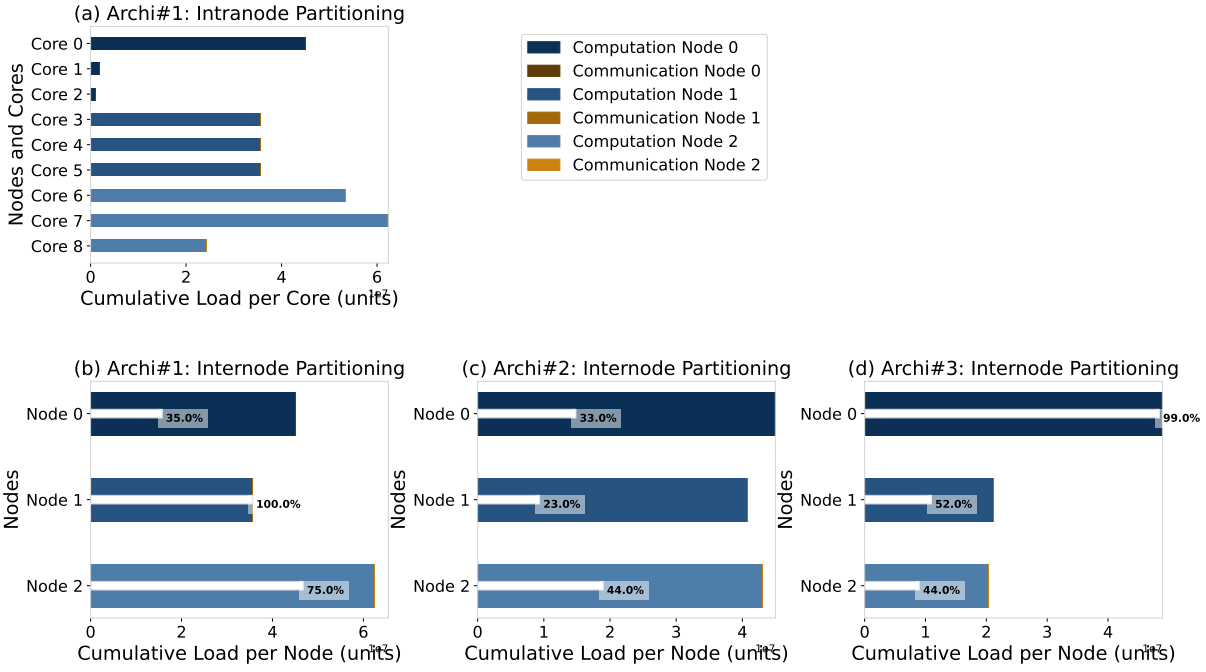

Figure 4.8 – Workload Analysis (a) on $Archi_{\#1}$, (b) on $Archi_{\#2}$, (c) on $Archi_{\#3}$: Intra and Inter-Node Performance for RFI Dataflow Model, highlighting dominant computations with minimal communication overhead

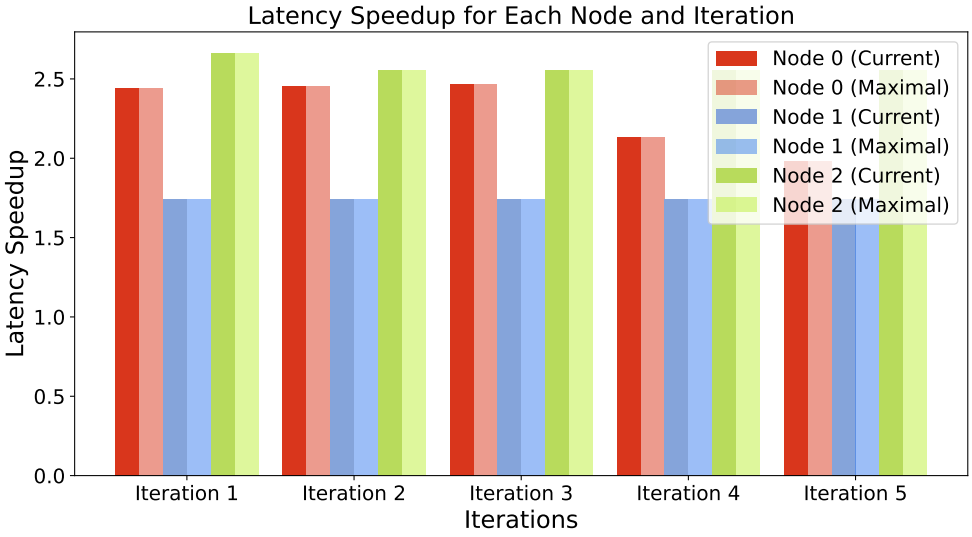

Figure 4.9 – Comparison of simulated (current) vs. theoretical maximum latency speedup for each node partition over five iterations of the method.

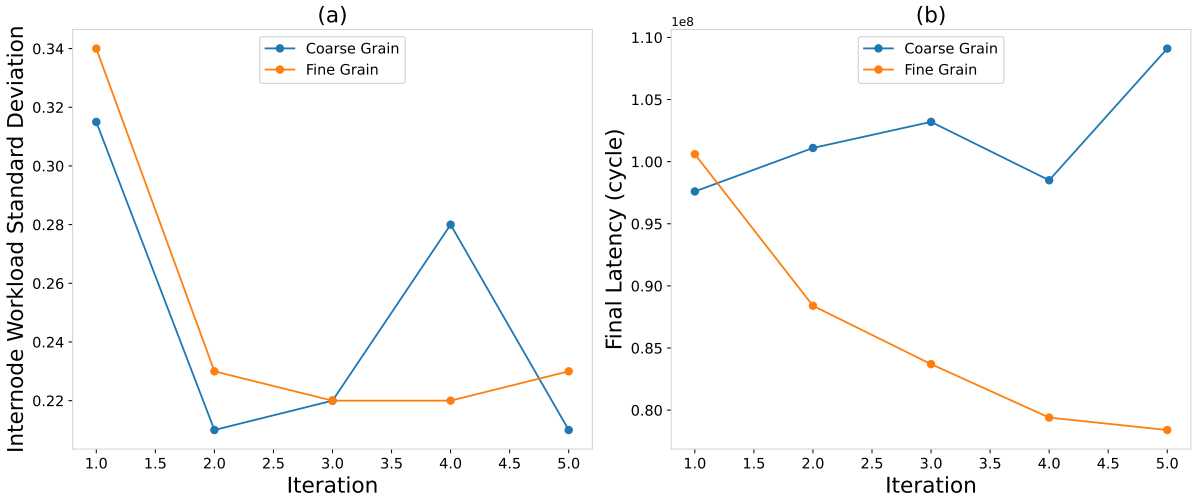

Figure 4.10 – Comparative Analysis of (a) Internode Workload Standard Deviation and (b) Latency on Homogeneous Architectures ($Archi_{\#1}$) across Fine (where communication time exceeds computation time) and Coarse (where computation time exceeds communication time) Granularities over Iterations

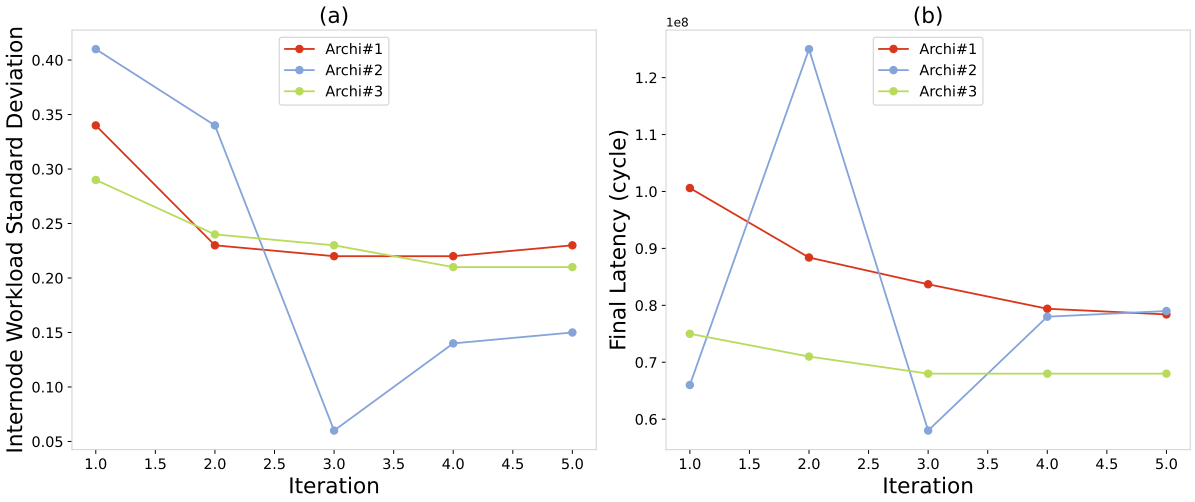

Figure 4.11 – Comparative Analysis of (a) Internode Workload Standard Deviation and (b) Latency across Three Architectures over Iterations

(Figure b) RFI dataflow models. The cumulative resource allocation, depicted in different colors, represents several key phases: the node partitioning step is represented in red, the thread partitioning phase (denoted in three shades of blue) corresponds to the allocation of resources to the three nodes, and the generation of subgraphs, and the simulation process is depicted in green. The cumulative resource allocation time is analyzed across three distinct multi-node architectures, as these architectures significantly impact placement decisions. Additionally, Figure 4.8 displays the workload partitioning within and between nodes on the RFI dataflow model with the small dataset given that the RFI dataflow model with the bigger dataset gives similar results with our partitioning method. This analysis takes into account the three multi-node target architectures, particularly when striving to minimize latency across five iterations. The upper figure provides a detailed breakdown of the intranode workload, which is summarized by a white percentage representing the proportion of intranode utilization on the lower figures. The chart delineates computation time in blue and communication time in orange. Figure 4.9 compares the latency speedup for each node partition over five iterations of the method between simulation and the theoretical maximum. The first node is represented in red, the second in blue, and the third node in green, with simulation results shown in dark colors and the theoretical maximum in light colors. This comparison demonstrates that the method achieves a distribution that matches the theoretically achievable maximum, thereby validating its effectiveness. Moreover, Figure 4.10 showcases the standard deviation of internode workload and latency

on the RFI dataflow model, focusing on the homogeneous target architectures across fine and coarse granularities during five iteration processes. Lastly, Figure 4.11 presents the standard deviation of internode workload and latency for the RFI dataflow model, observed across five iteration processes on the three multi-node target architectures.

4.3.2 Resource Allocation Time Analysis

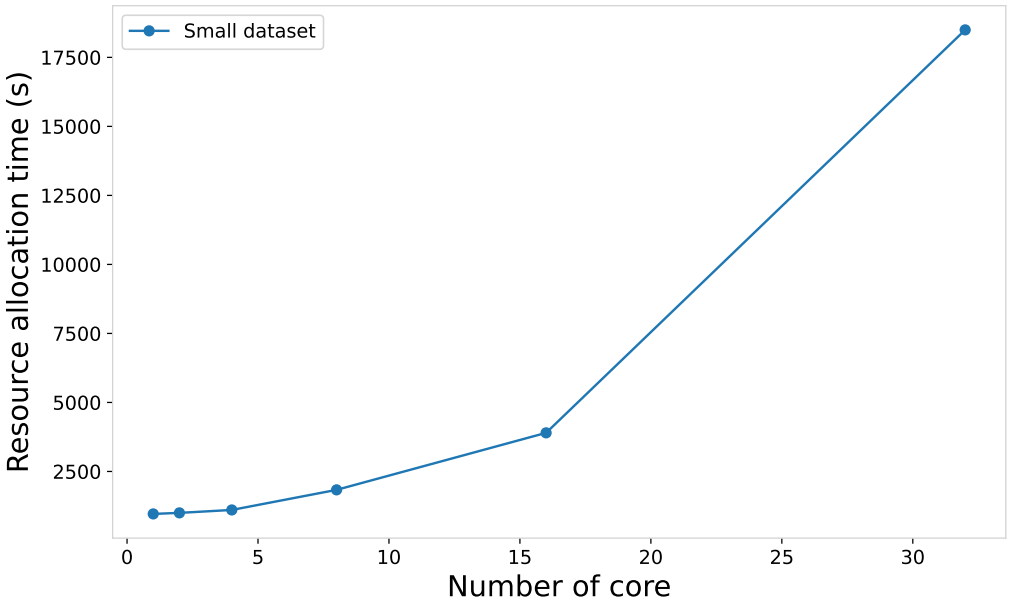

Figure 4.12 – Standard resource allocation method time on an increasing number of CPU cores on the RFI mitigator dataflow application

The experiment result depicted in Figure 4.7 shows that the cumulative resource allocation time is faster thanks to the method on multi-node target than the standard single-node resource allocation time illustrated in Figure 4.12 if we compare architecture with the same component number. This is because of the complexity of allocating resources with the standard dataflow method, which can be mathematically expressed as:

$$O(A.log(A) + P.(A + E)) \tag{4.6}$$

Where A is the number of actors or tasks that need to be allocated to processors, P is the number of available processors or computing resources and E represents the number of

edges or dependencies between actors. In scenarios where P and E remain constant, this expression exhibits linear complexity concerning A. However, if P and E fluctuate with A, the complexity may transition into a cubic form. In the worst-case scenario, when E equals A^2, and assuming that the number of actors A is proportional to P to ensure sufficient computation parallelism, we have $E \propto A^2 \propto P^2$, leading to a complexity dominated by P^3 in Equation 4.6. Consequently, this results in a cubic complexity for the last term in the general formula [FO95]. Our method, by dividing the original graph over the number of nodes, divides the complexity of the graph to be processed separately on the one hand. On the other hand, the thread-level partitioning process, which adjusts granularity via an actor clustering method, also reduces the complexity of the subgraphs. For instance, the standard single-node resource allocation requires 20 minutes for 1 core and 40 minutes for 8 cores for a small application. In contrast, one iteration of our method achieves this in between 1 and 2 minutes across various architectures: $Archi_{\#1}$ (9 elements), $Archi_{\#2}$ (21 elements), and $Archi_{\#3}$ (9 elements). The method accelerates the resource allocation process by a factor of 10 to 72 for deploying the RFI filter with a small data set on these three architectures compared to the standard methods. In the case of the larger application and without our method, the graph complexity maxes out the test Personnal Computer (PC) capabilities even with just one core. In contrast, our method remains unaffected by the complexity of the graph, consistently allocating resources within the expected timeframes. In this paragraph, we delve into a detailed analysis of the evolution of the processes involved in achieving various targets, as outlined in Section 4.2. The method encompasses several crucial steps, including node-level partitioning, thread-level partitioning, simulation, and node-level readjustment. Notably, the execution duration for node-level partitioning and readjustment remains consistently rapid, merely taking a few seconds, regardless of the graph complexity. Although their execution time may slightly increase with an expanding number of nodes, it remains low. The method is iterative so several iterations of the resource allocation and rebalance process can occur. In each iteration, the thread-level partitioning process runs as many times as there are nodes—3 times on each architecture in our context. The number of iterations is fixed by the user as a parameter.

4.3.3 Multinode Multicore Partitioning Analysis

Considering that the SCAPE method produces consistent configurations for the RFI filter dataflow model, whether applied to small or large datasets and when scaled to

diverse architectures, we have opted to depict a single model in Figure 4.8. This model is used to assess the allocation of intranode and internode workloads. Within subfigure (a), two graphs are presented. The upper graph presents the intranode workload distribution, encompassing both computation and communication times. A notable observation is that communication time is negligible compared to computation time for both intranode and internode configurations. This phenomenon is attributed to the SCAPE method capacity to reduce communication overhead through the precise alignment of parallelism. The lower graph in sub-figure (a) represents the distribution of the workload between nodes. The computation time associated with each node corresponds to the makespan of the subgraph relating to that specific node, which effectively characterizes the node final latency. At the same time, the percentage values shown next to each node indicate the intranode workload. Upon interpreting Figure 4.8, the minimum final latency over five iterations is not necessarily achieved when the distribution is optimal, as depicted in all sub-figures. In fact, according to Amdahl's law [Amd07], the workload distribution is constrained by the sequential portion of the workload. This implies that even with an optimal distribution of parallelism, the final latency might not reach its minimum due to the limitations imposed by the sequential part of the dataflow graph. The significance of this limitation becomes even more evident when considering the typical structure of most applications, which follows a recurring topology. These applications initiate and conclude with sequential, time-intensive source and sink actors. The core of parallelism is concentrated in the middle of the graph, showcasing actors with the potential for shorter execution times. This is illustrated in the lower graph of sub-figure (a) when deploying the RFI dataflow graph on $Archi\#1$, the 3-node homogeneous architecture. The illustration highlights the concentration of the parallelism of the application on the middle node of the architecture, namely, $Node_1$. This results in a perfectly distributed intranode workload of 100% and the lowest cumulative load per node. In contrast, the nodes located at the extremities, responsible for handling the source and sink actors of the dataflow graph, exhibit a low intranode workload and a high cumulative load. Concerning $Archi\#2$, the 3-node heterogeneous architecture concerning the number of cores per node, the method achieves a well-balanced internode workload. However, the intranode workload remains low due to the limitations imposed by Amdahl's law. Finally, in the case of $Archi\#3$, the best final latency is obtained when the first node concentrates a significant portion of the dataflow graph, incorporating the core parallelism of the application. As a result, the intranode workload reaches 99%, while the subsequent nodes receive the sequential sink

actors, leading to a low intranode workload.

Additionally, Figure 4.9 demonstrates that the method provides efficient internode partitioning, as the simulated final latency speedup on each node equals the theoretical maximum speedup. The maximum theoretical speedup is calculated as the ratio of work length to span length, where the work length is the sum of actor times on the main core, and the span length is the CPN time excluding communication overhead. However, due to the constraints of Amdahl's law, achieving a 3-core CPU node acceleration remains unattainable, suggesting that this architecture may be oversized for the given application.

4.3.4 Iterative Impact: Workload Deviation Trends and Latency Evaluation

The present results are obtained when the iterative process is fixed by 5 iterations to show workload standard deviation and latency trends. Given that the workload is represented as a percentage of node activity, in a well-balanced workload, the average tends to approach one and the deviation tends to approach zero. The experimental results in Figure 4.10 illustrate the influence of the granularity on the performance of the iterative process of the method. In this context, granularity is determined by intentionally adjusting communication speed to either accelerate or decelerate the data transfer. For applications where communication time is non-negligible in comparison to computation time, we consider this as having a fine granularity. In contrast, when communication times are negligible compared to computation times, we consider this as having a coarse granularity. Regarding the results, in coarse-grained description and right-sized architecture, the method readily finds a balanced distribution and minimal latency. Subsequently, over iterations, the latency fluctuates around this minimum. Conversely, when communication times become significant relative to computation times, with fine granularity, the algorithm benefits from iteration, leading to a descending trend until it achieves minimal latency. Since fine granularity influences the behavior of the iterative process, we proceed with the rest of the study on this granularity. The experimental results depicted in Figure 4.11 offer insights into the behavior of standard deviation on fine-grained description. On a sufficiently simple architecture, like $Archi_{\#1}$ and $Archi_{\#2}$, which can accommodate the entire graph without overuse, latency trends follow the deviation trend, showing improvement over iterations. However, when dealing with a larger architecture, we notice that achieving minimal latency involves concentrating workloads on only two nodes. To achieve

this, the method effectively prioritizes high-performing nodes by sorting them based on performance and actors. However, throughout iterations, the method strives for a more widespread workload distribution across the entire architecture. This pursuit results in an oscillating trend in latency. The reason behind this behavior lies in the transition of one end-of-graph actor from one subgraph to another. While this transition may appear excessive for building balanced subgraphs, it inadvertently introduces significant communication overhead in the subsequent iteration, contributing to the observed oscillations in latency.

4.4 Conclusion

This chapter introduces a new clustering approach called SimSDP, designed to deploy applications on heterogeneous multi-node and multi-core HPC architectures. The method employs a combination of divisive and agglomerative clustering techniques to adapt the dataflow graph effectively across the available nodes and the CPU cores within those nodes. It takes into account variations in the number of CPU cores per node, their individual speeds, and the transfer costs when making resource allocation decisions. SimSDP relies on two key tools: PREESM for resource allocation and SimGrid for simulation. The integration of these tools enables the method to produce valuable feedback for potential readjustments throughout the deployment process. By leveraging this hybrid approach, SimSDP demonstrates its capability to optimize resource allocation and enhance application performance on complex architectures. The method is the result of a collaboration between IETR, IRISA, and CSIRO to develop simulation tools and provide an application for processing astronomical signals and data. Experimental results show that the method accelerates the resource allocation process by a factor of 10 to 72 for deploying the RFI filter with a small data set on three multi-node and multi-core architectures compared to the standard methods and deals with both small dataflow graphs and more complex ones.

Chapter 5

Static Dataflow and Homogeneous Multi-Node, Multi-Core and Network Topology Co-design

5.1 Introduction & motivation

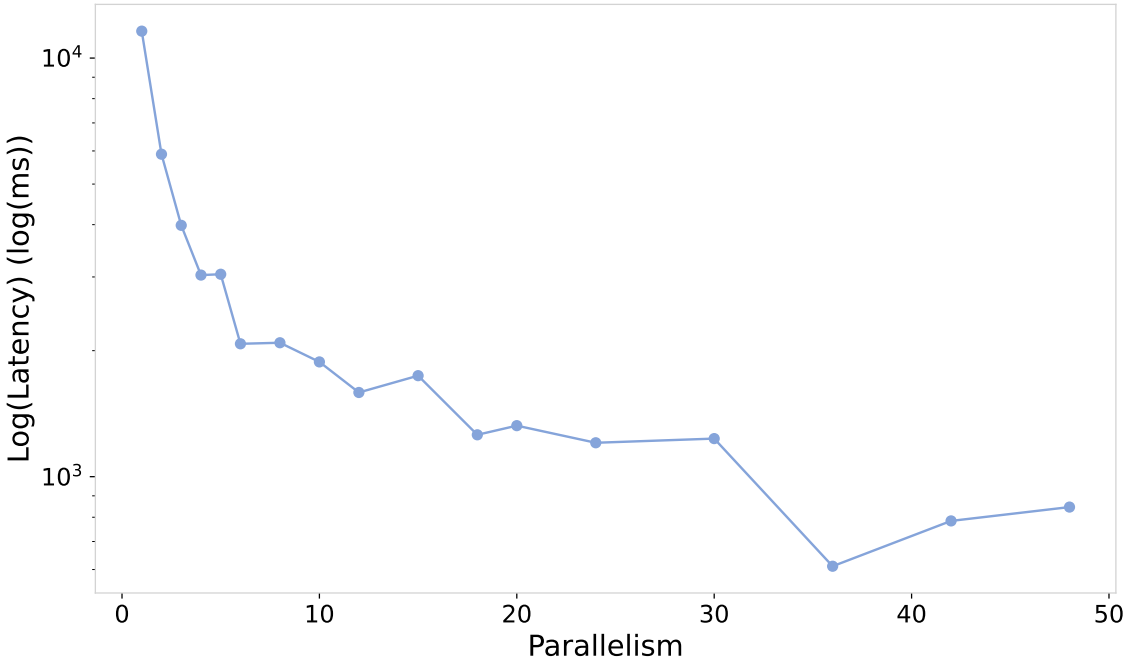

Figure 5.1 – Analysis of Final Latency in Sobel Application Deployment Across Various Architectures with Enhanced Parallelism

In the context of astronomical image processing, the main challenge is to minimize the final latency but, the relation between final latency and parallelism tends to asymptotically reach the parallel slowdown conforming to Amdahl's law [Amd07]. In practice, the trend

admits several local minimums due to mapping opportunities and task-to-core matching. Figure 5.1 illustrates the final latency of a Sobel application deployed on multi-core CPU architectures, with an increasing number of cores as the simulation advances. We obtain a global minimum on 36 cores, but also several local minimums on 4, 6, 12, and 18 cores.

This chapter presents an exploration framework for multinode multicore HPC systems. The proposed method allows to explore rapidly various network topologies to find the optimized solutions in terms of latency for a given application aware of the non-linearity of the trend. The proposed method facilitates the in-depth DSE to identify Pareto-optimal solutions in HPC systems. Experiments demonstrate that our method accelerates HW-SW co-design search by a factor of 3.7 compared to a semi-exhaustive method.

The rest of this chapter is organized as follows: Section 5.2 describes the proposed method. Section 5.3 outlines the experimental evaluation of the process for narrowing the hardware DSE scope regarding duration and effectiveness. Finally, Section 5.4 concludes this paper.

5.2 Algorithm and HPC system co-design process

The proposed approach is composed of two main steps. The process starts with an initialization step (5.2.1) to provide exploration boundaries for final latency and memory that will be used in the later section (5.2.2) to guide the user and prevent useless architecture configurations. For each architecture, the method allocates resources due to our previous work introduced in the previous chapter (Chapter 4) and performs simulations for four key metrics across five main network topologies (5.2.3). Figure 5.2 presents an overview of the method.

To encapsulate the exploration for an adequate architecture tailored to a given application, we define:

$$\alpha(N, C, T) \implies \rho(L_{\text{final}}, M, E, \hat{C}) \tag{5.1}$$

Where α represents the architecture parameterized by the number of nodes N, the number of cores per node C, and the network topology T. ρ denotes the Pareto optimal function, evaluating the final latency L_{final}, memory footprint M, energy consumption E, and system cost $\hat{C}$.

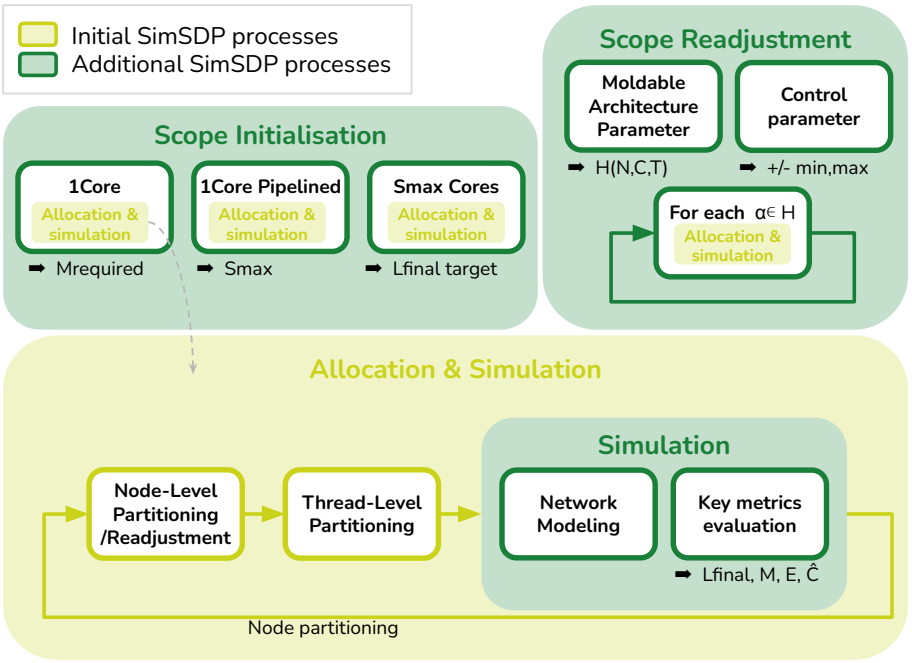

Figure 5.2 – Visualization of the SimSDP Co-Design Procedure

5.2.1 Scope Initialisation

The initial step aims to evaluate the boundaries of the hardware DSE scope. This step encompasses three first simulations. The first one simulates a single node and single core architecture estimating the minimal memory footprint which later fixes the minimal number of nodes to support the application execution.

The second one simulates the dataflow graph where as many pipeline stages as possible are introduced in the application model giving the maximum theoretical achievable speedup S_{max}. Our method integrates the work of [Hon+20] to automatically insert pipelines into a dataflow graph. The maximum achievable speedup is defined as follows:

$$S_{max} = \frac{\text{work length}}{\text{span length}} \tag{5.2}$$

The work length is defined as the sum of all actor times and the span is the length of the longest sequential path in the dataflow graph. In our case, the span corresponds to a theoretical deployment of the algorithm on an architecture with an infinite number of homogeneous cores without taking into account communication time.

The third one simulates the estimated maximal parallelism such as multiplying the

number of nodes by the core count equals the maximum achievable speedup $N \times C = S_{max}$. This step gives an idea of the potential best final latency $L_{\text{final}}(S_{max})$ to verify with the following iterative process.

Example 31. For example, let's consider the Sobel application, whose SDF graph is composed of 27 actors and 7 FIFO buffers. The first simulation on 1 core gives a memory footprint of 128 348 160 bits, and since there is no pipeline on this configuration the estimated speedup is 27 as shown in Figure 5.1 this is not a global minimum. Next, the simulation of the same application with pipelining on all edges and deployed on 1 core returns a maximum speedup estimate of 33, which is closer to the minimum. The 3rd simulation involves deploying this application on 33 cores, giving a final target latency of $8s$. These statistics will be used in the subsequent method process.

5.2.2 Scope Readjustment

The process proceeds to delineate the hardware DSE scope by establishing parameters for the number of nodes, and number of cores, iterating over this defined range. The *Scope Readjustment* process involves two key phases: the introduction of moldable architecture parameters and the subsequent control of these parameters.

Architecture Moldable Parameters

Architecture moldable parameters involve assigning a range of values to a parameter, allowing for the automation of allocating resources and simulating across all delineated architecture configurations feeding the Equation 5.1. This is achieved through fixing values for the number of nodes $(N_{\min}, N_{\max})$, number of cores $(C_{\min}, C_{\max})$, and network topology $(T_{\min}, T_{\max})$, each with a defined step size $(\Delta N, \Delta C, \Delta T)$. The hardware DSE scope, denoted as H, is mathematically expressed as:

$$\begin{aligned} H = \{ \quad & (N, C, T) \mid \\ & N_{\min} \leq N \leq N_{\max}, \\ & C_{\min} \leq C \leq C_{\max}, \\ & T_{\min} \leq T \leq T_{\max} \} \end{aligned} \tag{5.3}$$

Control Parameter

Moldable parameters are readjusted throughout the process to match the specific needs of the given application and to reduce the exploration time. Given that resource allocation is the most time-consuming step and needs recalculations whenever the number of nodes and cores changes, we focus on adjusting the scope of these two parameters: node number and core count ranges. The network topology range is set by the user without much influence on the search time. Therefore, the parameters to readjust are $N_{\min}$, $N_{\max}$, $C_{\min}$ and $C_{\max}$. $N_{\min}$ is readjusted considering the memory requirements of the application given by the initial step. The minimal node needed to support the entire application is defined as follows:

$$N_{\min} = \lceil \frac{M_{\text{Requirement}}}{N_{\text{Capacity}}} \rceil \tag{5.4}$$

Where $M_{\text{Requirement}}$is the memory requirement and N_{Capacity} is the node storage capacity.

Example 32. For example, let's consider the Sobel application from Example 31. The application requires 128 348 160 bits, and if the capacity of a node is 200 000 000 bits, then the minimum number of nodes to be iterated is at least 1.

For each simulated configuration, we determine the maximum level of parallelism achieved when the final latency stops decreasing for at least δ_α consecutive architecture configurations, despite increasing parallelism, and remains less than or equal to the potential best final latency $L_{\text{final}}(S_{\max})$. This ensures that we identify the global minimum of the final latency trend.

$$\begin{aligned} P_{\max} = \max \{ & N \times C \mid \\ & \exists i \geq \delta_I : L_{\text{final}}(P, i) \leq L_{\text{final}}(S_{\max}) \} \end{aligned} \tag{5.5}$$

Where $P_{\max}$ is the maximal parallelism, N the node number and C the cores count, $L_{\text{final}}(P, i)$ is the final latency at the iteration i with a parallelism P.

Example 33. For example, let's consider the Sobel application from Example 31. We set $\delta_I = 2$ iterations. The first iteration on 36 cores returns a final latency (600s) lower than the target latency (800s). and the next iteration, on 42 cores, returns a final latency (700s) lower than the target latency. The method then considers 36 to be the maximum parallelism and stops the search iteration.

5.2.3 Simulation

For each configuration defined as $\alpha(N, C, T)$, a model of architecture is automatically generated. The main goal is to analyze how these parameters affect different architectures. The architecture modeling considers these parameters as variables while keeping other parameters at default values that reflect real-world architectures.

Architecture Modeling

The architecture modeling includes two phases: multinode modeling and network topology modeling. In the multinode modeling phase, the structure is influenced by the number of nodes (N), and the number of cores (C). This structure impacts resource allocation and is modeled upstream in each iteration. In the network topology modeling phase, the structure is influenced by the number of nodes (N) within the five main topology families introduced in Chapter 1. Our focus is solely on comparing these five primary network topologies. Therefore, we propose generic models with basic routing for each, with the only variable being the number of nodes.

— $\text{Crossbar}_{\text{cluster}}$ and $\text{Backbone}_{\text{cluster}}$: These configurations are excluded if the number of nodes exceeds the number of ports on the router R.

— $\text{Torus}_{\text{cluster}}$: Characterized by the parameters "x, y, z " where x, y and z are the dimensions of the torus such as $N = x \times y \times z$. Such torus exists if N is divisible by $x \cdot y \cdot z$.

— $\text{Ftree}_{\text{cluster}}$: Defined by the parameters "$L, \mathbf{D}_{\text{link}}, \mathbf{U}_{\text{link}}, \mathbf{P}_{\text{link}}$" where L is the number of levels, $\mathbf{D}_{\text{link}} = \{N_{\text{pr}}, ..., com\}$ a vector containing the number of downlink for each level, where N_{pr} is the number of nodes or leaf per router and $com = \frac{N}{N_{\text{pr}}}$ is the level of communication. $\mathbf{U}_{\text{link}} = \{1, ..., com\}$ is a vector containing the number of uplink for each level, and $\mathbf{P}_{\text{link}} = \{1, ..., 1\}$ is a vector containing the number of parallel link for each level disabled in our case. The value of L is incremented according to the router capacity and the number of nodes. Such fat trees exists if the number of nodes is a power of two.

— $\text{Dragonfly}_{\text{cluster}}$: Defined by "$G, G_{\text{link}}; C, C_{\text{link}}; R, R_{\text{link}}; N_{\text{pr}}$" Where G is the number of group, G_{link} the number of link between group, C is the number of chassis per group, C_{link} the number of link between chassis, R the number of router per chassis, R_{link} the number of link between routers, N_{pr} is the number of node per router. We

compute $N = G \times C \times R \times N_{\text{pr}}$. Such a dragonfly exists if the number of nodes is even.

Key metrics

The final latency L_{final} is defined as the elapsed time between the initiation of the first task and the completion of the last task achieved when the application reaches its maximum throughput. It represents the total time taken for the entire dataflow graph to be processed, considering the concurrent execution of tasks on multiple cores.

The memory requirements M are determined by the cumulative size of individual FIFO buffers of the dataflow graph.

The energy consumption E is the sum of node-level energy and link-level energy. The node-level energy is calculated by combining dynamic power multiplied by node execution time and static power multiplied by node voltage. The product of transmission power, link distance, and link bandwidth determines the link-level energy.

The system cost is calculated by summing the individual elements associated with nodes, cores, routers, and links computed when modeling architectures and topologies.

5.3 Experiments

5.3.1 Experimental Setup

The proposed method is applied to three applications, including an RFI filter from the SKA project, along with two other image processing applications: Sobel and Squeezenet. This diverse set of applications underscores the broad applicability of the proposed method. The resource allocation processes are performed on a desktop computer with an 8-Core Intel i7-8665U processor and 31,2 GB of RAM. Some simulations are subsequently deployed on the multinode cluster *Parasilo* of the grid5000. The proposed method has been implemented into the PREESM rapid prototyping framework in open-source projects.

	Sobel	**RFI**	**Squeezenet**
Final latency	0,52 %	1,06 %	0,98 %
Memory	3,15 %	6,52 %	0,02 %

Table 5.1 – In vivo vs. in silico error analysis: deploying Sobel, RFI Filter, and Squeezenet on 5 Architectures

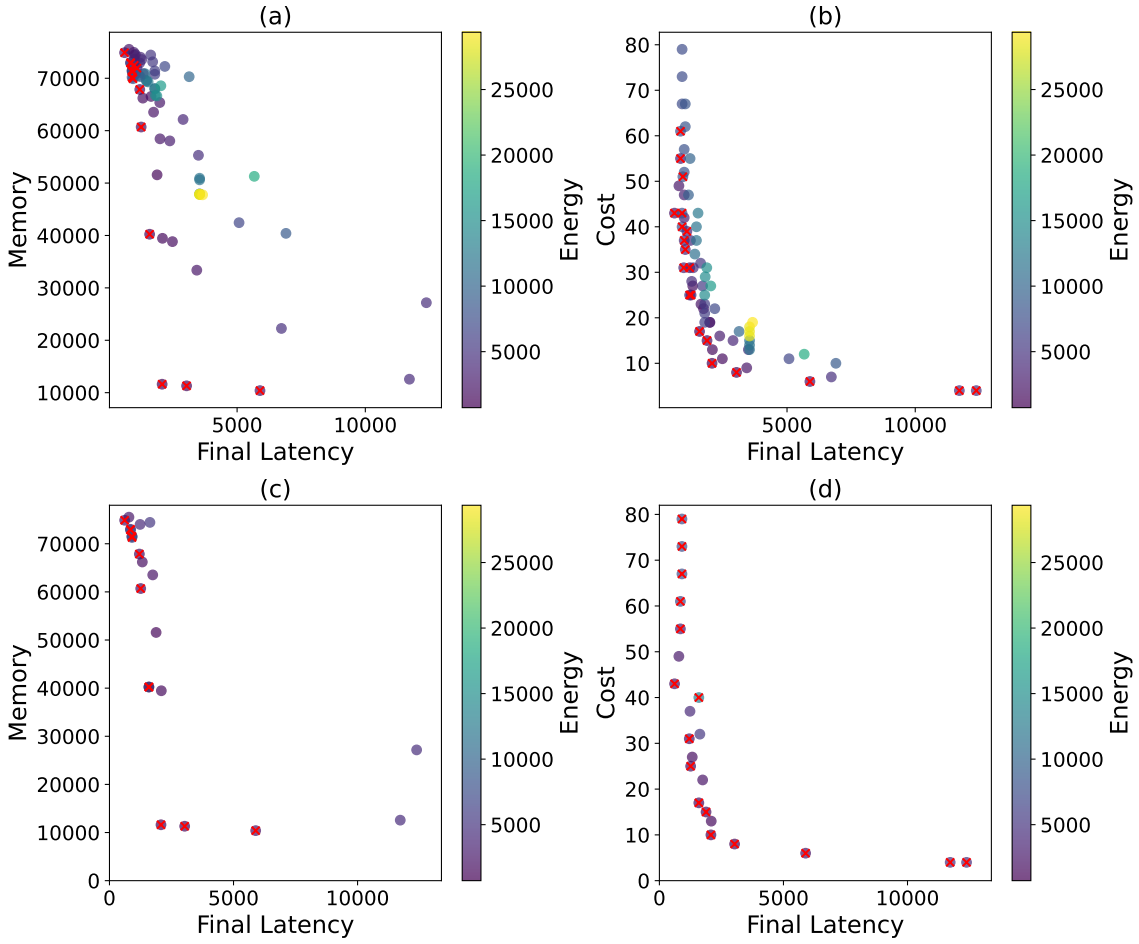

Figure 5.3 – Simulation deploying the Sobel dataflow algorithm on a range of architecture (a, b) without scope tuning. Simulation time = 52 min. 58 sec., and (c, d) with scope tuning. Simulation time = 14 min. 26 sec.

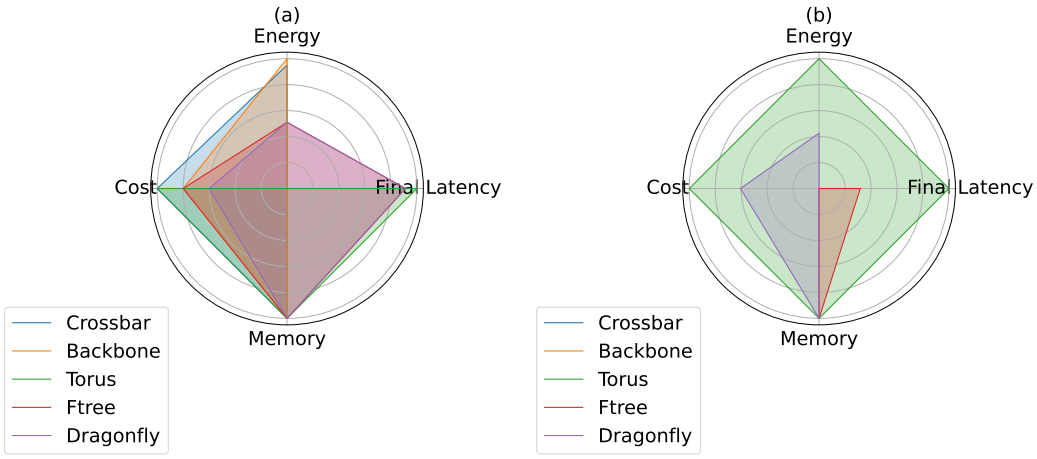

Figure 5.4 – Analysis of network impact on performance (a) on a 3-node architecture, (b) on a 24-node architecture

In Figure 5.3, the simulation results for deploying the Sobel dataflow algorithm are presented, considering final latency, memory, energy, and cost. The simulations are conducted across various architectures defined by input moldable parameters. The moldable parameters are define as follow: $\{N_{\min}; \Delta N; N_{\max}\} = \{1; 1; 12\}$, $\{C_{\min}; \Delta C; C_{\max}\} = \{1; 1; 6\}$, $\{T_{\min}; \Delta T; T_{\max}\} = \{1; 1; 5\}$. In Figure 5.3, Subfigure (a, b) illustrates the results without narrowing the hardware DSE scope, while subfigure (c, d) includes narrowing the hardware DSE scope. Pareto-optimal architectures are marked with red crosses. In Figure 5.4, Subfigure (a) compares the results of the simulation of the Sobel application for the best node number configuration across the five main topologies concerning the four metrics. The metrics are normalized and inverted to obtain the optimum at 1 in the radar chart. Subfigure (b) compares the results of the simulation of the Squeezenet application for the best node number configuration across three out of the five main topologies, two of them do not support large-scale architecture. Additionally Table 5.1 provides a summary of the average Mean Squared Error (MSE) between the ratio of values obtained on a single core versus the values obtained on the tested architecture. These values are simulated and measured across the five best Pareto optimal architectures retrieved by the method within the scope for the Sobel, RFI filter and Squeezenet application. This indicates the gap between simulation and measurement, and therefore the robustness of our tool. Due to hardware limitations on the Grid5000 cluster, we were unable to precisely measure the energy consumption and machine cost for this study. As a result, we could not accurately assess the discrepancy between the simulated results and the actual values, affecting the

reliability of the simulations. Our focus is primarily on comparing final latency and memory usage, chosen for their critical importance in our research, along with the availability of measurable data.

5.3.2 Tune Architecture Simulation Analysis

Figure 5.3 shows that the method found the best architecture for deploying the Sobel application in less than 15 minutes, which for manual methods can take weeks. Additionally, the results in Figure 5.3 illustrate that scope tuning narrows down the simulation to Pareto-optimal architectures for the Sobel dataflow algorithm deployment, leading to a decreased number of simulation points in Subfigure (c, d) compared to Subfigure (a, b). The method divides simulation time by a factor of 3.7, dividing the number of simulations by 3.2 while preserving 59.09% of Pareto-front-safe simulations. The resource allocation process already accelerated thanks to previous work on SimSDP now allow to focus on relevant architectures only. For this Sobel application, optimal final latency results are obtained on an architecture composed of 6 nodes, and 6 cores.

5.3.3 Network Simulation Analysis

The results in Figure 5.4 highlight the impact of network topologies on (a) a small architecture where the deployment of the Sobel application achieves the best final latency result and on (b) a bigger architecture where the deployment of the Squeezenet application achieve the best final latency result, respectively on homogeneous 6-node 6-core and 24-node 6-core architectures. Despite the higher degree of parallelism of Squeezenet compared to Sobel, enabling latency gains through deployment on a larger architecture, both applications share similarities in their regular communication patterns, making them more suitable for deployment on a torus cluster configuration. Indeed, for both applications, the torus consistently yields the lowest latency, as illustrated in the figure. However, for the Sobel application, the Fat Tree configuration offers a superior trade-off across all considered metrics. In this specific scenario, the Fat Tree configuration provides more efficient routing and a more balanced distribution of communication compared to the torus, resulting in energy savings.

5.3.4 In Vivo and Silico comparison

Table 5.1 shows that the average MSE between simulated and measured final latency speedup for the three simulated applications on the top five Pareto-optimal architectures is less than 2%. These errors are minimal, reflecting the reliability of the method. The top five Pareto-optimals revolve around a parallelism of: Sobel 36, RFI, and Squeezenet 72, respecting the limit of scope.

5.4 Conclusion

This chapter introduces a new dataflow-based co-design method for HPC systems. This method extends the multi-node simulator, SimSDP, to rapidly identify architectures optimizing final latency for a given application and evaluate crucial metrics such as final latency, energy consumption, memory usage, and associated cost. The method employs architecture moldable parameters and scope-tuning strategies to narrow down the search for the best combination of the number of nodes, core count, operating frequency, and network topology. The method is the result of collaboration between IETR for the development of simulation tools and Eviden [1] to meet the real constraints of HPC systems. The result shows that the method accelerates the search by a factor of 3.7 compared to a semi-exhaustive approach.

1. `https://eviden.com/fr-fr/`

Chapter 6

Conclusion

6.1 Summary

Resource allocation for image processing applications has become increasingly complex as target architectures and algorithms have advanced.

The contributions presented in this thesis address optimizing program execution on multiple architectural scales. These contributions have been implemented within the PREESM application development framework.

In Chapter 3, three new hardware-specific clustering methods, referred to as SCAPE, were introduced focusing on matching the granularity of an application to the parallelism of a target architecture node. The first one reduces the excessive data parallelism of the application to the parallelism of the target architecture node. The second one increases the sequential part of the application to match the parallelism of the target architecture node thanks to the software pipelining method. The last one ensures consistency over a wider spectrum of dataflow graphs. These approaches reduce resource allocation time and speed up program execution on target thanks to improved parallelism management.

In Chapter 4, a new method combining agglomerative and divisive hardware-specific clustering methods was presented as SimSDP focusing resource allocation on multi-node and multi-core heterogeneous HPC DCS architecture. The method divides the graph into balanced subgraphs based on workload estimation and simulation, each associated with an architecture node, and reuses the SCAPE method for each node to adjust granularity. Experimental results show that the method produces parallel multi-node code in a matter of minutes, outperforming the standard method of allocating resources to a single node. Additionally, the method deals with both small dataflow graphs and more complex ones.

In Chapter 5, a new multi-node, multi-core, and network topology codesign for a Pareto-optimal solution was presented to find an optimal architecture for a given application and optimize the allocation of resources on this architecture. The method is part of SimSDP and consists of exploring HPC DCS architectures, readjusting the search scope

during the simulations, based on estimates and simulations. The result shows that the method accelerates the search by a factor of 3.7 compared to a semi-exhaustive approach. Additionally, comparing simulated latency speedups with real speedups yields a MSE of less than 2%, indicating the high reliability of our simulator.

6.2 Future work

The work of this thesis opens opportunities for future research on the AAA method coverage on a wider spectrum of existing and future architectures.

6.2.1 Static Dataflow Synthesis for Heterogeneous CPU-GPU systems

As shown in the SCAPE contributions, clustering techniques based on the concept of pattern matching are efficient techniques for isolating computations and matching their behavior to the target. However, current SCAPE methods are limited to CPU-based architectures. To meet the needs of astronomers dealing with very data-intensive applications with real-time ambitions, accelerators such as GPUs should be considered. For this reason, we intend to extend SCAPE expertise to heterogeneous CPU-GPU systems. In [Mic+ed], we propose a GSLA based AAA method that addresses dataflow cluster mapping, scheduling, timing, and code generation specific to heterogeneous CPU-GPU systems. This is a first step towards rapid prototyping, but code optimization remains limited by the simplistic design of the GSLA model.

6.2.2 Hardware-Specific Polyhedral Optimizations for Dataflow Application

Polyhedral optimization techniques aim to enhance data locality and optimize cache utilization by reorganizing loop iterations, merging or distributing loops, and applying tiling techniques. By clustering loops, we can isolate them for more precise and efficient transformations, simplify analysis, facilitate parallelization, and foster the development of novel optimization strategies. These advantages make the polyhedral approach particularly promising for boosting the performance of computationally intensive applications. Previous work by Fontaine et al. [FGM18] demonstrated the benefits of integrating

polyhedral compilation with dataflow languages. Their research utilized the polyhedral optimizer tool Pluto, which takes a delineated loop and a set of reorganized loop schedules as input. In our current framework, the internal schedules of clusters are represented by nested for-loops generated by the APGAN algorithm [BML97], which could serve as automatic inputs for the Pluto optimizer. Furthermore, in [Laz20], the author proposed embedding the Automatic Speculative Polyhedral Loop Optimizer (APOLLO) tool within the dataflow design framework PREESM to provide a runtime manager equipped with a multi-versioning system. This system evaluates various optimizing transformations and selects the most suitable one based on the runtime context. Exploring these directions will not only optimize execution at runtime but also adapt dynamically to different hardware-specific constraints, thus opening new avenues for research and development in polyhedral optimization and dataflow applications.

6.2.3 Hardware/Software Co-design on Heterogeneous HPC systems

Extending our hardware/software co-design method, presented as SimSDP, initially developed for homogeneous multi-node and multi-core HPC architectures, to heterogeneous systems offers significant advantages. By exploiting the diversity of available resources, such as CPUs, GPUs, and FPGAs, this future work would enable optimal resource utilization, maximizing performance and energy efficiency. It would also offer greater flexibility and scalability to a variety of workloads. This heterogeneity increases the number of parameters defining the ideal solution architecture, making fast search algorithms challenging. One solution could be to represent all these elements as nodes of the MoA, for example extending the GSLA to the FPGA, then estimating the scalability ratio of executing a program on each of the elements.

Appendix A

French Summary

A.1 Introduction

La croissance de la puissance de calcul dans les systèmes de Calcul Haute Performance (HPC) a révolutionné divers domaines, des applications astronomiques aux modèles d'intelligence artificielle avancés comme ChatGPT. Par exemple, le Réseau d'un Kilomètre Carré (SKA), un radiotélescope exascale, ouvre de nouvelles perspectives en astronomie. Son principal composant, le pipeline du Processeur de Données Scientifiques (SDP), doit traiter des données à plusieurs téraoctets par seconde, tout en respectant des contraintes de stockage et d'énergie. Pour relever ces défis, un supercalculateur HPC dédié au SDP reste à construire.

La construction d'une architecture HPC implique l'interconnexion de serveurs de calcul pour former des grappes, facilitant le traitement parallèle et l'exécution efficace de diverses tâches. Les supercalculateurs HPC partagent des composants fondamentaux avec les ordinateurs de bureau conventionnels, mais avec une puissance et une interconnexion accrues, ce qui leur permet d'atteindre des performances de calculs remarquables.

Alors que la loi de Moore atteint ses limites, les systèmes HPC exploitent le parallélisme pour surmonter les contraintes matérielles. Cependant, cette complexité croissante nécessite des efforts pour combler les disparités entre conception et logiciel. Le modèle de flux de données offre une représentation graphique claire du parallélisme, facilitant la conception et l'implémentation efficace d'algorithmes parallèles.

Cette thèse s'intègre à la méthodologie Adéquation Algorithme-Architecture (AAA) pour optimiser l'utilisation des ressources, améliorer la productivité logicielle et favoriser la co-conception des architectures et des applications HPC (résumée dans les sections A.2 et A.3). Les contributions comprennent des méthodologies d'optimisation de l'allocation des ressources (résumées dans la section A.4), de distribution sur des architectures hétérogènes (résumée dans la section A.5) et de co-conception d'applications HPC (résumée dans la section A.6).

A.2 Exigence de Co-Conception Matériel/Logiciel pour SKA

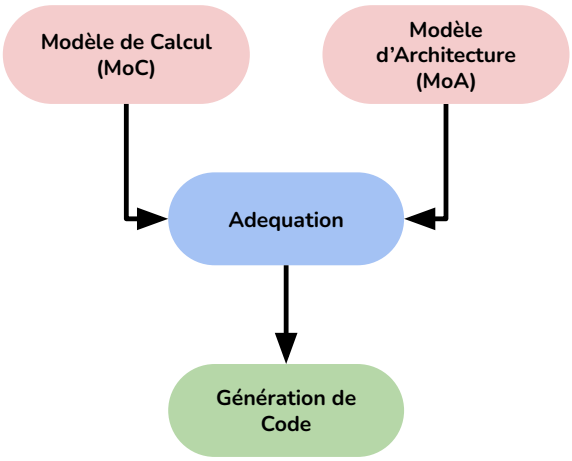

Figure A.1 – Représentation graphique en Y de la méthodologie AAA

Les progrès dans l'industrie des semi-conducteurs et les outils de compilation et de synthèse, ainsi que la recherche d'une meilleure adéquation entre algorithmes et architectures, sont essentiels pour augmenter la vitesse de calcul. Cette thèse utilise la méthodologie AAA pour aborder ce problème à l'échelle exascale du SKA. Initialement développée pour les systèmes embarqués, cette méthodologie a conduit à des outils de prototypage rapide comme Syndex, PREESM, et Spear pour automatiser l'exploration architecturale. La méthodologie AAA se base sur un modèle en Y, qui sépare la conception de l'application et de l'architecture en deux modèles indépendants. Elle utilise des modèles graphiques pour spécifier les algorithmes (Modèle de Calcul, MoC) et l'architecture matérielle (Modèle d'Architecture, MoA). La phase d'adéquation mappe l'application sur l'architecture via des transformations graphiques, optimisant ainsi l'implémentation de l'application sur l'architecture.

A.2.1 Modèle d'Architecture HPC

Cette thèse se concentre sur le Modèle d'Architecture au Niveau du Système (S-LAM) plutôt que sur les détails matériels. Le S-LAM définit le MoA $\Lambda = \langle M, L, t, p \rangle$ où M est l'ensemble des composants, L est l'ensemble des liens qui les relient, et t et p donnent respectivement un type et une propriété au composant. Il implique la distinction de

plusieurs catégories d'architecture :

— **Architecture mono-nœud et multi-nœud**: Les systèmes mono-nœuds rassemblent toutes les ressources informatiques dans une seule machine, tandis que les systèmes multi-nœuds étendent les capacités de calcul sur plusieurs machines interconnectées. Les systèmes multi-cœurs intègrent plusieurs cœurs de processeur sur une seule puce, tandis que les systèmes multiprocesseurs utilisent plusieurs processeurs distincts au sein d'une seule machine.

— **Architecture homogène et hétérogène**: Les systèmes homogènes intègrent des composants matériels identiques, tandis que les systèmes hétérogènes accueillent différents types de composants, tels que les systèmes accélérés par Unité de Traitement Graphique (GPU).

— **Architecture mémoire**: En ce qui concerne la mémoire, il existe trois types d'architectures principales : UMA, NUMA et NORMA. UMA partage la mémoire entre tous les processeurs, tandis que NUMA a une mémoire distribuée avec des accès transparents et NORMA nécessite une communication via des messages.

— **Topologies de réseau**: Les topologies de réseau dans les systèmes HPC jouent un rôle crucial dans la communication entre les nœuds. Les topologies courantes incluent la grappe avec barre transversale, le grappe avec dorsale partagée, la grappe de tores, la grappe arbre gras et la grappe libellule. Chacune a ses avantages et ses limitations en termes de bande passante, de latence et de tolérance aux pannes.

A.2.2 Modèle de Calcul Flux de donnée

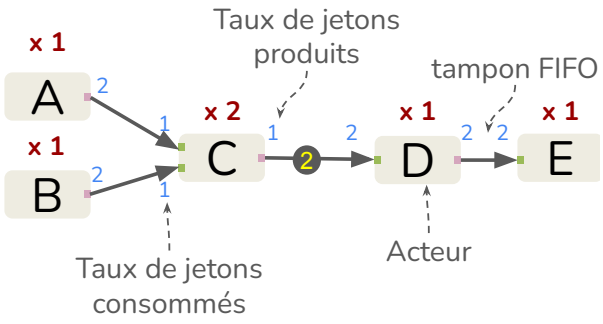

Figure A.2 – Exemple de graphe flot de donnée synchrone

La méthodologie AAA adopte une approche de flux de données qui consiste à représenter les algorithmes sous forme de graphes. Dans ces graphes, les nœuds, appelés acteurs,

représentent les calculs effectués, tandis que les arcs dirigés, désignés comme des mémoires tampons en FIFO, représentent les échanges de données entre les nœuds. Les graphes de flux de données sont particulièrement attrayants pour leur capacité à représenter les principales formes de parallélisme, telles que le parallélisme de tâches, le parallélisme de données, le parallélisme de pipeline et le parallélisme interne.

Une extension populaire de ce modèle est le modèle de flux de données synchrones (SDF), qui attribue des nombres entiers fixes aux ports d'entrée/sortie des acteurs, représentant respectivement les taux de jetons consommés et produits par un acteur lors de chacune de ses exécutions. Certains modèles de flux de données, comme le modèle Flux de Données Synchrone Basé sur l'Interface (IBSDF), définissent le comportement d'un acteur en utilisant des sous-graphes hiérarchiques, introduisant la composabilité à travers l'incorporation d'interfaces de données et de règles spécifiques à l'exécution des sous-graphes. D'autres modèles de flux de données, comme le modèle Flux de Données Paramétré et Interfacé tenant compte de l'État (SPiDF), permettent de représenter l'état interne d'un acteur à l'aide de jetons initiaux dans les FIFOs, appelés délais. Les délais sont utilisés pour créer un décalage sur la consommation de jetons par un acteur du graphe. En introduisant des jetons initiaux, il est possible de rompre les dépendances entre les acteurs connectés qui peuvent avoir une incidence sur les possibilités de programmation. Le modèle SPiDF introduit trois types de persistance de délai : Délais Locaux (LD), Délais à Persistance Locale (LPD), et Délais à Persistance Globale (GPD). Les LD persistent dans le cadre d'une itération unique de graphe, les LPD persistent dans le cadre d'une exécution du sous-graphe parent, et les GPD persistent au cours de l'ensemble des itérations de graphes. Cette propriété permet une plus grande flexibilité du modèle et une plus fiable représentation des applications.

A.3 Défis liés à l'allocation des ressources

A.3.1 Prototypage rapide

Le prototypage rapide permet aux développeurs de créer des versions initiales de ces systèmes, d'explorer des choix de conception, d'évaluer des idées et de faciliter le processus décisionnel. Il vise à réduire le temps d'exécution en permettant la division des tâches en unités plus petites exécutées simultanément, à accroître la tolérance aux pannes en expérimentant avec des mécanismes tolérants aux pannes, à exploiter explicitement le

parallélisme inhérent à certaines applications, et à faciliter l'évaluation précoce et les choix de conception corrects.

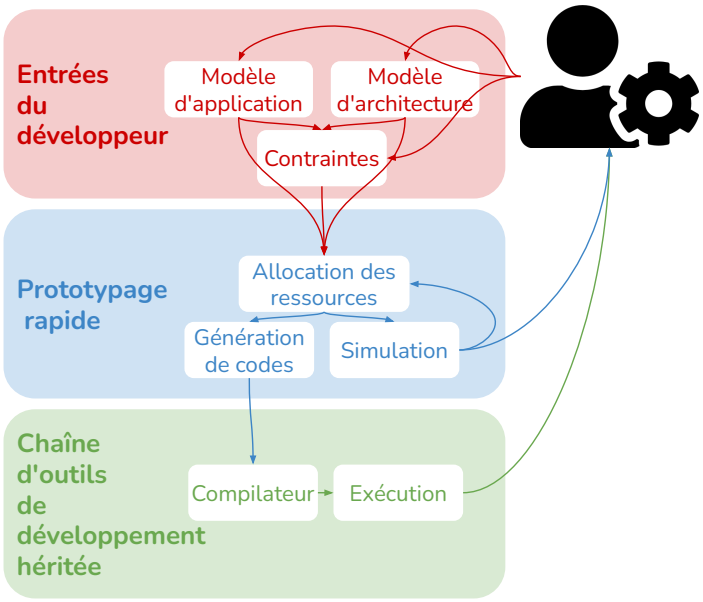

Figure A.3 – Vue d'ensemble d'un flux de conception de prototypage rapide

Le flux de conception de prototypage rapide typique comprend trois parties : les *entrées des développeurs*, le *prototypage rapide* et la *chaîne d'outils de développement héritée*. Les entrées des développeurs incluent le modèle de calcul de l'application, le modèle d'architecture ciblé et les contraintes de déploiement. Le prototypage rapide commence par la création de prototypes initiaux de l'algorithme ou du système parallèle, en mettant en œuvre rapidement différentes stratégies de parallélisation et de choix architecturaux. La chaîne d'outils de développement héritée fournit une infrastructure, des outils et des ressources pour soutenir ces efforts.

La phase du prototypage rapide comprend à son tour trois étapes : l'*allocation des ressources*, la *génération de code* et la *simulation*. Les étapes typiques de l'allocation statique des ressources incluent l'*extraction*, le *placement*, l'*ordonnancement* et la *synchronisation*. Ces étapes permettent de déterminer quelles tâches seront exécutées parallèlement ou séquentiellement, d'attribuer chaque tâche à un cœur spécifique, de planifier l'ordre d'exécution des tâches en fonction de leurs dépendances, et enfin, d'attribuer des horaires de début pour chaque tâche. La génération de code traduit les spécifications en code optimisé, avec des approches statiques ou dynamiques. La simulation relie les modèles théoriques aux phénomènes réels, en explorant le comportement du matériel et des

logiciels pour gérer des projets à grande échelle.

La phase d'extraction des méthodes classiques d'allocation des ressources pour des modèles de flux de donnée est composée des étapes suivantes :

— La **mise à plat** consiste à placer tous les acteurs d'un graphe au même niveau, ce qui signifie que tous les acteurs hiérarchiques sont remplacés par leur sous-graphe.

— La **transformation de Graphe Acyclique Dirigé à Taux Unique (SrDAG)** est utilisée pour révéler le parallélisme sur le graphe aplati. Elle met en évidence le nombre minimal d'exécutions de chaque acteur pour ramener le graphe à son état d'origine, donné par le calcul du Vecteur de Répétition RV **q**. Elle révèle également les interdépendances entre les acteurs, évitant ainsi les blocages lors de leur itération infinie.

Chaque acteur du graphe SrDAG est par la suite placé, ordonnancé et synchronisé individuellement. La complexité accrue du SrDAG augmente les possibilités de placement, ce qui permet de mieux répartir les calculs sur les différents Éléments de Traitement (PEs) et de réduire la latence de l'application sur la cible. Cependant, le placement est limité par le nombre de PEs de l'architecture. Il s'avère donc inutile et chronophage d'exposer plus de parallélisme que le nombre de PEs.

A.3.2 Optimisation l'ordonnancement des graphes flux de données

Les méthodes existantes de simplification et d'optimisation de la planification des flux de données présentent des défis importants, notamment la minimisation de la latence. Diverses approches heuristiques sont utilisées, telles que la planification par liste et la programmation linéaire en nombres entiers (ILP). Ces méthodes ont des limitations en termes de complexité du problème. Deux techniques pour simplifier et optimiser la planification des flux de données sont présentées : la mise en grappe et le recalage. La mise en grappe réduit le nombre d'acteurs à mapper tout en préservant le comportement de l'application, tandis que le recalage facilite la distribution efficace des acteurs sur les ressources disponibles. Cependant, les techniques de l'état de l'art sont limités lorsque les graphes de flux de données et le S-LAM d'entrées sont complexes. Cette thèse se concentre plus particulièrement sur l'optimisation de l'exécution d'application sur des architectures à grande échelle pour répondre aux contraintes de SKA.

A.4 Optimisation statique des applications de flux de données sur des architectures multicœurs

Traditionnellement, les méthodes de placement et d'ordonnancement de flux de données utilisent des transformations graphiques complexes, souvent inadaptées aux applications massivement parallèles en raison de leur complexité et du parallélisme dépassant les capacités des architectures cibles. Les premiers travaux de cette thèse proposent trois méthodes de mise en grappe, regroupées sous la méthode Mise à l'Échelle de Grappes d'Acteurs sur les Éléments de Traitement (SCAPE). Ces méthodes ajustent la granularité d'une application pour correspondre au parallélisme d'une architecture multi-coeur disponibles, en optimisant les opportunités de mappage et de planification. La méthode SCAPE utilise une approche de mise en grappe agglomérative, fusionnant les acteurs en fonction de leurs similitudes. Elle identifie des motifs d'acteurs, les regroupe en sous-graphes, calcule leur planification, génère le code associé, et remplace le comportement des acteurs hiérarchiques par ce code.

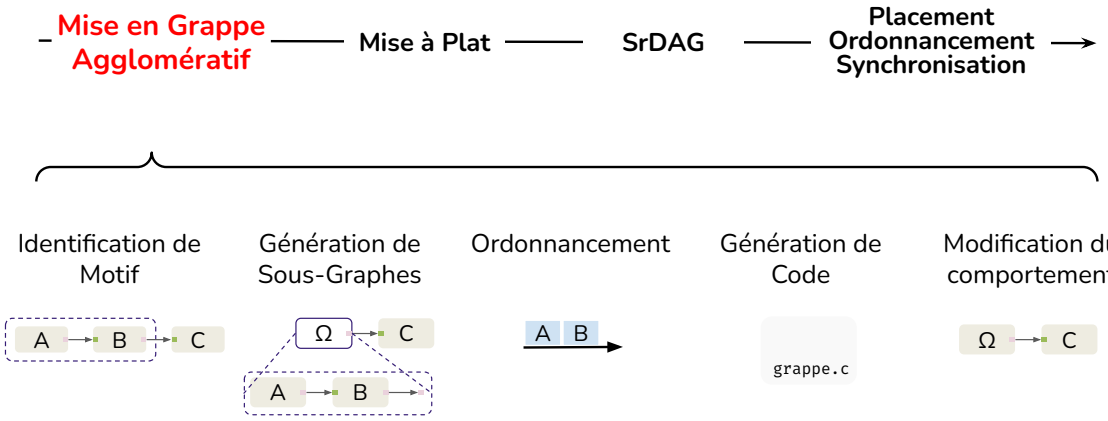

Figure A.4 – Principe de la mise en grappe agglomérative

A.4.1 Ajustement du parallélisme des données

1. **Identification de motifs particuliers**: La première version de SCAPE se concentre sur la mise à l'échelle du parallélisme de donnée sur la cible multi-coeur CPU. Pour ce faire, la méthode identifie deux motifs particuliers :
   — Nombre de Répétitions Uniques (URC) : Introduit dans [PBL95] est une séquence d'au moins deux acteurs séquentiels avec le même coefficient du RV **q** sans état

interne.

— Nombre de Répétitions Isolé (SRV) : Introduit dans [Ren+23a], est un acteur unique qui n'est pas un candidat URC, avec un RV **q** supérieur ou égal au nombre de PE de l'architecture cible.

2. **Transformation des sous-graphes**: Ces deux motifs identifiés sont traités comme deux grappes d'acteurs. Une fois que les acteurs sont isolés dans un sous-graphe, l'étape suivante, appelée mise à l'échelle, consiste à faire correspondre la répétition des grappes d'acteurs à l'architecture cible.
3. **Génération du code de la grappe d'acteur**: Le code généré par l'approche de mise à plat standard dans le cadre de prototypage rapide PREESM consiste à traduire un graphe de flux de données en code parallèle. La méthode SCAPE prend la forme de fichiers C où une fonction est définie contenant le déclenchement programmé des acteurs du sous-graphe.
4. **Transformation du graphe**: La dernière étape consiste à remplacer le comportement de l'acteur hiérarchique par le code généré au préalable. La méthode réduit la taille du graphe SDF traité par la suite par la génération de code standard.

En résumé, la version de la méthode SCAPE axée sur l'ajustement fin du parallélisme de données réduit le temps de placement et d'ordonnancement tout en préservant le parallélisme des graphes SDF. Elle consiste à réduire la taille du graphe en regroupant les acteurs reproduisant des motifs particuliers, puis à réduire les répétitions de déclenchement de ces grappes sur l'architecture cible.

A.4.2 Ajustement du parallélisme des pipelines

1. **Identification de motifs particuliers**: La deuxième version de SCAPE étend la première en considérant deux nouveaux motifs pour maximiser le parallélisme, tout en conservant les motifs précédents (URC et SRV) :

— Motif de Boucle : Introduit dans [Ren+23b] permet la création de parallélisme sur des graphes cycliques. Une partie cyclique est une séquence d'acteurs où le dernier est connecté au premier par une ou plusieurs FIFO avec un délai local (LD).

— Motif Séquentiel : Introduit dans [Ren+23b] permet la création de parallélisme sur des graphes séquentiels ou une partie avec un degré de parallélisme inférieur

au nombre de cœurs CPU. La méthode consiste à regrouper les acteurs dans un ordre topologique de telle sorte que la somme des temps d'exécution des acteurs contenus dans une grappe tend à être également répartie. Le nombre de grappes doit être égal au nombre de cœurs CPU.

2. **Transformation des Sous-graphes**: Les LD sont extraits des sous-graphes pour permettre le calcul de l'ordonnancement interne au sous-graphe. Dans le cas du motif de Boucle, la transformation consiste à dupliquer n_{loop} fois l'acteur hiérarchique, où n_{loop} est le plus grand diviseur commun des répétitions des acteurs du sous-graphe juste au-dessus du nombre de cœurs CPU.
3. **Génération du Code de la grappe** : Comme pour la version précédente de SCAPE, la troisième étape consiste à générer un code pour la grappe. La particularité du code résultant du sous-graphe contenant des acteurs en boucle réside dans la copie des sorties retardées sur les entrées retardées en utilisant la fonction *memcpy*.
4. **Transformation du Graphe** : La dernière étape consiste à remplacer le comportement de l'acteur hiérarchique par le code généré au préalable. Ensuite, la méthode intègre les pipelines en ajoutant un délai global persistant (GPD) entre les étapes de regroupement en boucle et séquentiel.

En résumé, cette version de la méthode SCAPE réduit le temps de traitement du placement et de l'ordonnancement tout en augmentant le parallélisme du graphe SDF. La méthode consiste à réduire la taille du graphe en regroupant les acteurs détectant des motifs particuliers et en augmentant le parallélisme en ajoutant des pipelines en fonction de l'architecture cible. Cependant, cette technique présente des limites lorsque le graphe d'entrée est hiérarchique et présente des cycles englobant les acteurs hiérarchiques. La méthode de nature agglomération et ascendante est aveugle du contexte hiérarchique, peut introduire un pipeline dans un cycle pouvant entrainer un désordre dans le traitement des jetons.

A.4.3 Prise en Compte de l'État du contexte Hiérarchique dans l'Ajustement de granularité

La dernière version de SCAPE étend les deux autres en tenant compte du contexte hiérarchique. La méthode évalue la possibilité de générer des pipelines non bloquante, en identifiant le niveau hiérarchique le plus élevé dans un cycle. Selon la répétition des acteurs

et le nombre d'éléments de traitement, la méthode évalue les scenarii pour optimiser les mise en grappe et éviter les pipelines dans les cycles.

1. **Identification des Motifs**: Les motifs de regroupement dépendent de l'état des niveaux hiérarchiques et de la répétition des acteurs :
   — Répétition inférieure aux éléments de traitement : Utilisation des motifs URC, SRV pour réduire le parallélisme des données et ajout d'un nouveau motif, LLI, pour réduire le parallélisme de tache avec un faible impacte sur les performances.
   — Répétition supérieure aux éléments de traitement : Regroupement grossier des niveaux inférieurs et déroulement partiel de boucle.
2. **Transformation des Sous-Graphes**: Après l'identification des motifs, la méthode génère des sous-graphes contenant les acteurs identifiés. Ces sous-graphes sont mis à l'échelle pour s'aligner sur l'architecture cible.
3. **Génération de Code de Regroupement**: La méthode génère le code pour les regroupements, spécifié en C, en encapsulant le déclenchement programmé des acteurs.
4. **Transformation du Graphe**: Le comportement des acteurs hiérarchiques est remplacé par le code généré. Selon la position du niveau hiérarchique courant par rapport au niveau hiérarchique sur lequel se positionne le cycle le plus haut dans le graphe les motifs intégrant le pipeline tels que *Loop* et *Sequentiel* sont activés.

En résumé, la méthode optimise l'utilisation des pipelines et des ressources en adaptant la mise en grappe des acteurs hiérarchiques en fonction des cycles et des niveaux hiérarchiques, garantissant ainsi une meilleure performance sur les architectures mono-noeud multicœurs homogènes.

A.5 Distribution statique des applications de flux de données sur des architectures hétérogènes multi-nœuds et multi-cœurs

Les architectures en charge de supporter les applications SKA prendront la forme de systèmes HPC hétérogènes. Les systèmes HPC standards se composent d'architecture multi-nœud et multi-cœur complexifiant la gestion des ressources. Bien que l'approche de

flux de données facilite le déploiement d'application sur des architectures mono-nœd, elle rencontre dans l'état des difficultés à l'échelle des applications exascales. Pour répondre à ces défis, notre solution proposée, le Simulateur du Processeur de Données Scientifiques (SimSDP), exploite la programmation parallèle de flux de données et propose une optimisation conjointe de l'allocation de ressources au niveau multi-nœuds et multi-cœurs, améliorant significativement le prototypage rapide en déployant rapidement et automatiquement des applications sur n'importe quelle architecture pour évaluer leur viabilité, ce qui permet de déployer des applications complexes en quelques minutes seulement. Les étapes du processus itératif sont les suivantes :

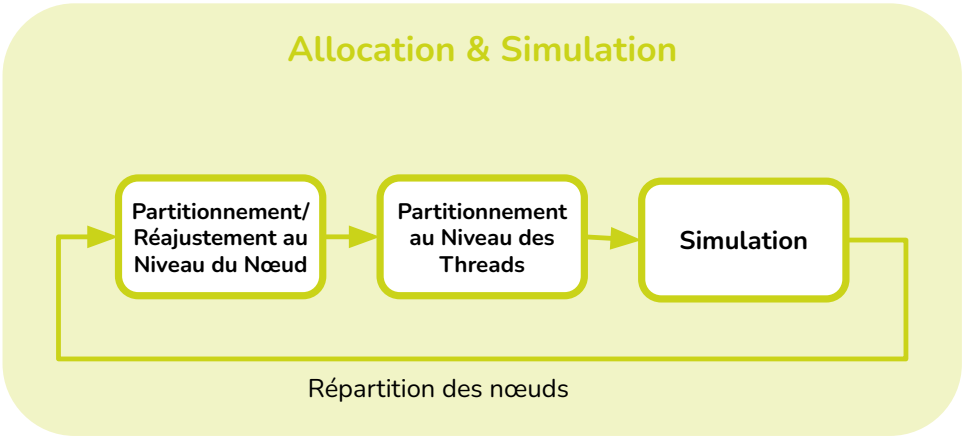

Figure A.5 – Visualisation de la procédure SimSDP

1. **Partitionnement au Niveau des Nœuds**: Cette étape génère des sous-graphes équilibrés dans le temps pour chaque nœud, en utilisant une métrique des cœurs équivalents pour tenir compte de la capacité de calcul relative de chaque nœud. Les sous-graphes sont construits séquentiellement pour chaque nœud en ordre décroissant de performance.
2. **Partitionnement au Niveau des Threads**: Cette phase ajuste la granularité de chaque sous-graphe pour correspondre à l'architecture multi-cœurs des nœuds. Elle comprend deux transformations : la transformation Euclidienne et la transformation SCAPE, pour adapter la granularité du calcul parallèle.
3. **Simulation Intra/Inter-Nœud**: Deux outils sont utilisés : PREESM pour les simulations intra-nœud et SimGrid pour les simulations inter-nœud. Ces simulations

permettent d'évaluer l'impact des communications sur l'efficacité de la distribution actuelle.

4. **Réajustement au Niveau des Nœuds**: Les résultats de la simulation précédente sont utilisés pour réévaluer le temps cumulé équivalent des sous-graphes, en tenant compte des coûts de communication et de la distribution de la charge de travail entre les nœuds.

   Génération de Code: Cette étape se concentre sur la synchronisation et le transfert des données entre les éléments du système. Les communications et synchronisations entre les nœuds sont prises en charge par la bibliothèque MPI, tandis que la synchronisation entre threads sur le même nœud est gérée par Pthreads. Les fichiers générés sont organisés en trois catégories : le fichier principal, les fichiers sous-jacents et les fichiers de cœurs. Ces fichiers contiennent les informations nécessaires pour lancer les nœuds et les threads sur chaque cœur de nœud.

En résumé, la méthode proposée vise à optimiser la répartition des charges de travail sur une architecture multi-nœuds et multi-cœurs hétérogènes, en équilibrant la charge et en minimisant les latences de communication et d'exécution. Cependant, les performances obtenues dépendent de l'architecture fournie en amont qui peut ne pas correspondre au besoin effectif de l'application.

A.6 Co-conception statique des applications flux de données tenant compte des architectures multi-nœuds, multi-cœurs et de la topologie réseau

Dans le traitement d'images astronomiques, réduire la latence finale est essentiel. Cependant, la relation entre la latence finale et le parallélisme suit la loi d'Amdahl asymptotiquement avec plusieurs minima locaux, en raison d'opportunités de placement et d'ordonnancement variable à chaque cible, complexifiant le processus d'exploration de l'espace de conception (DSE) matériel. Cette section résume une méthode qui propose un cadre d'exploration pour les systèmes HPC multi-nœuds et multi-cœurs, accélérant la recherche de co-conception matériel-logiciel. Pour encapsuler l'exploration d'une architecture adéquate adaptée à une application donnée, nous définissons :

$$\alpha(N, C, T) \implies \rho(L_{\text{final}}, M, E, \hat{C}) \tag{A.1}$$

Où α représente l'architecture paramétrée par le nombre de nœuds N, le nombre de cœurs par nœud C et la topologie du réseau T. ρ représente la fonction optimale de Pareto, évaluant la latence finale L_{final}, l'empreinte mémoire M, la consommation d'énergie E et le coût du système $\hat{C}$.

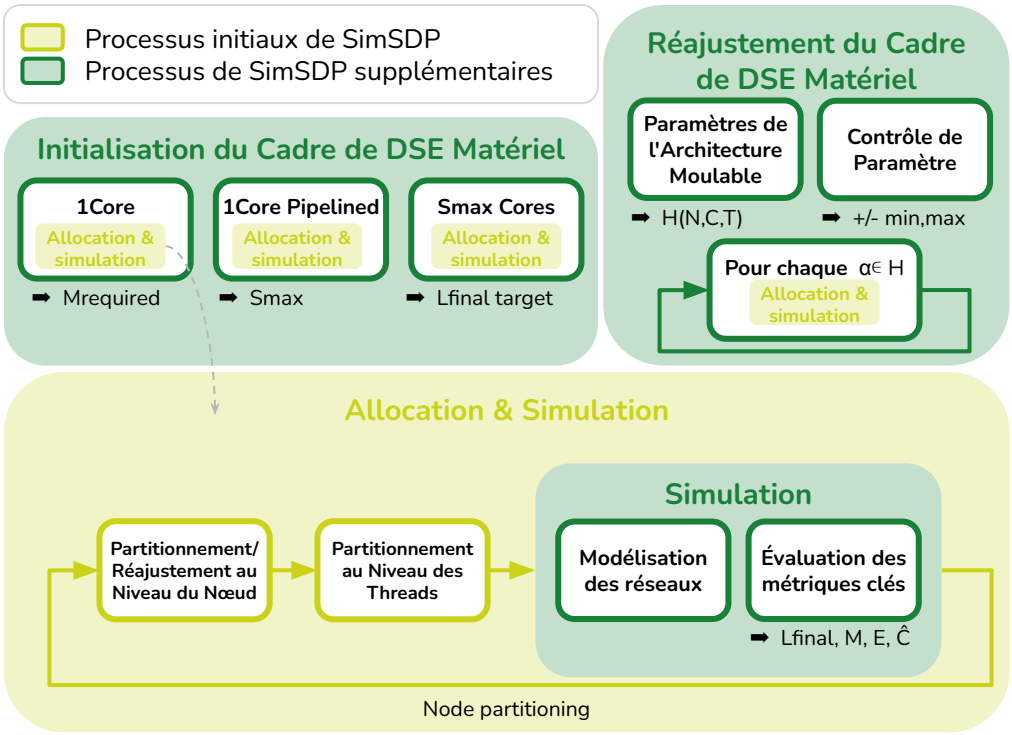

Figure A.6 – Visualisation de la procédure de Co-Conception SimSDP

1. **Initialisation du Cadre de DSE Matériel**: La phase initiale consiste à définir les limites d'exploration pour la latence finale et la mémoire. Il s'agit de trois simulations : la première alloue les ressources sur un cœur pour estimer l'empreinte mémoire minimal, la deuxième pipeline le graphe au maximum à l'aide de technique de pipeline automatique pour déterminer l'accélération maximal théorique puis la dernière alloue les ressources sur une architecture avec un parallélisme égale à accélération maximale théorique pour déterminer la potentielle latence finale optimal.
2. **Réajustement du Cadre de DSE Matériel**: La phase suivante consiste à itérer les simulations sur les architectures dont les paramètres sont réajustés pendant l'itération pour retrouver l'architecture offrant la meilleure latence. La méthode définie des paramètres d'architecture moulables, c'est-à-dire définir des plages de

valeurs pour le nombre de nœuds (N), de cœurs (C), et la topologie de réseau (T). La méthode contrôle les bornes des paramètres en réajustant les plages de N et C pour répondre aux besoins spécifiques de l'application et réduire le temps d'exploration, et identifie le niveau de parallélisme où la latence finale cesse de diminuer pendant un nombre défini d'itérations.

Simulation: Pour chaque phase, le comportement de l'application est simulé sur une architecture paramétrée. La méthode modélise les architectures avec le modèle S-LAM au niveau des nœuds ainsi qu'au niveau de la topologie réseau, alloue les ressources à l'aide de la méthode SimSDP et simule les métriques clés que sont la latence finale, les besoins mémoires, la consommation énergétique et le coût du système à l'aide des outils PREESM et SimGrid.

En résumé, la méthode proposée vise à retrouver l'architecture HPC idéale pour une application donnée. La méthode définie une architecture par un nombre de nœuds, number of cœurs par nœud et une topologie réseau et identifies celles offrant la meilleure latence for une application donnée et simule des métriques clés telles que l'énergie, la mémoire, et le coût machine. Cependant la méthode se concentre sur des architectures HPC standard, c'est à dire composé uniquement de processeurs CPU multi-cœurs.

A.7 Conclusion

L'allocation des ressources pour les applications de traitement d'images est devenue de plus en plus complexe au fur et à mesure de l'évolution des algorithmes et des architectures cibles, notamment dans le contexte du radiotélescope SKA, qui projette de traiter les données à très haut débit sous contraintes de stockage et d'énergie.

Les contributions présentées dans cette thèse portent sur l'optimisation de l'exécution des programmes sur plusieurs échelles d'architectures CPU. Basées sur des techniques de mise en grappe d'application de flux de donnée, ces méthodes améliorent à la fois le temps d'allocation des ressources et les performances sur cible, contribuant ainsi à améliorer le prototypage rapide. Ces contributions ont été implémentées dans le cadre de développement d'applications PREESM.

List of Figures

List of Tables

GLOSSARY

AAA Algorithm-Architecture Adequation. 13, 15, 18, 19, 48, 50, 51, 60, 69, 130, 132–134, 146

AAM Algorithm-Architecture Matching. 13

ALAP As Late As Possible. 101

ALU Arithmetic and Logic Unit. 19

APGAN Pairwise Grouping of Adjacent Nodes for Acyclic graph. 66, 68, 75, 77, 85, 86, 131

API Application Programming Interface. 32, 34, 35, 38–40, 48

APOLLO Automatic Speculative Polyhedral Loop Optimizer. 131

ASAP As Soon As Possible. 99, 101, 102

ASIC Application-Specific Integrated Circuit. 38

BSC Barcelona Supercomputing Center. 35

CC-NUMA Cache Coherent NUMA. 26

CISC Complex Instruction Set Computer. 25

CN Communication Node. 24

CP Critical Path. 54

CPN Critical Path Network. 104, 105, 116

CPU Central Process Unit. 24, 25, 39, 61, 67, 72, 74–78, 83–86, 88–92, 113, 116, 117, 130, 131, 138, 140, 145, 147–149, 174

CSP Central Signal Processor. 40

CUDA Compute Unified Device Architecture. 38–40

DAG Directed Acyclic Graph. 46, 47, 53, 54, 58, 61, 68

DALiuGE Data Activated 流 Graph Engine. 13

Dark Era Dataflow Algorithm aRchitecture co-design of SKA pipeline for Exascale Radio Astronomy. 10

DCS Distributed Computing System. 26, 35, 36, 129

DNN Deep Neural Network. 67

DPN Dataflow Process Network. 41, 61, 146

DPU Data Processing Unit. 38

DSE Design Space Exploration. 14, 119–121, 126, 143, 144, 174

DSL Domain-Specific Language. 40

DSP Digital Signal Processor. 61

EA Evolutionary Algorithm. 62, 67, 68

FIFO First In First Out. 41–43, 45–47, 53, 54, 59, 74, 75, 77, 78, 86–88, 92, 101, 102, 105, 109, 121, 124, 135, 139, 149

FLOPS Floating-point Operations Per Second. 9, 10

FORTRAN mathematical FORmula TRANslating system. 34

FPGA Field-Programmable Gate Array. 12, 24, 38, 39, 131

GFT Generalized Fat Trees. 29

GPD Globally Persistent Delay. 45, 46, 78, 79, 86, 135, 140

GPGPU General-Purpose Computing on Graphics Processing Unit. 38

GPU Graphics Processing Unit. 12, 24, 34, 38–40, 67, 69, 130, 131, 134

GSLA GPU-oriented System-Level Architecture Model. 24, 130, 131

HPC High-Performance Computing. 5, 7, 9–15, 19, 21–29, 31, 36, 42, 48, 49, 51, 52, 55, 56, 60–63, 69, 94, 117, 119, 128, 129, 132–134, 141, 143, 145, 149, 174

HPIR High-Performance Image Reconstruction. 39

HSDF Homogeneous Synchronous Dataflow. 46, 47

IBSDF Interface Based Synchronous Dataflow. 44, 45, 53, 135

IETR Institute of Electronics and Telecommunications of Rennes. 14

II Initiation Interval. 58

ILP Integer Linear Programming. 62, 68, 137

KPN Kahn Process Network. 41

LD Local Delay. 45, 77, 99, 135, 139, 140

LIDE DSPCAD Lightweight Dataflow Environment. 61

LLI Low latency impact. 84, 141

LP Linear Programming. 62

LPD Locally Persistent Delay. 45, 135

LSLA Linear System-Level Architecture Model. 23, 24

MAD Median Absolute Deviation. 109

MEG Memory Exclusion Graph. 59

MIMD Multiple Instruction Multiple Data. 25, 36

MISD Multiple Instruction Single Data. 25

MoA Model of Architecture. 5, 19, 21, 23–25, 27, 29, 48, 131, 133

MoC Model of Computation. 5, 19, 31–33, 35, 37, 39–49, 54, 61, 71, 72, 133, 146

MPI Message Passing Interface. 36, 37, 61, 62, 107, 108, 143, 146

MSE Mean Squared Error. 126, 128, 130

NORMA NO Remote Memory Access. 26, 27, 134

NUMA Non Uniform Memory Access. 25, 26, 134

OmpSs Open Multi-Processing Super scalar. 35

OpenACC Open Accelerators. 40

OpenCL Open Computing Language. 12, 38–40

OpenMP Open Multi-Processing. 12, 32–35, 39, 40, 146

ORCC Open RVC-CAL Compiler. 61

OS Operating System. 32, 55

PC Personnal Computer. 114

PE Processing Element. 13, 24, 42, 52–55, 62, 71, 73, 74, 76, 81, 83–85, 108, 137, 139, 146, 147

PGAN Pairwise Grouping of Adjacent Nodes. 66, 68

PGFT Parallel ports Generalized Fat Trees. 30

PiSDF Parameterized and Interfaced Synchronous Dataflow. 44, 45, 53, 72, 80, 108, 109

PREESM Parallel and Real-time Embedded Executives Scheduling Method. 14, 19, 61, 62, 75, 78, 85, 87, 92, 105, 106, 108, 109, 117, 124, 129, 131, 133, 139, 142, 145, 174

PSINS PMaC's Open Source Interconnect and Network Simulator. 61

Pthreads POSIX Threads. 32–34, 143, 146

RAM Random Access Memory. 59

RFI Radio Frequency Interference. 94, 108–110, 112–115, 117, 124, 126, 128, 147, 148

RISC Reduced Instruction Set Computer. 25

Rising STARS RISE International Network for Solutions Technologies and Applications of Real-time Systems. 10

RV Repetition Vector. 43, 54, 66, 72, 74, 75, 77, 78, 81, 85, 97–99, 103, 104, 137–139

SA-PiSDF State-Aware Parameterized and Interfaced Synchronous Dataflow. 45

SAT Satisfiability. 62

SCAPE Scaling up of Clusters of Actors on Processing Element. 71–73, 75–80, 82–84, 87–93, 101, 103–105, 114, 115, 129, 130, 138–140, 142, 147, 149

SDF Synchronous Dataflow. 41–47, 53–55, 63, 65–69, 76, 78, 79, 88, 100, 105, 121, 135, 139, 140, 146

SDP Science Data Processor. 10, 40, 132

SE Symbolic Execution. 43

SIMD Single Instruction Multiple Data. 25, 34

SimSDP Simulator of the Science Data Processor. 62, 94, 96, 107, 117, 120, 127–129, 131, 142, 145, 147, 148

SIMT Single Instruction Multiple Threads. 38

SISD Single Instruction Single Data. 25

SKA Square Kilometre Array. 5, 7, 10, 13, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 55, 59, 62, 63, 124, 132, 133, 137, 141, 145, 146

S-LAM System-Level Architecture Model. 19, 133, 137, 145

SMP Symmetric Multi-Processor. 26, 35

SPiDF State-Aware Parameterized and Interfaced DataFlow. 44, 45, 53, 86, 87, 135

SPMD Single Program, Multiple Data. 36, 146

SrDAG Single rate Directed Acyclic Graphs. 53–55, 63, 76, 78, 81, 87, 88, 90, 91, 109, 137, 146, 149

SrSDF Single rate Synchronous Dataflow. 46, 47, 54, 67, 68

SRV Single Repetition Vector. 74, 75, 77, 84, 85, 139, 141

SYCL SYstem-wide Compute Language. 38, 40

SynDEx Synchronized Distributed Executive. 18, 60

UMA Uniform Memory Access. 25, 26, 134

URC Unique Repetition Count. 66, 68, 72, 74, 75, 77, 84, 85, 88, 138, 139, 141

VAADER Video Analysis and Architecture Design for Embedded Resources. 14

VLSI Very Large Scale Integration. 67

WCRT Worst-Case Response Time. 67

XGFT Extended Generalized Fat Trees. 30

Personal Publications

[Mic+ed] Ewen Michel, Ophélie Renaud, Karol Desnos, Adam Deller, Chris Phillips, and Jean-François Nezan, « Static Dataflow Synthesis for Heterogeneous CPU-GPU systems », *in*: to be published.

[Ren+23a] Ophélie Renaud, Dylan Gageot, Karol Desnos, and Jean-François Nezan, « SCAPE: HW-Aware Clustering of Dataflow Actors for Tunable Scheduling Complexity », *in*: *Design and Architecture for Signal and Image Processing*, ed. by Miguel Chavarrías and Alfonso Rodríguez, Cham: Springer Nature Switzerland, 2023, pp. 3–14.

[Ren+23b] Ophélie Renaud, Naouel Haggui, Karol Desnos, and Jean-François Nezan, « Automated Clustering and Pipelining of Dataflow Actors for Controlled Scheduling Complexity », *in*: *2023 31st European Signal Processing Conference (EUSIPCO)*, 2023, pp. 1698–1702, DOI: 10.23919/EUSIPCO58844.2023.10290113.

[Ren+24a] Ophélie Renaud, Hugo Miomandre, Karol Desnos, and Jean-François Nezan, « Automated Level-Based Clustering of Dataflow Actors for Controlled Scheduling Complexity », *in*: 2024, p. 103217, DOI: https://doi.org/10.1016/j.sysarc.2024.103217, URL: https://www.sciencedirect.com/science/article/pii/S1383762124001541.

[Ren+24b] Ophélie Renaud, Erwan Raffin, Karol Desnos, and Jean-François Nezan, « Multicore and Network Topology Codesign for Pareto-Optimal Multinode Architecture », *in*: *2024 32nd European Signal Processing Conference (EUSIPCO)*, 2024.

[Ren+ed] Ophélie Renaud, Adrien Gougeon, Karol Desnos, Chris Phillips, John Tuthill, Martin Quinson, and Jean-François Nezan, « SimSDP: Dataflow Application Distribution on Heterogeneous Multi-Node & Multi-Core Architectures », *in*: to be published.

Bibliography

[Ace+17] F Acero, J-T Acquaviva, R Adam, N Aghanim, M Allen, M Alves, R Ammanouil, R Ansari, A Araudo, E Armengaud, et al., « French SKA White Book-The French Community towards the Square Kilometre Array », *in*: *arXiv preprint arXiv:1712.06950* (2017).

[Agu+22] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta Mancho, « OmpSs-2@Cluster: Distributed Memory Execution of Nested OpenMP-style Tasks », *in*: *Euro-Par 2022: Parallel Processing*, ed. by José Cano and Phil Trinder, Cham: Springer International Publishing, 2022, pp. 319–334, ISBN: 978-3-031-12597-3.

[AK97] Ishfaq Ahmad and Yu-Kwong Kwok, « High-performance algorithms of compile-time scheduling of parallel processors », PhD thesis, Hong Kong University of Science and Technology, 1997.

[All+95] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan, « Software pipelining », *in*: *ACM Computing Surveys (CSUR)* 27.*3* (1995), pp. 367–432.

[Arr+18] Florian Arrestier, Karol Desnos, Maxime Pelcat, Julien Heulot, Eduardo Juarez, and Daniel Menard, « Delays and states in dataflow models of computation », *in*: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, Pythagorion, Greece: Association for Computing Machinery, 2018, pp. 47–54, ISBN: 9781450364942, DOI: 10.1145/3229631.3229645, URL: https://doi.org/10.1145/3229631.3229645.

[Arr20] Florian Arrestier, « Extension and Analysis of Dataflow Models of Computation for Embedded Runtime », Theses, INSA de Rennes, Sept. 2020, URL: http://www.theses.fr/s190989.

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier, « StarPU: a unified platform for task scheduling on heterogeneous multicore architectures », *in*: *Concurrency and Computation: Practice and Experience*, Euro-Par 2009 best papers 23.*2* (2011), pp. 187–198, DOI: 10.1002/cpe.1631, URL: https://inria.hal.science/inria-00550877.

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Perez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec, « Adding Virtualization Capabilities to the Grid'5000 Testbed », *in*: *Cloud Computing and Services Science*, ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, Cham: Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04519-1.

[Baz+00] Jacek Bazewicz, Denis Trystram, Klaus Ecker, and Brigitte Plateau, *Handbook on Parallel and Distributed Processing*, 1st, Berlin, Heidelberg: Springer-Verlag, 2000, ISBN: 3540664416.

[Ber+05] Martin Bernreuther, Markus Brenk, Hans-Joachim Bungartz, Ralf-Peter Mundani, and Ioan Lucian Muntean, « Teaching High-Performance Computing on a High-Performance Cluster », *in*: *Computational Science – ICCS 2005*, ed. by Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–9, ISBN: 978-3-540-32114-9.

[Bil+96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete, « Cycle-static dataflow », *in*: *IEEE Transactions on signal processing* 44.*2* (1996), pp. 397–408.

[BJ15] Fathollah Bistouni and Mohsen Jahanshahi, « Scalable crossbar network: a non-blocking interconnection network for large-scale systems », *in*: *J. Supercomput.* 71.*2* (Feb. 2015), pp. 697–728, ISSN: 0920-8542, DOI: 10.1007/s11227-014-1319-2, URL: https://doi.org/10.1007/s11227-014-1319-2.

[BL93] S. S. Bhattacharyya and E. A. Lee, « Scheduling synchronous dataflow graphs for efficient looping », *in*: *Journal of VLSI signal processing systems for signal, image and video technology* 6 (1993), pp. 271–288.

[BML96] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee, *Software synthesis from dataflow graphs*, vol. 360, Springer Science & Business Media, 1996.

[BML97] S.S. Bhattacharyya, P. Murthy, and E. Lee, « APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations », *in*: *Design Automation for Embedded Systems* 2 (Sept. 1997).

[Cap+05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, « Grid'5000: a large scale and highly reconfigurable grid experimental testbed », *in*: *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* 2005, 8 pp.-, DOI: 10.1109/GRID.2005.1542730.

[Cas+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter, « Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms », *in*: *Journal of Parallel and Distributed Computing* 74.*10* (June 2014), pp. 2899–2917, URL: http://hal.inria.fr/hal-01017319.

[CDR98] P.-Y. Calland, A. Darte, and Y. Robert, « Circuit retiming applied to decomposed software pipelining », *in*: *IEEE Transactions on Parallel and Distributed Systems* 9.*1* (1998), pp. 24–35, DOI: 10.1109/71.655240.

[CH89] J.E. Cooling and T.S. Hughes, « The emergence of rapid prototyping as a real-time software development tool », *in*: *Second International Conference on Software Engineering for Real Time Systems, 1989.* 1989, pp. 60–64.

[Cha+21] Daniel Charlet, Karol Desnos, Mickaël Dardaillon, André Ferrari, Chiara Ferrari, Nicolas Gac, Jean Francois Nezan, François Orieux, Simon Prunet, Martin Quinson, Frédéric Suter, Cyril Tasse, and Cedric Dumez-Viou, *Dataflow Algorithm aRchitecture co-design of SKA pipeline for Exascale Radio Astronomy*, ISC High Performance 2021, Poster, June 2021, URL: https://hal.science/hal-03233205.

[Chi+19] Steven WD Chien, Stefano Markidis, Vyacheslav Olshevsky, Yaroslav Bulatov, Erwin Laure, and Jeffrey Vetter, « TensorFlow doing HPC », *in*: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 509–518.

[CJV07] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, MIT press, 2007.

[CZ12] Yuankai Chen and Hai Zhou, « Buffer minimization in pipelined SDF scheduling on multi-core platforms », *in*: *17th Asia and South Pacific Design Automation Conference*, IEEE, 2012, pp. 127–132.

[DDP18] Renaud De Landtsheer, Jean-Christophe Deprez, and Christophe Ponsard, « Optimal mapping of task-based computation models over heterogeneous hardware using placer », *in*: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '18, Copenhagen, Denmark: Association for Computing Machinery, 2018, pp. 17–21, ISBN: 9781450359658, DOI: 10.1145/3270112.3270136, URL: https://doi.org/10.1145/3270112.3270136.

[Den74] Jack B. Dennis, « First version of a data flow procedure language », *in*: *Symposium on Programming*, 1974, URL: https://api.semanticscholar.org/CorpusID:31419836.

[Der19] Hamza Deroui, « Etude et implantation d'algorithmes pour le placement et l'ordonnancement d'applications Dataflow », PhD thesis, INSA de Rennes, 2019.

[Des+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi, « Pimm: Parameterized and interfaced dataflow metamodel for mpsocs runtime reconfiguration », *in*: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, 2013, pp. 41–48.

[Des+15] Karol Desnos, M. Pelcat, J.F. Nezan, and Slaheddine Aridhi, « Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs », *in*: *Journal of Signal Processing Systems* 80 (July 2015), DOI: 10.1007/s11265-014-0952-6.

[Des14] Karol Desnos, « Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs », 2014ISAR0004, PhD thesis, 2014, URL: http://www.theses.fr/2014ISAR0004/document.

[Dev22] TensorFlow Developers, « TensorFlow », *in*: *Zenodo* (2022).

[Dew+09] Peter E. Dewdney, Peter J. Hall, Richard T. Schilizzi, and T. Joseph L. W. Lazio, « The Square Kilometre Array », *in*: *Proceedings of the IEEE* 97.*8* (2009), pp. 1482–1496, DOI: 10.1109/JPROC.2009.2021005.

[Dur+11] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas, « Ompss: a proposal for programming heterogeneous multi-core architectures », *in*: *Parallel processing letters* 21.*02* (2011), pp. 173–193.

[ECP06] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel, « Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design », *in*: *IEEE Transactions on Evolutionary Computation* 10.*3* (2006), pp. 358–374, DOI: 10.1109/TEVC.2005.860766.

[Eke+03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong, « Taming heterogeneity - the Ptolemy approach », *in*: *Proceedings of the IEEE* 91.*1* (2003), pp. 127–144, DOI: 10.1109/JPROC.2002.805829.

[EKF16] Saliya Ekanayake, Supun Kamburugamuve, and Geoffrey C. Fox, « SPIDAL Java: high performance data analytics with Java and MPI on large multicore HPC clusters », *in*: *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, Pasadena, California: Society for Computer Simulation International, 2016, ISBN: 9781510823181, DOI: 10.22360/SpringSim.2016.HPC.031, URL: https://doi.org/10.22360/SpringSim.2016.HPC.031.

[Euc00] Euclide, *Stoïkheïa*, -300.

[Faa+12] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard, « Cray cascade: a scalable HPC system based on a Dragonfly network », *in*: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, ISBN: 9781467308045.

[Fly72] Michael J. Flynn, « Some Computer Organizations and Their Effectiveness », *in*: *IEEE Transactions on Computers* C-21.*9* (1972), pp. 948–960, DOI: 10.1109/TC.1972.5009071.

[For94] Message P Forum, *MPI: A message-passing interface standard*, 1994.

[Fox+15] Geoffrey C. Fox, Judy Qiu, Supun Kamburugamuve, Shantenu Jha, and Andre Luckow, « HPC-ABDS High Performance Computing Enhanced Apache Big Data Stack », *in*: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 1057–1066, DOI: 10.1109/CCGrid.2015.122.

[Gac20] Nicolas Gac, « Adéquation algorithme architecture pour l'accélération de méthodes d'inversion de données en grande dimension », Habilitation à diriger des recherches, Université Paris Saclay, Nov. 2020, URL: https://theses.hal.science/tel-03090950.

[Gen91] J. E. Gentle, *in*: *Biometrics* 47.*2* (1991), pp. 788–788, ISSN: 0006341X, 15410420, URL: http://www.jstor.org/stable/2532178 (visited on 12/25/2023).

[Gha+06] Amir Hossein Ghamarian, Marc CW Geilen, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, and Sander Stuijk, « Liveness and boundedness of synchronous data flow graphs », *in*: *2006 Formal Methods in Computer Aided Design*, IEEE, 2006, pp. 68–75.

[Gho12] Jayshree Ghorpade, « GPGPU Processing in CUDA Architecture », *in*: *Advanced Computing: An International Journal* 3.*1* (Jan. 2012), pp. 105–120, ISSN: 2229-726X, DOI: 10.5121/acij.2012.3109, URL: http://dx.doi.org/10.5121/acij.2012.3109.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel, « Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors », *in*: *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99) (IEEE Cat. No.99TH8450)*, 1999, pp. 74–78, DOI: 10.1145/301177.301489.

[GPR18] Thierry Gautier, Christian Perez, and Jérôme Richard, « On the Impact of OpenMP Task Granularity », *in*: *Evolving OpenMP for Evolving Architectures*, ed. by Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, Cham: Springer International Publishing, 2018, pp. 205–221, ISBN: 978-3-319-98521-3.

[Gra69] R. L. Graham, « Bounds on Multiprocessing Timing Anomalies », *in*: *SIAM Journal on Applied Mathematics* 17.*2* (1969), pp. 416–429, DOI: 10.1137/0117039, eprint: https://doi.org/10.1137/0117039, URL: https://doi.org/10.1137/0117039.

[GTA06] Michael I Gordon, William Thies, and Saman Amarasinghe, « Exploiting coarse-grained task, data, and pipeline parallelism in stream programs », *in*: *ACM SIGPLAN Notices* 41.*11* (2006), pp. 151–162.

[GW99] S.A Ghozati and H.C Wasserman, « The k-ary n-cube network: modeling, topological properties and routing strategies », *in*: *Computers & Electrical Engineering* 25.*3* (1999), pp. 155–168, ISSN: 0045-7906, DOI: https://doi.org/10.1016/S0045-7906(99)00003-8, URL: https://www.sciencedirect.com/science/article/pii/S0045790699000038.

[Hag+22] Naouel Haggui, Wassim Hamidouche, Fatma Belghith, Nouri Masmoudi, and Jean-François Nezan, « OpenVVC Decoder Parameterized and Interfaced Synchronous Dataflow (PiSDF) Model: Tile Based Parallelism », *in*: *Journal of Signal Processing Systems* (2022), DOI: 10.1007/s11265-022-01819-7, URL: https://hal.science/hal-03884560.

[Has00] Wilhelm Hasselbring, « Programming languages and systems for prototyping concurrent applications », *in*: *ACM Comput. Surv.* 32.*1* (Mar. 2000), pp. 43–79, ISSN: 0360-0300, DOI: 10.1145/349194.349199, URL: https://doi.org/10.1145/349194.349199.

[Has18] Julien Hascoët, « Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications », Theses, INSA de Rennes, Dec. 2018, URL: https://theses.hal.science/tel-02132613.

[Hea95] Steve Heath, *Microprocessor architectures RISC, CISC and DSP (2nd ed.)* GBR: Butterworth-Heinemann Ltd., 1995, ISBN: 0750623039.

[Heu+12] Julien Heulot, Karol Desnos, Jean-François Nezan, Maxime Pelcat, Mickaël Raulet, Hervé Yviquel, Pierre-Laurent Lagalaye, and Jean-Christophe Le Lann, « An experimental toolchain based on high-level dataflow models of computation for heterogeneous MPSoC », *in*: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing* (2012), pp. 1–2.

[Hoe12] Bernd Hoefflinger, « ITRS: The International Technology Roadmap for Semiconductors », *in*: *Chips 2020: A Guide to the Future of Nanoelectronics*, ed. by Bernd Hoefflinger, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 161–174, ISBN: 978-3-642-23096-7, DOI: 10.1007/978-3-642-23096-7_7, URL: https://doi.org/10.1007/978-3-642-23096-7_7.

[Hon+20] A. Honorat, K. Desnos, M. Dardaillon, and J.-F. Nezan, « A Fast Heuristic to Pipeline SDF Graphs », *in*: *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Embedded Computer Systems: Architectures, Modeling, and Simulation 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings, Pythagorion, Samos Island, Greece, July 2020, pp. 139–151, DOI: 10.1007/978-3-030-60939-9_10, URL: https://hal.science/hal-02993338.

[Hon20] Alexandre Honorat, « Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints », Theses, INSA de Rennes, Nov. 2020, URL: https://theses.hal.science/tel-03337988.

[HSL10] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine, « LogGOPSim: simulating large-scale applications in the LogGOPS model », *in*: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 597–604.

[Ian+16] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, 2016, arXiv: 1602.07360 [cs.CV].

[IGL22] M. Numan Ince, Melih Günay, and Joseph William Ledet, « Lightweight distributed computing framework for orchestrating high performance computing and big data », *in*: *Turkish J. Electr. Eng. Comput. Sci.* 30 (2022), pp. 1571–1585, URL: https://api.semanticscholar.org/CorpusID:249289459.

[Jac10] Joan Jacobs, « D-Mod-K Routing Providing Non-Blocking Traffic for Shift Permutations on Real Life Fat Trees », *in*: 2010, URL: https://api.semanticscholar.org/CorpusID:1831393.

[Jeo+21] Dowhan Jeong, Jangryul Kim, Mari-Liis Oldja, and Soonhoi Ha, « Parallel Scheduling of Multiple SDF Graphs Onto Heterogeneous Processors », *in*: *IEEE Access* 9 (2021), pp. 20493–20507, DOI: 10.1109/ACCESS.2021.3054725.

[Joh97] G.W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, McGraw-Hill Visual Technology Series, McGraw-Hill, 1997, ISBN: 9780070329157, URL: https://books.google.fr/books?id=zQNlQgAACAAJ.

[KA11] Mohammad Ayoub Khan and Abdul Quaiyum Ansari, « Quadrant-based XYZ dimension order routing algorithm for 3-D Asymmetric Torus Routing Chip (ATRC) », *in*: *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, 2011, pp. 121–124, DOI: 10.1109/ETNCC.2011.5958499.

[Kah74] Gilles Kahn, « The Semantics of a Simple Language for Parallel Programming », *in*: *IFIP Congress*, 1974, URL: https://api.semanticscholar.org/CorpusID:18030506.

[KC02] J. Knowles and D. Corne, « On metrics for comparing nondominated sets », *in*: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 1, 2002, 711–716 vol.1, DOI: 10.1109/CEC.2002.1007013.

[Khr11] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.1*, ed. by Aaftab Munshi, 2011, URL: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[Kie+97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter Wolf, « An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures », *in*: Aug. 1997, pp. 338–349, ISBN: 0-8186-7959-X, DOI: 10.1109/ASAP.1997.606839.

[Kim+08] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts, « Technology-Driven, Highly-Scalable Dragonfly Topology », *in*: *2008 International Symposium on Computer Architecture*, 2008, pp. 77–88, DOI: 10.1109/ISCA.2008.19.

[KKM16] Enagnon Cedric Klikpo, Jad Khatib, and Alix Munier-Kordon, « Modeling Multi-Periodic Simulink Systems by Synchronous Dataflow Graphs », *in*: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–10.

[KLE17] Sudeep Kanur, Johan Lilius, and Johan Ersfolk, « Detecting data-parallel synchronous dataflow graphs », *in*: *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, IEEE, 2017, pp. 1–6.

[KM08] Manjunath Kudlur and Scott Mahlke, « Orchestrating the execution of stream programs on multicore platforms », *in*: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 114–124, ISBN: 9781595938602, DOI: 10.1145/1375581.1375596, URL: https://doi.org/10.1145/1375581.1375596.

[KS92] Henry Kautz and Bart Selman, « Planning as satisfiability », *in*: *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, Vienna, Austria: John Wiley & Sons, Inc., 1992, pp. 359–363, ISBN: 0471936081.

[Lam04] Monica S. Lam, « Software pipelining: an effective scheduling technique for VLIW machines », *in*: *SIGPLAN Not.* 39.*4* (Apr. 2004), pp. 244–256, ISSN: 0362-1340, DOI: 10.1145/989393.989420, URL: https://doi.org/10.1145/989393.989420.

[Laz20] Raquel Lazcano López, « Automatic Runtime Performance Optimization Methodology Applied to Dataflow-based Hyperspectral Imaging Cancer Detection Algorithms », PhD thesis, ETSIS_Telecomunicacion, 2020, DOI: 10.20868/UPM.thesis.65813.

[LB98] Bil Lewis and Daniel J Berg, *Multithreaded programming with Pthreads*, Prentice-Hall, Inc., 1998.

[Lei85] Charles E. Leiserson, « Fat-trees: Universal networks for hardware-efficient supercomputing », *in*: *IEEE Transactions on Computers* C-34.*10* (1985), pp. 892–901, DOI: 10.1109/TC.1985.6312192.

[LH93] Ben Lee and A.R. Hurson, « Issues in Dataflow Computing », *in*: ed. by Marshall C. Yovits, vol. 37, Advances in Computers, Elsevier, 1993, pp. 285–333,

DOI: https://doi.org/10.1016/S0065-2458(08)60407-6, URL: https://www.sciencedirect.com/science/article/pii/S0065245808604076.

[Li+23] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen, « Analyzing Resource Utilization in an HPC System: A Case Study of NERSC's Perlmutter », *in*: *High Performance Computing*, ed. by Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse, Cham: Springer Nature Switzerland, 2023, pp. 297–316, ISBN: 978-3-031-32041-5.

[Liv+07] Nikolaos Liveris, Chuan Lin, Jia Wang, Hai Zhou, and Prithviraj Banerjee, « Retiming for synchronous data flow graphs », *in*: *2007 Asia and South Pacific Design Automation Conference*, IEEE, 2007, pp. 480–485.

[LM69] E. L. Lawler and J. M. Moore, « A Functional Equation and its Application to Resource Allocation and Sequencing Problems », *in*: *Manage. Sci.* 16.*1* (Sept. 1969), pp. 77–84, ISSN: 0025-1909, DOI: 10.1287/mnsc.16.1.77, URL: https://doi.org/10.1287/mnsc.16.1.77.

[LM87a] E Lee and D Messerschmitt, « Pipeline interleaved programmable dsp's: Synchronous data flow programming », *in*: *IEEE Transactions on acoustics, speech, and signal processing* 35.*9* (1987), pp. 1334–1345.

[LM87b] Edward A Lee and David G Messerschmitt, « Synchronous data flow », *in*: *Proceedings of the IEEE* 75.*9* (1987), pp. 1235–1245.

[LM87c] Edward Ashford Lee and David G. Messerschmitt, « Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing », *in*: *IEEE Transactions on Computers* C-36.*1* (1987), pp. 24–35.

[LP95] Edward A Lee and Thomas M Parks, « Dataflow process networks », *in*: *Proceedings of the IEEE* 83.*5* (1995), pp. 773–801.

[LS91] Charles E Leiserson and James B Saxe, « Retiming synchronous circuitry », *in*: *Algorithmica* 6.*1* (1991), pp. 5–35.

[MC09] Guillaume Mercier and Jérôme Clet-Ortega, « Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments », *in*: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ed. by Matti Ropo, Jan Westerholm, and Jack Dongarra, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 104–115, ISBN: 978-3-642-03770-2.

[MG13] Avinash Malik and David Gregg, « Orchestrating stream graphs using model checking », *in*: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.*3* (2013), pp. 1–25.

[Mio22] Hugo Miomandre, « Approximated Computing-based Methods for Hardware Resources Reduction Targeting Heterogeneous Systems », Theses, INSA de Rennes, Dec. 2022, URL: https://theses.hal.science/tel-04496146.

[Moo98] G.E. Moore, « Cramming More Components Onto Integrated Circuits », *in*: *Proceedings of the IEEE* 86.*1* (1998), pp. 82–85, DOI: 10.1109/JPROC.1998.658762.

[MR14] Michael Masin and Tal Raviv, « Linear programming-based algorithms for the minimum makespan high multiplicity jobshop problem », *in*: *Journal of Scheduling* 17 (Aug. 2014), DOI: 10.1007/s10951-014-0376-y.

[Nez09] Jean François Nezan, « Prototypage rapide d'applications de traitement des images sur systèmes embarqués », Habilitation à diriger des recherches, Université Rennes 1, Nov. 2009, URL: https://theses.hal.science/tel-00564516.

[NL04] Stephen Neuendorffer and Edward Lee, « Hierarchical reconfiguration of dataflow models », *in*: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 2004, pp. 179–188.

[Ohr+95] S.R. Ohring, M. Ibel, S.K. Das, and M.J. Kumar, « On generalized fat trees », *in*: *Proceedings of 9th International Parallel Processing Symposium*, 1995, pp. 37–44, DOI: 10.1109/IPPS.1995.395911.

[Par07] Keshab K Parhi, *VLSI digital signal processing systems: design and implementation*, John Wiley & Sons, 2007.

[PB98] Sundeep Prakash and Rajive L Bagrodia, « MPI-SIM: using parallel simulation to evaluate MPI programs », *in*: *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*, vol. 1, IEEE, 1998, pp. 467–474.

[PBL95] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee, « A hierarchical multiprocessor scheduling system for DSP applications », *in*: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, vol. 1, 1995, 122–126 vol.1, DOI: 10.1109/ACSSC.1995.540525.

[PBR09] Jonathan Piat, Shuvra S Bhattacharyya, and Mickaël Raulet, « Interface-based hierarchy for synchronous data-flow graphs », *in*: *2009 IEEE workshop on signal processing systems*, IEEE, 2009, pp. 145–150.

[Pel+09a] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, and Jean-Francois Nezan, « Scalable compile-time scheduler for multi-core architectures », *in*: *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 1552–1555, DOI: 10.1109/DATE.2009.5090909.

[Pel+09b] Maxime Pelcat, Jean Francois Nezan, Jonathan Piat, Jerome Croizer, and Slaheddine Aridhi, « A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems », *in*: *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, 2009, 8– pages.

[Pel+13] M. Pelcat, Slaheddine Aridhi, J. Piat, and J.F. Nezan, *Physical layer multi-core prototyping*, vol. 171, Jan. 2013, pp. 1–207, DOI: 10.1007/978-1-4471-4210-2.

[Pel+14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, « Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming », *in*: *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, Sept. 2014, pp. 36–40, DOI: 10.1109/EDERC.2014.6924354.

[Pel+18] Maxime Pelcat, Alexandre Mercat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, Wassim Hamidouche, Daniel Menard, and Shuvra S Bhattacharyya, « Reproducible Evaluation of System Efficiency with a Model of Architecture: From Theory to Practice », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.*10* (Oct. 2018), pp. 2050–2063, DOI: 10.1109/TCAD.2017.2774822, URL: https://hal.science/hal-01646738.

[Pel10] Maxime Pelcat, « Prototypage Rapide et Génération de Code pour DSP Multi-Coeurs Appliqués à la Couche Physique des Stations de Base 3GPP LTE », Theses, INSA de Rennes, Sept. 2010, URL: https://theses.hal.science/tel-00578043.

[Pel17] Maxime Pelcat, « Models, Methods and Tools for Bridging the Design Productivity Gap of Embedded Signal Processing Systems », Habilitation à diriger des recherches, Université Clermont Auvergne, July 2017, URL: https://theses.hal.science/tel-01610096.

[Pet+11] Teodora Petrisor, Eric Lenormand, Remi Barrere, and Michel Barreteau, « SpearDE : Plateforme de développement de chaînes de parallélisation sur architectures distribuées hétérogènes », *in*: Nov. 2011.

[PPH21] Saman Payvar, Maxime Pelcat, and Timo D Hämäläinen, « A model of architecture for estimating GPU processing performance and power », *in*: *Design Automation for Embedded Systems* 25 (2021), pp. 43–63.

[PV97] F. Petrini and M. Vanneschi, « k-ary n-trees: high performance networks for massively parallel architectures », *in*: *Proceedings 11th International Parallel Processing Symposium*, 1997, pp. 87–93, DOI: 10.1109/IPPS.1997.580853.

[RK08] M. Radetzki and R. Salimi Khaligh, « Accuracy-Adaptive Simulation of Transaction Level Models », *in*: *2008 Design, Automation and Test in Europe*, 2008, pp. 788–791, DOI: 10.1109/DATE.2008.4484912.

[Roc15] Matthew Rocklin, « Dask: Parallel computation with blocked algorithms and task scheduling », *in*: *Proceedings of the 14th python in science conference*, 130-136, Citeseer, 2015.

[Sar87] V. Sarkar, « Partitioning and scheduling parallel programs for execution on multiprocessors », *in*: (Jan. 1987), URL: https://www.osti.gov/biblio/7043298.

[Sav97] John E. Savage, *Models of Computation: Exploring the Power of Computing*, 1st, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0201895390.

[Sha48] C. E. Shannon, « A mathematical theory of communication », *in*: *The Bell System Technical Journal* 27.*3* (1948), pp. 379–423, DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[She+11] Chung-Ching Shen, Lai-Huei Wang, Inkeun Cho, Scott Kim, Stephen Won, William Plishker, and Shuvra Bhattacharyya, « The DSPCAD Lightweight Dataflow Environment: Introduction to LIDE Version 0.1 », *in*: (Oct. 2011).

[Shp+17] Alexander Shpiner, Zachy Haramaty, Saar Eliad, Vladimir Zdornov, Barak Gafni, and Eitan Zahavi, « Dragonfly+: Low Cost Topology for Scaling Datacenters », *in*: *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, 2017, pp. 1–8, DOI: 10.1109/HiPINEB.2017.11.

[Shv11] Konstantin Shvachko, « Apache Hadoop », *in*: *The Scalability Update* 36.*3* (2011), pp. 7–13.

[Sin07] Oliver Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*, USA: Wiley-Interscience, 2007, ISBN: 0471735760.

[Sor04] Yves Sorel, « SynDEx : System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems », *in*: 2004, URL: https://api.semanticscholar.org/CorpusID:54187614.

[Spa18] Apache Spark, « Apache spark », *in*: *Retrieved January* 17.*1* (2018), p. 2018.

[Tik+09] Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely, « PSINS: An Open Source Event Tracer and Execution Simulator », *in*: *2009 DoD High Performance Computing Modernization Program Users Group Conference*, 2009, pp. 444–449, DOI: 10.1109/HPCMP-UGC.2009.73.

[Tör+06] Martin Törngren, Dan Henriksson, Ola Redell, Jad El-khoury, Daniel Simon, Zdeněk Hanzálek, and Karl-Erik Årzén, « Co-design of Control Systems and their real-time implementation - A Tool Survey », *in*: (Jan. 2006).

[Tri+13] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A Lee, « Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 12.*3* (2013), pp. 1–26.

[UGT09] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil, « Software pipelined execution of stream programs on GPUs », *in*: *2009 International Symposium on Code Generation and Optimization*, IEEE, 2009, pp. 200–209.

[Wal16] M Mitchell Waldrop, « More than moore », *in*: *Nature* 530.*7589* (2016), pp. 144–148.

[Wan+24] Sunrise Wang, Nicolas Gac, Hugo Miomandre, Jean-Francois Nezan, Karol Desnos, and Francois Orieux, « An Initial Framework for Prototyping Radio-Interferometric Imaging Pipelines », *in*: *Design and Architectures for Signal and Image Processing*, ed. by Tiago Dias and Paola Busia, Cham: Springer Nature Switzerland, 2024, pp. 56–67, ISBN: 978-3-031-62874-0.

[WHL19] Minjie Wang, Chien-chin Huang, and Jinyang Li, « Supporting Very Large Models Using Automatic Dataflow Graph Partitioning », *in*: *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, Dresden, Germany: Association for Computing Machinery, 2019, ISBN: 9781450362818, DOI: 10.1145/3302424.3303953, URL: https://doi.org/10.1145/3302424.3303953.

[Wie+12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey, « OpenACC — First Experiences with Real-World Applications », *in*: *Euro-Par 2012 Parallel Processing*, ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870, ISBN: 978-3-642-32820-6.

[Wu+17] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, Richard Dodson, Markus Dolensky, Ying Mei, and Feng Wang, *DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge*, 2017, arXiv: 1702.07617 [cs.DC].

[YGD13] Xuanxia Yao, Peng Geng, and Xiaojiang Du, « A Task Scheduling Algorithm for Multi-core Processors », *in*: *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, unknown: IEEE, 2013, pp. 259–264, DOI: 10.1109/PDCAT.2013.47.

[YH09] Hoeseok Yang and Soonhoi Ha, « Pipelined data parallel task mapping/scheduling technique for MPSoC », *in*: *2009 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2009, pp. 69–74.

[Yvi+13] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet, « Orcc: multimedia development made easy », *in*: MM '13, Barcelona, Spain: Association for Computing Machinery, 2013, pp. 863–866, ISBN: 9781450324045, DOI: 10.1145/2502081.2502231, URL: https://doi.org/10.1145/2502081.2502231.

[Zha+13] Jinglin Zhang, Jean-François Nezan, Maxime Pelcat, and J.-G. Cousin, « Real-time GPU-based local stereo matching method », *in*: *2013 Conference on Design and Architectures for Signal and Image Processing* (2013), pp. 209–214, URL: https://api.semanticscholar.org/CorpusID:9166047.

[Zho+13] Zheng Zhou, Karol Desnos, Maxime Pelcat, Jean-François Nezan, William Plishker, and Shuvra S Bhattacharyya, « Scheduling of parallelized synchronous dataflow actors », *in*: *2013 International Symposium on System on Chip (SoC)*, IEEE, 2013, pp. 1–10.

[Zhu+15] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk, « Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.*6* (2015), pp. 905–918.

[ZKK04] G. Zheng, Gunavardhan Kakulapati, and L.V. Kale, « BigSim: a parallel simulator for performance prediction of extremely large parallel machines », *in*: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 78–, DOI: 10.1109/IPDPS.2004.1303013.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Titre : Optimisation de la granularité basée sur un modèle pour les systèmes de calcul de haute performance en astronomie

Mot clés : Exploration d'Espace de Conception, Flux-de-Données, Calcul Haute Performance, CPU

Résumé : La croissance rapide de la puissance de calcul a ouvert la voie aux systèmes HPC, permettant des vitesses remarquables de traitement des données et de calculs complexes. Alors qu'un processeur de 3 GHz dans un ordinateur portable standard peut effectuer environ 3 gigaflops, les systèmes HPC peuvent atteindre des capacités de téra- ou même d'exaflop. Cette avancée a eu un impact significatif sur des domaines tels que l'astronomie et l'intelligence artificielle, mais elle présente également de nouveaux défis en matière de gestion et d'optimisation des systèmes entraînant des lacunes en termes de productivité. Ces lacunes incluent des défis liés à la conception de matériels avancés et à la création de logiciels qui les exploitent pleinement. Combler ces lacunes est crucial pour maximiser le potentiel de la technologie HPC.

Cette thèse aborde les défis des systèmes HPC en optimisant l'utilisation des ressources, en améliorant la productivité logicielle et en faisant progresser la co-conception des architectures et des applications. Elle présente des méthodes pour optimiser l'allocation des ressources dans les processeurs multi-cœurs, distribuer les ressources entre les processeurs hétérogènes et identifier les topologies optimales pour les applications HPC. Ces contributions ont été implémentées dans le logiciel PREESM.

Title: Model Based Granularity Optimization for High-Performance Computing Systems in Astronomy

Keywords: Design Space Exploration, Dataflow , High-Performance Computing, CPU

Abstract: The rapid growth in computational power has marked the emergence of HPC, enabling remarkable speeds in data processing and complex calculations. While a 3 GHz processor in a standard laptop can perform around 3 gigaflops, HPC systems can reach tera- or even exaflop capabilities. This advancement has significantly impacted fields like astronomy and artificial intelligence but also presents new challenges in system management and optimization leading to productivity gaps. These gaps include challenges in designing advanced hardware and creating software that fully utilizes it. Bridging these gaps is crucial for maximizing the potential of HPC technology.

This thesis addresses challenges in HPC systems by optimizing resource use, enhancing software productivity, and advancing the co-design of architectures and applications. It introduces methods for optimizing resource allocation in multi-core processors, distributing resources among heterogeneous processors, and identifying optimal topologies for HPC applications. These contributions have been implemented into the PREESM framework.