University of Pisa

and

Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Master Thesis

Processing-in-Memory in the Exascale Computing and Big Data era: a Structured Approach

Candidate

Francesco De Felice

Supervisor

Marco Vanneschi

Academic Year 2014/2015

To the memory of Prof. Giovanna Limongi,

hoping that, one day, I will come back to find you

in the high school of angels.

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Prof. Marco Vanneschi, whose expertise, invaluably-constructive criticism and friendly advice added considerably to my graduate experience.

A special thank goes to *all* the professors of the Master degree program in Computer Science and Networking; in particular, to Prof. Marco Danelutto for all the insightful conversations we had, and to Prof. Paolo Ferragina for his precious advices about the benchmark algorithm used in this Thesis.

Furthermore, I would like also to thank *all* my MCSN colleagues, in particular Luigi C. for his true friendship and invaluable support.

I will never find enough words to thank *all* my family, in particular my parents Sergio and Licia, whose unconditioned love and endless support gave me the necessary strength to live and study far from home these years, and my brothers Luca and Marco, for their precious mathematical hints and help for technical drawings.

Last but not least, *all* my *sincere* gratitude goes to Erika for *tirelessly* supporting me during these years in Pisa (therefore, this work is mainly dedicated to you).

Francesco De Felice

Pisa, *October 2015*

Contents

Chapter 1
Introduction

Over last decades, large-scale systems computations were characterized by a processing model referred to as *computing-centric*, for which data lives on large-scale distributed file systems and moves as needed to central computing engines across a deep storage hierarchy. Programs executions were clearly dominated by arithmetic and/or logic calculations and the final goal was to maximize the number of operations per unit of time.

The data explosion phenomenon in the *Big Data* era is leading to a shift in large-scale applications processing paradigm, which is ever more *data-centric* and for which computation should be logically moved to data and not the opposite. As a matter of fact, the time spent for moving huge amount of data within a computational system clearly dominates current large-scale applications executions and, even more, the resulting energy cost is one of the major causes of energy efficiency degradation.

The *Processing-in-Memory* (PIM) technology is an architectural model that fully embraces the *data-centric* paradigm by proposing a hardware solution that consists in placing simple or more complex processing logic close to memory. In so doing, data can be directly computed where they are stored and the latency to retrieve them is considerably reduced. PIM solutions have been studied for the first time in the 90's by multiple research groups with promising results; anyway, the widespread commercial adoption remained elusive due to the technology limitations of that time and the costs required to produce more complex chips with mixed logic [2].

Recently, the emergence of new enabling technologies, among all the *3D-stacked memories* that allow to effectively integrate processing logic with memory dies, and current computing trends requirements have motivated a revival of interest for PIM solutions.

As a matter of fact, both the *Exascale Computing* community, involved in the high-performance scientific applications context, and the Big Data one, dealing with *data analytics* applications, aim at improving performance and energy efficiency for their large-scale *data-intensive* computations by minimizing data movement. Among the other solutions, their interest for PIM is witnessed by different research works and recent scientific papers.

1.1 This work

1.1.1 Objectives

The present Thesis lays its foundation in the aforementioned context and, on the basis of the most recent results provided by the research world and reported in the supporting literature, it exploits the structured approach proposed in the main reference [1] in order to study PIM architectures from a methodological view point.

For this purpose, we lay emphasis on architectural aspects, structured parallel computations, performance and energy cost models. In so doing, we are able to perform an analytical experimentation and a performance, as well as energy, characterization of a given parallel program executed over a target PIM architecture.

Anyway, it should be remarked that many open research problems exist and the PIM technology itself is currently (October 2015) in the design phase. For this reason, different issues, such as benchmarking a wider set of applications, practical experiments/simulations and other methodological aspects, e.g. provide a formal characterization of the parallel programs run-time support over PIM architecture, are demanded to future works.

1.1.2 Methodology pursued

Starting from all the notions, concepts, hardware design choices and variants, as well as the energy/performance costs of specific hardware components retrieved from literature, we develop *Abstract Machine Models* for the PIM architectures studied. In this way, we are able to identify and capture all the relevant characteristics, features and parameters of the target physical architecture and neglect the useless ones.

From the general abstract PIM models, we then fix some parameters values provided by the literature, e.g. memory access time, memory capacity, interconnection network links bandwidth, number of PIM cores per PIM processor, etc., and we derive important architecture-dependent parameters, such as communication latencies for cache-to-cache and memory blocks transfers. These parameters will be then exploited to express *performance cost models* associated to the *structured parallel program* examples that we use as benchmarks.

Furthermore, starting from some energy-related parameters provided by the literature and associated to the PIM architecture's memory and interconnection network components, we derive novel data movement-based *energy cost models*. In this way, we are able to perform an energy characterization of a given parallel program, targeting a PIM architecture, according to the amount of energy that it consumes to transfer data during cache-to-cache cooperation and/or memory accesses (in terms of Joule per primary cache block transferred). Even more, we are able to reason about the efficiency of a certain communication pattern and/or a given parallel program mapping, and to compare different solutions from the point of view of energy.

Exploiting all the concepts, techniques and cost models detailed so far, we are then able to perform a formal and analytical analysis of parallel program examples, over PIM architectures variants, taking into account both performance and data movement energy consumption. For this purpose, we study different parallelization strategies for the well-known *Count-Min Sketch* algorithm, widely exploited in *real-time analytics* applications. Its characteristics, mainly in relation to the highly irregular data access pattern, seem to make it a suitable benchmark for comparing its parallel version variants, and relative program mappings, over PIM architectures.

Finally, we conclude with a parametric study in order to provide a quantitative yet general idea of the PIM architecture potential when executing a structured parallel program, and, hopefully, a starting point for further research.

1.2 Thesis organization

The Thesis is subdivided into seven chapters; the first one is the introduction we are carrying out. The other chapters and related topics are the following:

- Chapter 2 reviews the supporting literature, highlighting the motivations, the current computing trends, the enabling technologies, the practical applications, the existing research works and proposals concerning Processing-in-Memory.

- Chapter 3 deals with Abstract Machine Models for PIM architectures, identifying which aspects and parameters are relevant and should be taken into account in our modelling and what has to be abstracted. Moreover, a brief reminder to structured parallel computation concepts and performance cost models, in order to provide a background to a not expert reader, is detailed as well.

- Chapter 4 illustrates specific PIM architectures examples which are parametrized versions of the previous more general abstract models. In so doing, it identifies and lays emphasis on the possible interactions among system components of interest, e.g. cache-to-cache communications and memory accesses. For each one of them, the relative communication latency is computed on the basis of the methodology presented in [1] about pipelined inter-unit communications.

- Chapter 5 deals with data movement energy consumption in a PIM architecture. In this section, structured parallel computing theory is widely exploited in order to simplify the task of deriving novel data movement-based *energy cost models* for structured parallel programs targeting PIM architectures. Performance cost models from [1], in relationship to the presented parallel paradigms and collective communication forms, are reported as well.

- Chapter 6 carries out a comparative study of PIM architectures in relationship to parallel program variants, mappings, communication patterns and so on. After a brief introduction of the Count-Min sketch algorithm, a formal analysis of its parallel version variants, each one exploiting a different parallel pattern, is performed by taking into account all the above listed alternatives. The chapter ends with a concluding parametric study that provides a quantitative idea about PIM architecture potential.

- Chapter 7 sums up all the salient features of this Thesis and it also shows possible future works and open research problems.

Chapter 2
Processing-In-Memory (PIM): Overview

In this chapter the motivations, the current computational trends, the research work proposals and the practical applications of Processing-In-Memory (PIM) are properly detailed. In so doing, the following sections are not a comprehensive review of the supporting literature but, instead, key points will be emphasized in order to provide a general overview of PIM and a clear idea about the needs of PIM and its implications in current parallel applications and architectures.

In the next chapters, more specific and technical details that derive from literature will be detailed and properly referenced in the methodological treatment of PIM architectures.

2.1 Introduction: motivations for Processing-In-Memory

Over last decades, the traditional processing model adopted in large-scale systems was the *computing-centric* one. The data explosion phenomenon and current trends toward the *Big Data* era involving almost all the large-scale applications of practical interest, including the scientific and engineering computing ones [15], are leading to a shift in the computing paradigm which is ever more *data-centric*. This transition is due to the evolving nature of large-scale processing, which is no longer dominated by computational aspects like arithmetic/logic operations but, instead, by the movement of large volumes of data across memory hierarchies and interconnection networks.

The cost of moving data has been recognized as one of the major causes of performance degradations and energy consumption in current applications workloads, which are almost all *data-intensive* [28]; they typically operate on massive amounts of diverse data (structured, as the ones stored in relational databases, unstructured, such as text files, or semi-structured, for example using the XML or JSON formats) characterized by high inherent parallelism, with, often, irregular access patterns, limited locality and an intensive use of I/O and/or Memory operations [10].

The data-centric model is based on the notion of moving computation to data and not the opposite; generally speaking, the basic concept is to keep data in different memory hierarchy levels with processing engines surrounding them and operating on them locally.

In so doing, data movements are minimized or avoided.

The transition of architectures and programs towards an execution model that minimizes data movement has been recognized as a crucial step for the evolution of computational systems and the overcoming of current computational challenges [26]. In this context, the Processing-in-Memory technology lays its foundations as an architectural model which seeks to minimize data movements by computing them at the most appropriate location of the memory hierarchy (main memory or persistent storage). As detailed in the following sections, the PIM concept is not new; it was a rich area of research in the '90s with different architecture prototypes proposed, with promising results, but they never became the mainstream technology. However, current computing trends, as detailed so far, as well as the emergence of new enabling technologies (e.g. 3D-stacked memories) give enough motivations to believe that PIM will be real this time.

The limitations of current computing technologies enforce this idea; as it happened with the end of *Dennard* scaling, in which performance of a CPU with single core could no longer be improved by simply increasing the transistor's switching frequency, also the multi/many-core paradigm, in the *Post-Dennard* scaling, is running out of stream due to the *dark silicon* phenomenon: only a small fraction of transistors is allowed to remain active due to the limited power budget and the inability of decreasing power consumption while shrinking transistors size (i.e. increasing transistors density and, therefore, the number of on-chip cores).

In order to extend the effect of the *Moore's law* in the next years, new architectures freed from the “pure” *von Neumann* model are needed and PIM solutions seem to be promising candidates [2].

2.2 PIM in the Exascale computing and Big Data era

"Twins separated at birth": with this euphemism Daniel Reed, famous computational scientist, defined the two main communities in large-scale computing, the *data analytics* and the high-performance computing ones [44], by outlining how they are inherently tied, although they have diverged, and are now going to reunite.

The first one focuses on big data, machine learning and data-driven computing; nowadays they define the so called *fourth-paradigm* in scientific discovery, which continuously seeks to extract information buried in massive heterogeneous datasets in order to derive knowledge.

The second community is involved in the scientific and engineering computing areas, such as: biology and biomedicine, nuclear and high-energy physics, health science, chemistry, fluid dynamics and others [16]. For this half of the large-scale computing world, the *Exascale* (10^{18} operations per second with a power budget of about 20 MW) is the next step in the path of performance improvement that has continued for more than 50 years and that, for the first time, considers power management as a first-class design challenge.

As detailed in [15], both communities have successfully built their scalable infrastructures by relying on x86 hardware and a rich suite of (mostly) open source software tools. However, the two ecosystems (hardware, software and algorithms used) sharply differ in their targets and technical approaches. On one side, we have HPC clusters based on high-performance x86 processors, augmented with accelerators in the form of coprocessors (e.g. GPUs), high-speed low latency interconnects (e.g. Infiniband), Storage Area Network (SAN), as a global persistent data storage, and low-latency SSD disks on each node for local data storage. This hardware suite is mainly optimized for performance. Atop the cluster's hardware, Linux OS provide system services, augmented with parallel file systems (e.g. Lustre) and batch schedulers for parallel job management (e.g. SLURM). Applications are typically developed in C/C++ with the support of external libraries and tools such as MPI and OpenMP, to express inter-node and intra-node parallelism respectively, CUDA or OpenCL for coprocessor exploitation.

On the other side, a rich ecosystem has emerged for data analytics too. Data analytics clusters are typically based on commodity Ethernet networks and a large amount of local storage, with horizontal scaling (or scale-out), energy efficiency, fast I/O, and capacity as primary optimization criteria. Atop this hardware infrastructure, Apache Hadoop system implements the *MapReduce* model for data analytics, relying on a distributed file system (HDFS), for managing large number of files distributed across cluster's local storage, and HBase, an open source implementation of Google's BigTable key-value store providing tables that can serve as the input and output for Hadoop jobs. Atop Hadoop system, tools (such as Pig) provide a high level model for writing the *map* and the *reduce* phases of the MapReduce model. Together with streaming data (e.g. Storm) and graph (Giraph) processing, the Hadoop suite is mainly designed for big data analysis. Moreover, data analytics applications, differently from the HPC ones, often rely on high-level programming languages such as Java or, as in the last in-memory analytics frameworks (e.g. Spark), Scala.

Apart from the specific programming models and tools, that perhaps are the biggest point of divergence, as the scientific discovery and innovation increasingly depends on high performance computing and data analytics, the potential interoperability and scaling convergence of these two ecosystems is crucial for the future evolution of large-scale systems [15]. Consolidating HPC and Big Data workloads onto the same infrastructure by exploring *dual-use* technologies would result in cost savings, although many technical challenges exist; among all, minimizing cost associated to data movement, in terms of performance improvements and energy savings, is a crucial aspect. As said previously, the Big Data revolution involved also the scientific and engineering computing applications, by making them ever more data-intensive. At the Exascale projections, the energy cost associated to data movement will exceed the cost of floating-point operations, and memory interfaces and interconnection network links are the main contributors [32]. New “data-movement aware” architectures, as well as “data-location aware” algorithms, are therefore needed, and are studied, in order to reduce these costs.

The processing in memory solution is one of them and it has been widely accepted as a promising one by both the Big Data and the HPC/Exascale communities. As a matter of fact, large number of research works and proposals prove what stated; as an example, [15, 16, 25, 26, 31] are related to the Exascale computing and [2, 5, 11, 15, 24, 29] to Big Data.

2.3 From the first PIM generation to the second one (or Near Data Processing)

2.3.1 First PIM generation

In the '90s, multiple groups of researchers studied for the first time Processing-in-memory solutions. Different examples of PIM architecture prototypes were developed, such as: DIVA [13], IRAM [52], EXECUBE [51], FlexRAM [8] and others. Most of them were characterized by multiple PIM chips with tight integration of DRAM memory arrays with simple or more complex processing logic; such PIM chips were then connected to a traditional *host* processor. A run-time system was then responsible for spawning and off-loading tasks so that the computation was performed tightly close to where data reside.

As an example, the solution adopted by DIVA (Data-Intensive Architecture) is shown in the following figure, with PIM chips that are physically grouped as conventional memory chips, packed in a discrete PIM DIMM (Dual In-Line Memory Module) module.

A mechanism based on *parcels* exchange, very similar to the concept of active or intelligent messages [30], was then responsible of coordinating computation in memory. Each parcel incorporates the data memory address and the encoded operation/computation that have to be applied to them.

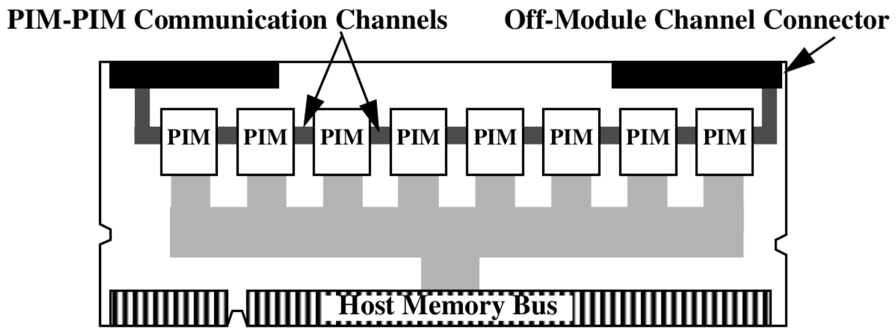

Figure 2.1: PIM DIMM module organization (*taken from* [13]).

All the PIM solutions studied in the past demonstrated performance gain and energy efficiency in many applications. As an example, DIVA was designed in order to efficiently execute irregular applications, such as sparse-matrix and pointer-based ones, obtaining a good speed-up, lower memory access latency and increased parallelism [13].

The VIRAM processor [14], designed in the scope of the previously mentioned IRAM (Intelligent RAM) project, combines vector processing logic with DRAM arrays by allowing to effectively speed-up applications characterized by fine-grained data parallelism.

Anyway, although PIM research yielded a number of promising results, widespread commercial adoption was not pursued. The economies of building PIM chips with costlier and/or suboptimal DRAM integration were not attractive to industry and, therefore, Moore's law allowed to prefer the “pure” von Neumann machines with economic affordability.

Nevertheless, in last few years a resurgence of interest in Processing-in-Memory, by both research communities and industry, has been renewed [8]. Different forms of PIM designs are now being proposed and can be differentiated according to the type and complexity of the in-memory operations supported and the processing logic used, e.g. fixed-function logic units, general-purpose energy-efficient cores, GPU cores, FPGAs, etc. As widely detailed, the emergence of new enabling technologies, among all the 3D-stacked memories that allow to effectively package processing logic with memory dies, as well as the limitations of current commodity systems, used to accommodate modern Big Data or any other Scale-Out workload in general [12], motivated the PIM revival. Therefore, research in the area of PIM can be categorized into two eras or generations from the implementation view point. As a matter of fact, the current/second PIM generation is also referred to as *Near Data Processing* (NDP) or *Near Data Computing* (NDC) in many scientific papers; this is due to the fact that now, at the physical/hardware level, the majority of the solutions propose to implement logic units *near* the main memory dies by leveraging the 3D-stacked memory organization.

A general overview of 3D-stacked memories, together with other aspects related to the second PIM generation, will be exemplified in the following sections.

2.3.2 3D-stacked memory: the enabling technology for second PIM generation

A 3D-stacked memory is an high-bandwidth, low-latency, limited-capacity and energy-efficient main memory technology. Its general organization consists of a multiple number of DRAM memory layers stacked on top of a single logic layer, which are all contained within the same chip package.

The logic-layer contains an internal network that interconnects the interfaces required to communicate with external devices and the memory interfaces/controllers. Other memory support logic, e.g. ECC (Error Correction Code) logic, is placed on the base logic die as well. Stacked memory layers are connected to the logic layer through the TSVs (Through Silicon Vias) connections. TSVs are basically high-speed vertical buses that directly connect memory banks/modules, at a certain memory stack layer, with the logic layer. Moreover, a 3D-stacked memory can be logically split into different vertical memory *slices*, with read/write memory accesses directed to each one of them handled by different memory controllers. This implies that memory slices are independent one another and parallelism can be exploited in accessing the single 3D-stacked memory. Latency and energy benefits arise from the shorter, on-chip, vertical traversed distances, and the reduced capacitance, compared to off-chip links with longer wires typical of 2D organizations [6].

3D-stacked memories are therefore useful, and have been designed, for HPC systems and high-end servers that accommodate workloads which typically exhibit little or no locality with consequent high memory bandwidth demands [7]. A representative figure of a 3D-stacked memory organization is shown in the following, where the group of vertical memory modules coloured in green represents a memory slice, MCs are Memory Controllers, I/Os are the interfaces to the external system environment and the network that interconnects them is the internal logic-layer network.

As of this writing, commercial products implementing the 3D-stacked memory model are emerging and are getting ready for the market launch, such as the Hybrid Memory Cube (HMC) by Micron, the High Bandwidth Memory (HBM) by Hynix/AMD and Wide I/O by Samsung.

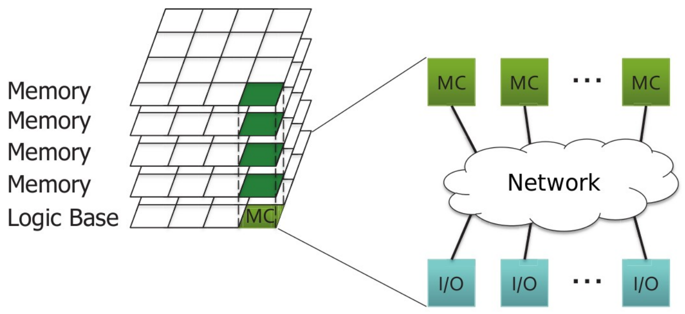

Figure 2.2: 3D-stacked memory organization (*taken from* [20]).

Wide I/O has been designed to provide energy-efficient high-bandwidth memory to mobile SoCs (System on Chips), while the HBM has been explicitly designed for graphics; as a matter of fact, both AMD and Nvidia are planning to adopt it for next generation GPUs, e.g. Nvidia *Pascal* and AMD *Radeon R9 Fury X* [45].

Finally the HMC have been designed for high-end servers, Exascale systems and many-core architectures. A notable example of a commercial chip product integrating 3D-stacked memories, based on Micron's HMCs, is the new *Xeon-Phi Knigths Landing* by Intel (2016).

Without loss of generality and according to most of the current research works on PIM, we will mainly refer to HMC as 3D-stacked memory model for its design targets, characteristics and facilities. For this reason, a brief but more detailed description of an HMC and its peculiarities will be given in the following.

Micron's Hybrid Memory Cube (HMC)

The Micron's Hybrid Memory Cube (HMC) is a 3D-stacked memory technology which, according to the last specifications released [42], consists of: 8 DRAM stacked memory dies/layers + 1 logic-layer, 32 memory slices (called *vaults* in HMC terminology) and 32 memory controllers (or *vault controllers*) one for each memory slice, up to 4 interfaces attached to external full-duplex short-range high-speed links, called SerDes (Serializer/Deserializer) links, up to 8 GB memory capacity (256 MB for each vault), maximum offered bandwidth of 320 GB/s (10 GB/s for each vault) and a crossbar logic-layer internal network. Early results showed a reduced power, reduced footprint and an increased 15x speed-up compared to traditional DDR3 DIMM technology [43].

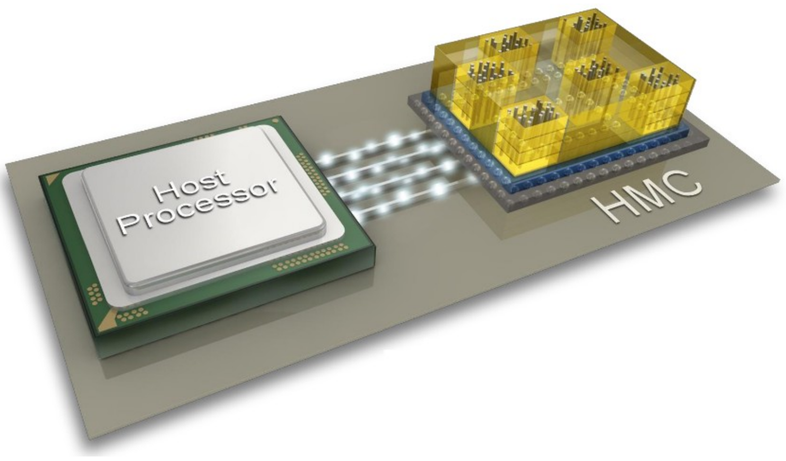

Figure 2.3: An example of a HMC directly connected to a host processor (*taken from* [43]).

Apart from numbers, one of the most important peculiarity of the HMC is the *chaining* one: in addition to the direct connection with a *host* processor, as shown in the figure above, HMCs can be connected (or *chained*) one another with point-to-point links and, in so doing, they realize a *limited-degree* interconnection network [1_Sec.18], where switching units are represented by HMC interfaces. In other words, each HMC interface unit in the logic-layer has switching capabilities for *pass-through* messages (i.e. directed to another HMC) and, moreover, routing capabilities for incoming packets that have to be directed to the correct internal vault. The chaining facility allow to realize network of 3D-stacked memories, or *memory networks* [19], which could be exploited in single- or multi-processor architectures. Different topologies can be defined, such as rings, 2D-meshes, daisy-chains, etc. An example is shown in the figure below.

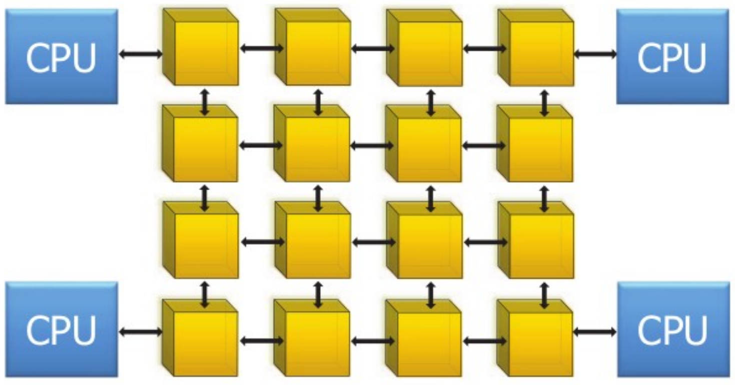

Figure 2.4: A multi-processor system with a 2D-mesh memory network (*taken from* [20]).

Another important and relevant characteristic of a HMC is its flexibility in customizing logic-layer components. Different examples, both from the research and the commercial world, exist; for instance authors in [50] propose a logic-layer network variant, called *Mesh-of-Trees,* characterized by logarithmic latency but a more scalable solution respect to the crossbar one. The above mentioned *Xeon Phi Knigths Landing* chip integrates 3D-stacked memories, called MCDRAM (multi-channel DRAM), which are basically HMCs with customized external interfaces.

Last but not least, the possibility of adding new components, by exploiting the available *rooms* in the logic layer [27], is probably the most relevant HMC aspect from the point of view of this work because it de-facto enables processing-in-memory opportunities. In this way, the *intra-stack* bandwidth can actually be exploited by PIM processors, thus increasing benefits already provided by high-bandwidth 3D memories. Therefore, as confirmed by [5, 8, 11, 16, 18] and other PIM research works, it seems that HMC are the best target for future and feasible processing-in-memory solutions.

2.3.3 Software model and software/hardware interface

According to the current research works, the software model of PIM architectures is still unclear and should be deeper researched. Obviously, the software/hardware interface will depend on the complexity of the operations intended to be performed in memory and the kind of processing logic used to support them, e.g. energy-efficient general-purpose cores [5, 11, 24, 29], GPU cores [6], programmable or fixed-functions logic units [34], etc.

An interesting PIM taxonomy is provided in [3], where logic in memory is categorized according to the operation supported, the visibility to the software level and the programming abstractions adopted. According to this taxonomy, PIM solutions can be divided into three main classes: *non-compute* PIM, *fixed-function* PIM and *programmable* PIM. The first class is related to PIM solutions that are fully transparent to the software level, for example many of the primitive processing-in-memory functionalities already provided by the HMC logic-layer, such as memory control or error detection, correction and repair, fall into this class.

The second class consists of a fixed set of software-visible operations that could be performed in memory; some potential examples include: data reductions, atomics, memory layout transformations (e.g. matrix transpose, convolutions, etc.), fixed-width vector operations, etc. These operations could be potentially exploited by program using abstractions like assembler intrinsics or library calls.

The third class consists of fully-programmable logic in memory solutions providing expressiveness and flexibility along with a series of associated overheads, in terms of energy, area and complexity, that derive from the integration of fully capable PIM processors in memory.

The programming model can potentially resemble existing multi-core architectures models, including standard threading and tasking models exposed via PIM-augmented APIs or pragmas typical of OpenMP. Furthermore, standards for heterogeneous computing, such as OpenCL, may be enhanced in order to provide task- and data-parallel programming models targeting PIM architectures [4].

Other higher-level abstractions based on C++ template formalism have been proposed in [9]; while in [27], the adoption of PGAS (Partitioned Global Address Space) programming languages providing explicit formalisms that are able to enforce the concept of data-locality are recommended.

Anyway, as said at the beginning of this section, these PIM related aspects are still an open research problem and deserve a deeper investigation in the future.

2.3.4 PIM prototypes, applications and results

As previously anticipated, PIM solutions have been experimented, through simulations, in both Exascale and Big Data domains with performance and energy improvements. As an example, different research works, such as [11, 24], concentrate on simulations performed over in-memory[1] MapReduce workloads executed on a single cluster node which consists of a host processor and multiple HMCs, each one enriched with a few number of general-purpose, ARM-like and only L1 cache (Data+Instructions) PIM cores. Two architectures of interest have been compared in [11]: the first one, let us call it the *baseline* architecture, in which only the host cores have been used for the whole executions of MapReduce applications, and the second one, let us call it the PIM architecture, in which PIM cores (of the same number of the host ones) have been used for executing Map phases, that are often memory bandwidth constrained, and local reductions only; the final centralized reduction has been executed by the host. The obtained results show that the more memory-intensive is the MapReduce workload the greater is the achieved speed-up (at most 15x). Moreover, a 18x reduction in system energy consumption has been achieved as well thanks to the data movement reduction typical of PIM architectures.

In the area of *in-memory computing*, PIM has been examined also in [34] where authors focused on the *Join* database operation execution, which is time and power consuming due to the high number of memory accesses. In fact, differently from the previous case, the Join workload is characterized by *irregular* memory access patterns and, for large datasets, it requires cross HMCs communications. By using a programmable logic unit (e.g. a micro-controller) within each HMC vault, results obtained from the in-memory Join execution showed a performance improvement of at most 5.6x, and energy reduction of at most 14.9x, with respect to the execution over a traditional CPU processor.

Better results have been achieved by authors of [21] that experimented PIM in the area of *real-time* analytics by performing in-memory *bitcount* operations over large bitmaps, in the range of 8 KB - 4 MB, achieving about 100x speed-up.

An interesting Exascale workload characterization, together with PIM architecture solution support, has been proposed in [31], where authors highlight how some scientific applications are not memory-limited/bounded at small scale but they become memory-limited at Exascale because of data-size increase (again the Big Data phenomenon comes back). Results show that PIM solutions are able to speed-up examined Exascale workloads of at most 4.2x and convert them to compute/CPU-limited.

1 The *in-memory computing* term is used to indicate current computing trends in data analytics frameworks, such as Spark or Redis, or contemporary databases, such as Oracle Database In-Memory, in which input datasets are pinned in memory before being computed/queried. In this way the I/O performance bottleneck is substantially reduced but, anyway, the cost of moving data from main memory into CPU caches remains [33].

In [22], the executions of Fast Fourier Transform (FFT) and matrix transposition algorithms, recurring in scientific computing applications, have been off-loaded to a PIM configurable accelerator layers stacked on-top of given 3D-stacked memories. Experimental results show an energy reduction of 179x for the FFT and 96x for the matrix transposition algorithm executions.

Finally, in the Exascale computing domain, the IBM Research's Active Memory Cube (AMC) will be briefly described in the following sub-section as a promising and current PIM architecture prototype example.

IBM Research's Active Memory Cube (AMC)

The Active Memory Cube (AMC), proposed in [16] by IBM Research, is a PIM architecture prototype that leverages Micron's HMCs by integrating in the logic-layer 32 energy-efficient processing elements, called *processing lanes*, in order to speed-up Exascale applications and, at the same time, lowering power consumption.

An AMC processing lane has been designed with characteristics of a general-purpose core but tuned to be area- and power-efficient in order to meet Exascale requirements. As an example, the absence of caches and the virtual address space shared with the host processor, commonly indicated as Unified Memory View (UMV) [5], are evidences of this trade-off.

The computation part of each processing lane in the AMC is composed of four *computation slices*. Each slice includes a memory-access pipeline (also called the load/store pipeline) and an arithmetic pipeline.

The instruction set architecture (ISA) includes vector instructions; an instruction in the ISA specifies all the operations that must be executed simultaneously in all the lane's pipelines. In this way parallelism can be exploited at different levels such that AMC enabled nodes are comparable to GPU-accelerated architectures [46].

The AMC software model consists of classical C/C++ or FORTRAN programs annotated with AMC-enhanced OpenMP 4.0 directives. These directives allow the compiler to identify code sections that should be executed by AMC lanes and data regions accessed during the execution on an AMC lane.

Experiments conducted via simulations demonstrated benefits mainly in terms of energy efficiency, which is comparable to, or better than, the one achieved with notable Nvidia's GPUs [46].

2.4 Final considerations

In this chapter a limited review of the supporting literature related to PIM subjects has been presented, by highlighting how this promising architectural solution is involved in current computing contexts, trends, goals and requirements.

Many open research problems remain in this field, such as the one related to PIM programming model and abstractions, the complexity of PIM logic units and how they should be exploited (e.g. acceleration of parts of a sequential application vs execution of a real parallel application), Operating System related aspects (e.g. memory management), cache coherence, and others. Moreover, the role of the host processor has still to be well understood since, till now, it has been intended (or proposed) to be used to: control PIM core executions [5], accommodate OS services to manage system resources [25], allow to restart the computation of a PIM core if some problem arises (i.e. enact *fault-tolerance*) [11], execute CPU-intensive code or, more in general, code that benefits from multi-level cache hierarchies of host cores [6].

Actually, this last point deserves an important reasoning in that the given architecture (host processor and HMCs with PIM cores) can be exploited in the most flexible way according to the characteristics of the application that has to be executed; if compute-intensive and/or cache friendly then the host can actually be exploited, with PIM cores switched-off for energy efficiency reasons [29]. Conversely, if it is memory-intensive, then PIM cores are the best candidates to execute it.

In conclusion, it is important to stress the fact that PIM solutions are good candidates to reduce energy and increase performance of (Exascale and/or Big Data) applications with the following characteristics: little or no locality [18], short data tenancy in the processor [16], irregular or fine-grained streaming access patterns with no prefetching exploitation [17]. In few words, PIM solutions are appealing in contexts in which caches are poorly or not exploited at all.

Chapter 3
Abstract Machine Models for Processing-in-Memory

In this chapter, all the concepts, notions and issues about Processing-in-Memory, learnt previously and referring to the supporting literature, will be exploited in order to derive abstract models for PIM architectures.

More specifically, in order to take into account all the main fundamental aspects that characterize a target PIM system and neglect/abstract the minor or the useless ones, we will exploit a well-known formalism in Computer Science: the *Abstract Machine Model* or *abstract architecture*.

A brief reminder to structured parallel computation concepts and performance cost models, in relation to the abstract architecture, will be reported as well in order to illustrate and elucidate the methodology of study pursued in this Thesis that, in turn, has been inherited and it is widely detailed in [1].

3.1 Background: reminder on structured parallel programs, abstract machine and cost models

Let us briefly remind preliminary yet important concepts that characterize our structured approach of study in order to provide a background for a not expert reader.

An important peculiarity of our methodology is to conceive a computational system as organized/structured by vertical independent levels. Thus, a key concept is the following one:

- parallel applications are developed through high-level and architecture independent programming paradigms;
- the impact of the underlying architecture is captured by some parameters, in relationship to the process execution and cooperation mechanisms, used to express the *performance cost model* associated to the parallel application. Notable examples of architecture dependent parameters are the interprocess communication latency, indicated as L_{com}, and the process mean calculation time, indicated as T_{calc}.

A performance cost model is a simple yet significant analytical formulation that allows to estimate and predict typical performance parameters, like parallelism degree, processing bandwidth, completion time, etc., that characterize the performance evaluation of a given parallel program.

In order to take into account all the relevant issues and the cost model definition, we exploit an Abstract Machine Model (AMM) or abstract architecture. An AMM is a simplified view of several and different physical architectures which is able to describe the essential performance properties and to abstract all the useless ones. Moreover, these models are typically intended as a useful "communication bridge" between application developers and hardware architects during a co-design process [25].

A key concept of the AMM exploitation, in relationship to the cost models, is the following one:

- The *specificity* of individual concrete architectures is expressed by the value of *some key parameters* of the cost model, e.g. communication latency and process calculation time.

In this way, not only the application developers are able to focus on the aspects of the target architecture that are relevant to structure their algorithms, in order to maximize performance and energy efficiency [25], but the parallel compiler is also able to "see" the AMM and optimize parallel applications by applying the associated cost models.

Structured Parallel Computations

Another important aspect of our methodology of study is related to the parallel program we will deal with; as a matter of fact, we will exploit *structured parallel program* examples, i.e. programs made up of a limited set of *parallel paradigms*, in order to reduce the parallelization complexity, enhance cost models effectiveness and the performance predictability.

As detailed in [1, 47] and here reported, the parallel paradigms, or *parallelism forms*, are schemes of parallel computations with the following important features:

1. they restrict the parallel computation structure to a limited set of predefined

patterns;

2. they have a precise semantics;
3. they are characterized by a specific cost model;
4. they can be composed each other to form different and more complex parallel computations.

Exploiting this approach, the parallel programmer is free from low-level details and can reason only on computational aspects having an abstract high-level view of the parallel application and, furthermore, the availability of a set of parallel paradigms as useful building-blocks for develop it.

A notable and widely recurrent parallel pattern is the so-called *farm* paradigm which consists of the functional replication of a sequential *stateless* computation, such that distinct input stream items can be processed, in parallel, by different, independent and identical functional modules, called *workers*. The following figure illustrates a graphical representation of the farm paradigm logical organization: an input data scheduling module called *emitter*, *n* workers and a data collecting module called *collector*.

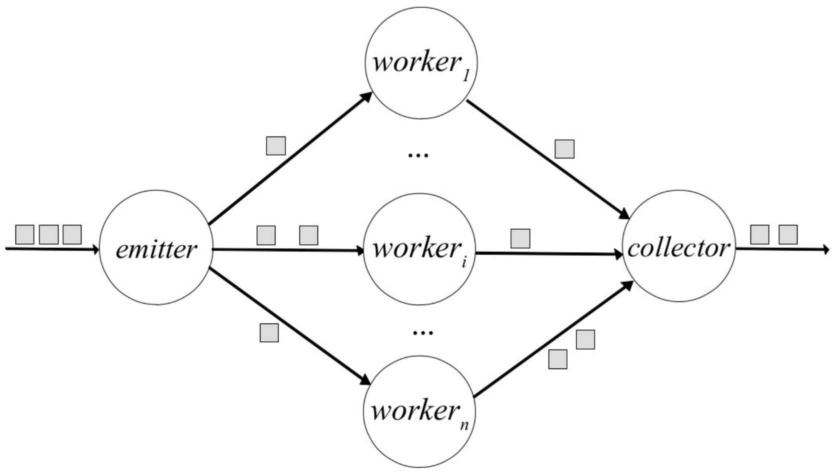

Figure 3.1. The farm parallel paradigm.

As it can be observed in the figure, every parallel paradigm is characterized by a well-defined logical structure; in particular, in the farm case, the data distribution, the parallel functional computation and the data collection stages are clearly distinguished.

3.2 General Abstract Machine Models for PIM architectures

Let us now consider Abstract Machine Models for PIM architecture by explicitly distinguishing the two possible architectural organization variants: the single-host and the multi-host PIM architecture.

3.2.1 Single-host PIM architecture

The following figure represents the abstract model of a PIM architecture, e.g. it could be part of a single cluster node in a cluster of workstations, which contains a single many/multi-core processor, called *host*, different high-bandwidth 3D-stacked memories, indicated as *3D Memory Units* (3DMUs), and the interconnection networks.

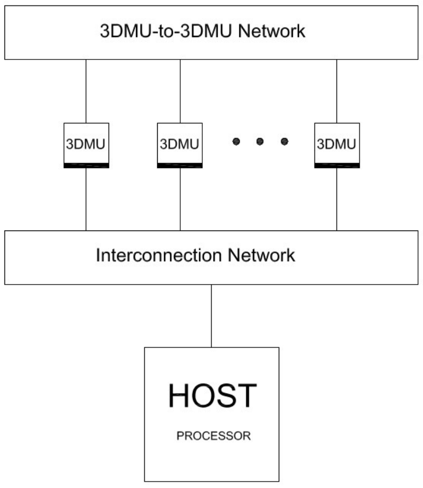

Figure 3.2. The general Abstract Machine Model for a single-host PIM architecture.

Let us study every system component singularly by considering only the details that are relevant at this level of abstraction:

- *Host Processor*

  A multi or many-core processor, according to the computational context in which it is exploited, each one characterized by a cache hierarchy of *at least* two levels and by a certain clock-cycle latency τ (which is the inverse of the clock frequency f). We will indicate with $P_{host\text{-}PE}$ the number of cores, or simply *Processing Elements* (PEs). Moreover, we will assume that each host PE has some local I/O units and/or co-processors, dedicated to interprocessor communications (i.e. they are not

peripheral devices controllers) and/or to the execution of specific run-time support functionalities, and an interface unit W, which stands for “Wrapping” unit, that plays several important architectural roles, such as interfacing the host core with the external interconnection network (i.e. the host processor's on-chip network).
A graphical representation of a host core is shown in the following picture:

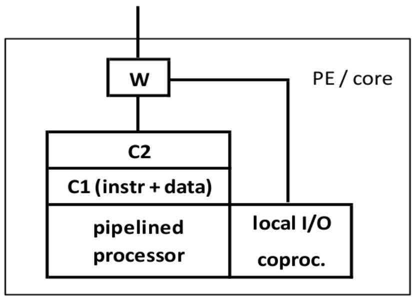

Figure 3.3. The internal structure of a host core (*taken from* [1]).

- *Interconnection Networks*

  In figure 3.2 we can distinguish two kind of interconnection networks: the Host-to-3DMUs one, simply indicated as *interconnection network*, that, as the name suggests, connects the host processor to the 3DMUs with *average distance* d_{net}. Instead, the *3DMU-to-3DMU network*, also referred as *memory network*, is used to interconnect all the 3D memories within the system and it is characterized by an average distance indicated as $d_{mem\text{-}net}$. It is interesting to note that different variants are admissible; as an example, the two networks can actually coincide if we use HMC-like 3D memories, as we will assume in the following, and the chaining facility is exploited realizing, for instance, a *daisy-chain* topology. If this is the case, then we have: $d_{net} \sim d_{mem\text{-}net}$.

- *3D Memory Units*

  Each 3DMU is an high-bandwidth 3D-stacked memory that consists of a logic layer at the bottom side of the stack, containing, among the other logic, a PIM processor, and different memory layers. The total number of 3DMU in a given system is indicated with N_{3DMU}. The description of a 3DMU with an underlying PIM processor, i.e. a *PIM-enabled* 3DMU, deserves a wider and more accurate detailing and, therefore, is postponed to the following sub-section.

In figure 3.2, some components are missing but they are anyway present in every system that, for instance, constitutes a single node of a cluster of workstations. As an example, in addition to the high-bandwidth 3D memories, that constitute the primary level of a main memory hierarchy, larger capacity and lower bandwidth secondary-level memory technology, e.g. standard DRAM or the new NVRAM (Non-Volatile RAM), could be present as well. Thus, 3D memories behave as large caches for slower levels of the memory hierarchy. This would require proper low-level mechanisms that manage this “multi-level cached” memory system [25].

Furthermore, a *Network Interface Card* (NIC) is obviously present and it could be directly integrated in the host processor chip package, as for the new *Xeon Phi Knigths Landing* processor, or it could be a discrete system component. In this last case, it can be reached through a typical *PCI-express* bus or by more efficient interconnection (as in the case of *Infiniband* technologies).

PIM-enabled 3D Memory Unit

The abstract model for a PIM-enabled 3DMU is shown in the following figure:

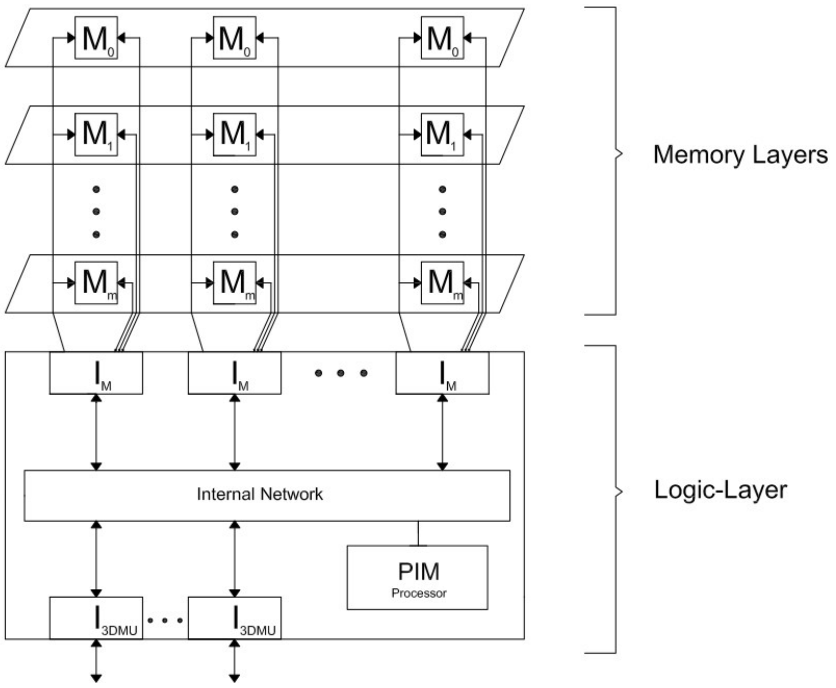

Figure 3.4. Abstract modelling of a PIM-enabled 3DMU.

As it can be observed, each 3DMU consists of a set of vertical *memory slices*, each one modelled as a memory macro-module, with *m* interleaved memory modules, and the accesses to it are managed by an underlying memory interface/controller unit I_M. Each memory module, within a given memory slice, is located in a different memory layer of the stack and, through the TSV (Through-Silicon Via) vertical connection, it is directly connected to the relative I_M in the logic layer. Thus, the maximum interleaved memory bandwidth for every memory slice is m/τ_M, where τ_M is the memory access time.

The unit I_M is internally characterized by an associative cache component used to buffer the most recently accessed information, thus realizing a *caching in memory*, and by a *Memory Request Queue* (MRQ), containing outstanding memory access requests that can be served according to a non-FIFO strategy in order to exploit caching in memory of previously referred blocks.

Each unit I_M is then connected to an internal logic-layer network that could be a crossbar, as in the case of HMC memories, or a limited-degree interconnection such as a *generalized fat tree* (similar to the logarithmic *Mesh-Of-Trees* network proposed in [50]). In particular, with these specific solutions, possible contentions in the on-chip logic-layer network are minimized [1].

In addition to the memory interfaces and to the PIM processor (if, as in our case, a *PIM-enabled* 3DMU is exploited) the logic-layer internal network is also connected to a set of units I_{3DMU} that interface the 3DMU to the external system components. As an example, if the 3DMU is an HMC-like memory, as we will assume in the next chapter, then each I_{3DMU} is a SerDes (Serializer/Deserializer) interface that allows to connect the 3DMU to a host processor, or to another 3DMU, through a SerDes link. Therefore, each I_{3DMU} has routing capabilities for incoming messages, directed to a certain memory slice, and switching capabilities for pass-through messages, that have to be forwarded to a directly connected 3DMU.

Let us now study the PIM logic embedded in the logic layer of a 3D memory unit. As a matter of fact, we will assume that the PIM processing logic is a PIM processor with $P_{pim\text{-}PE}$ energy-efficient, e.g. ARM-like, cores/PEs. According to the information provided by the literature, $P_{pim\text{-}PE}$ is at most 16 with current semiconductor chip fabrication widths and thermal constraints (although, according to [23], they are not so stringent).

As for other research works, each PIM core is characterized by a one-level cache hierarchy and by the same ISA (*Instruction Set Architecture*) of the host processor. Moreover, we will assume that the internal PIM core structure, apart from the secondary-level cache C2, is the same of the host PE (shown in figure 3.3), and that the clock latency is τ as for the host cores (anyway, we could have cases for which $\tau_{host} \neq \tau_{pim}$ and, more realistically, $\tau_{host} < \tau_{pim}$).

Finally, in the following, we will indicate the total number of PIM cores in the whole system with N_{pim}; the same definition applies for N_{host} that, in the single-host architecture case, coincides with $P_{host\text{-}PE}$. Furthermore, we will use P_{pim} to indicate the total number of PIM processors in the system; it goes without saying that $P_{pim} = N_{3DMU}$ if all the 3DMUs are PIM-enabled, otherwise $P_{pim} < N_{3DMU}$.

3.2.2 Multi-host PIM architecture

The following picture represents the abstract architecture of a PIM system that contains different host processors.

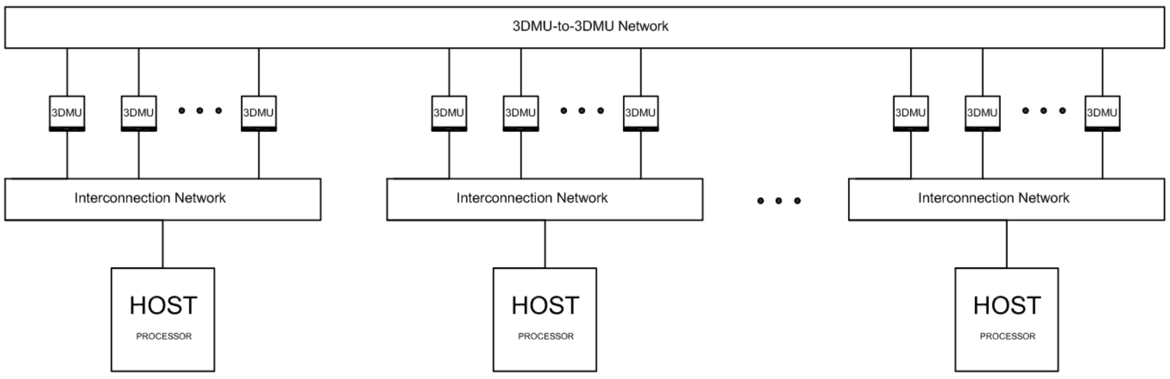

Figure 3.5. The general Abstract Machine Model for a multi-host PIM architecture.

The definition of previously presented key parameters, such as N_{pim}, N_{host}, $P_{pim\text{-}PE}$. $P_{host\text{-}PE}$, N_{host}, etc., applies also to this case. Furthermore, we will use P_{host} to indicate the total number of host processors in the system (obviously, in the single-host case $P_{host} = 1$).

Anyway, in this PIM architectural variant, we should distinguish different single-host logical sub-systems, each one with its own d_{net}, i.e. average distance host-to-3DMUs network, and $d_{mem\text{-}net\text{-}local}$, i.e. average distance 3DMU-to-3DMU *local* network (it is not shown in the picture above but it can be logically thought as a sub-network of the 3DMU-to-3DMU global network; obviously, in a concrete implementation they are distinct, as we will see in Chapter 4).

Finally, we can define $d_{mem\text{-}net\text{-}global}$ as the average distance of 3DMU-to-3DMU *global* network (i.e. the one shown in the picture above).

In the following chapter, all the parameters defined in this section will be exploited, and their values fixed, in the presentation of *parametrized* abstract architecture models *closer* to concrete PIM systems.

Chapter 4

PIM Architectures and Low-Level Communication Latencies Cost Models

In this section, parametrized versions of the abstract machine models for PIM architectures presented in the previous chapter will be described. Therefore, we will fix some architectural parameter values, with the ones provided by the literature, e.g. memory access time, memory capacity, interconnection network links bandwidth, number of PIM cores per PIM processor, etc., and we will derive important architecture-dependent parameters, such as communication latencies for cache-to-cache and memory blocks transfers, that we will use for expressing cost models of parallel program examples reported in Chapter 6.

A brief reminder of interprocess cooperation, low-level pipelined communications and cache coherence mechanisms will be reported as well.

4.1 Background: reminder on interprocess, inter-unit communications and cache coherence mechanisms

In this section we will briefly remind, always referring to [1], some important aspects related to interprocess communications, pipelined communications among system units/components, whose latencies are relevant to express key parameters for the interprocess cooperation cost models, and cache-coherence mechanisms, tied to the parallel program communications run-time support.

Base latency of inter-unit pipelined communications

As detailed in [1], parallel architectures frequently exploit forms of system-wide pipelined communication. Notable examples are the high-performance interconnection networks, e.g. *Infiniband*, exploiting the *wormhole* flow-control strategy, where data packets are decomposed into smaller units of few bytes, called *flits*, that, in turn, are transmitted following the <u>same</u> path in a *streaming pipelined* fashion.

Thus, let us consider the whole PIM system at hand as a network of processing units, such as host/PIM cores, interface and switching units, memory modules, caches, etc., and links, and let us assume we want to compute the latency of a communication among two units in the system, e.g. a Last-Level cache (LLC) unit of a given host core requesting a cache block to a 3D Memory Unit (which is a sub-system of units) after a LLC fault, then we should first of all recognize the communication *path* among the two cooperating units. The path is characterized by its distance *d*, which in general is an average measure, defined as the (average) number of processing units, or, equivalently, the number of links, belonging to the path. Let us indicate with *s* the length of the data stream that has to be transmitted, consisting of the number of sub-stream elements following the same path, e.g. it could be the number of flits of a data stream packet. Moreover, with $T_{hop} = \tau + T_{tr}$ we indicate the ***hop latency***, i.e. the latency of a "hop" in the path consisting of a pair (*unit*, *output link*), where τ is the single unit processing latency and T_{tr} is the output-link transmission latency (as a matter of fact, we will use the clock latency τ as the reference time unit for measuring all the latencies and service times that follow in this and in the next chapters).

The latency $L(s)$ spent for transmitting a stream of length *s* over a path of distance *d*, exploiting a pipelined (i.e. inter-unit pipeline-effect) communication, is equal to:

$$L(s) = (2s + d - 3)\ T_{hop}$$

The correctness of the above formula is graphically proved by the following picture, where it has been assumed s = 5 and d = 4:

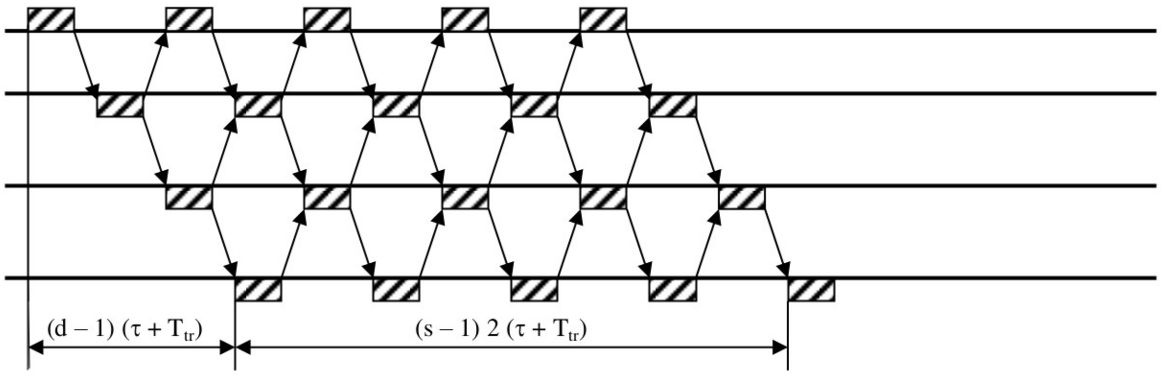

Figure 4.1: A pipelined communication scheme for transmitting a data stream of length s = 5 over a path of distance d = 4 (taken from [1]*).*

The above formula refers to the *single-buffering* communication protocol, as named in [1]. Actually, it can be substantially improved, as stated again in [1], by using a *double-*

buffering inter-unit cooperation scheme with a resulting communication latency equal to:

$$L(s) = (s + d - 2)\, T_{hop}$$

It should be noted that, in general, T_{hop} can vary. More specifically, for an inter-unit on-chip communication $T_{tr} \sim 0$ and $T_{hop} = \tau$; instead, when two units in different chips are involved, then $T_{tr} > 0$ and $T_{hop} > \tau$. If this is the case then, according to the pipeline scheme properties, the *maximum* T_{hop} has to be used in the above latency formula, in that T_{hop} represents the "hop" service time.

In the following sections, the presented latency formula will be extensively used to calculate all the communication latencies among PIM system components of interest.

Interprocess communications

In the examples of Chapter 6 we will consider structured parallel programs that consist of a collection of communicating processes exploiting a *message-passing*, or *local environment*, cooperation model. This kind of communication mechanism is characterized by an explicit exchange of values, among the involved processes, that do not share any variable, *at least* at the process level. Anyway, if the target architecture is a shared-memory one, as in the case of PIM, then the *run-time support*, i.e. the implementation, of the message-passing communication mechanisms can imply variable and/or data structure sharing according to the implementation adopted.

The following figure shows an example of two cooperating processes, e.g. according a *producer-consumer* communication pattern:

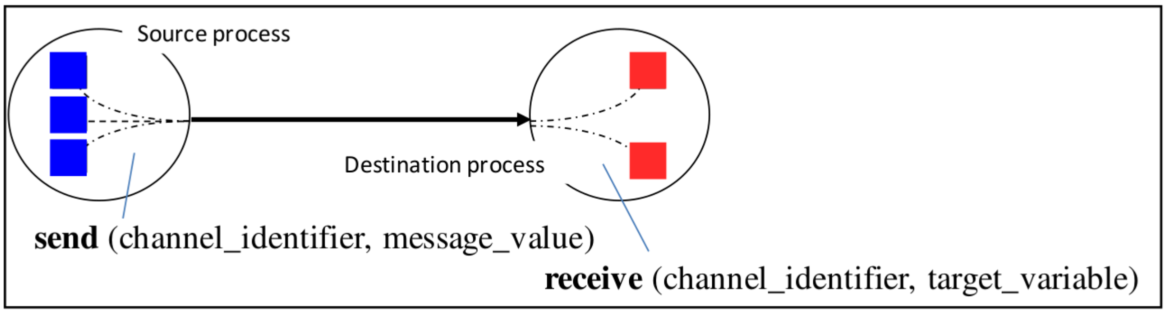
Source process
Destination process
send (channel_identifier, message_value)
receive (channel_identifier, target_variable)

Figure 4.2: A point-to-point communication scheme between two processes that cooperate according to the message-passing model (*taken from* [1]).

As can be observed, every process works with its local variable (represented in the figure as squares of different colours). When a communication takes place, each process has its own primitive to communicate over a given logical *channel* with *asynchrony degree k*, i.e. *send* and *receive* procedures for sender and receiver processes respectively. The asynchrony degree of a channel establishes the number $k \geq 0$ of *send* primitives that can be executed by the sender process without being blocked due to the missing *receive* primitives executions by the receiver process; thus, the case $k = 0$ corresponds to the synchronous communication.

The semantic of an interprocess cooperation that exploits a local-environment is such that, when the communication terminates, the target variable of the receiver process contains the message value sent by the sender process (in other words the message value is *copied* into the target variable).

Different implementations exist, more or less optimized according to the number of message copies required; in our case, we will assume a *zero-copy* communication, meaning that the number of message copies is reduced to the minimum, i.e. one. This solution is implemented in the most advanced message-passing libraries for which the *send-receive* primitives are executed in *user space*; thus, no degrading *kernel space* is involved.

The communication latency cost L_{com} of a *zero-copy* interprocess communication can be expressed as:

$$L_{com}(L) = T_{send}(L) + T_{receive} = 2T_{setup} + L\ T_{transm}$$

with:

- $T_{send}(L) = T_{setup} + L\ T_{transm}$
- $T_{receive} = T_{setup}$

where L is the message length (in general measured in words), T_{setup} is the latency of all run-time support actions *except* the message copy (i.e. sender-receiver synchronization, low-level scheduling, etc.), and T_{transm} is the latency for copying one word of the message. As a matter of fact, it can be noted that T_{transm} impacts only on the *send* latency in that, according to the *zero-copy* implementation semantic, the *send* primitive run-time support is responsible of copying the message value into the target variable.

It is interesting to note that the PIM models presented in Chapter 3 are ***all-cache*** architectures, meaning that each information is transferred into the primary cache before being used and, in general, the data transfer units are actually cache blocks. Moreover, we will assume that, for the host PEs, characterized by a cache hierarchy of at least two levels, every C1-block fault that is also a C2-block fault (and a C3-block fault if C3 is present) will be handled on a C1-block basis. In other words, in order to minimize memory and network congestion, C2 (or C3 if any) requests only one C1-block at a time to the memory.

Thus, it could be convenient to rewrite the above L_{com} formula in function of the primary cache block of size σ_1, i.e. :

$$L_{com}(L/\sigma_1) = T_{send}(L/\sigma_1) + T_{receive} = 2T_{setup} + L/\sigma_1\, T_{transm}(\sigma_1)$$

where $T_{transm}(\sigma_1)$ is the latency needed for transmitting a primary cache block of size σ_1, e.g. in a cache-to-cache communication or memory block transfer, and L/σ_1 is the message length measured in terms of C1-block units.

Last but not least, we will adopt the *I/O-based RDY/ACK* communication model proposed in [1]. Theoretically, the *send-receive* RDY/ACK implementation for an asynchronous channel with $k = 1$ is of the kind:

```
send(msg)::                                receive(vtg)::
    wait_until(ACK);                           wait_until(RDY);
    copy msg into vtg;                         use vtg;
    notify(RDY);   // set RDY = 1              notify(ACK);   // set ACK = 1
```

where *wait_until* and *notify* are sender-receiver low-level synchronization primitives, *msg* and *vtg* are symbols referring to the message value and to the target variable respectively, ACK (standing for *acknowledge*) and RDY (standing for *ready*) are binary boolean variables with the following meaning:

- RDY = = 1 → new message value present (at the beginning RDY is initialized to 0, is set to 1 by the *send* implementation, with the *notify* primitive, and it is set again to 0 at the end of the *receive* execution);
- ACK = = 0 → previous message received (at the beginning ACK is initialized to 1,

is set to 0 by the *send* implementation, after *msg* has been copied into *vtg*, and it is set again to 1 by the receive execution with the *notify* primitive).

Moreover, as said before, we will adopt the I/O-based RDY/ACK communication model, meaning that synchronizations, i.e. RDY and ACK notifications, are performed by means of asynchronous interprocessor communications, involving I/O mechanisms and interrupt management. In this way, sender-receiver synchronization is greatly simplified and it is more efficient with respect to the one that exploits a *retry*-based solution over shared synchronization variables (for an extensive explanation of the RDY/ACK solutions we refer to [1_Sec.23]).

Cache coherence mechanisms

Cache coherence (CC) mechanisms in PIM architectures are still an open research problem, as confirmed by [3]. For this reason, among all the coherence approaches/solutions detailed in [1], we will choose, for our PIM system models, the one that minimizes data movement, coherence traffic and contentions, i.e. the *automatic directory-based* CC protocol with the *home-flush* and *local/self-invalidation* optimizations.

According to the chosen solution, the directory partition is maintained in primary cache for PIM cores and in secondary cache for host cores. Therefore, cache controllers of C1 and C2, for PIM and host PEs respectively, will be used to implement the CC protocol.

Moreover, the *home-flush* and *local/self-invalidation* optimizations can be exemplified by a simple yet significant example: a producer process P sends a message value *msg*, of size σ_1, to a consumer process C. According to what said before, if a RDY/ACK communication model is exploited, then P and C behave as follows:

P:: …; copy *msg* into *vtg*; *notify*(RDY); …
C:: …; …; *wait_until*(ACK); use *vtg*; …

where "copy *msg* into *vtg*" implies one or more STORE assembler instructions, performed by P, over a shared primary cache block b (i.e. the one that will contain *msg* and, therefore, the same that will be read as *vtg*). Let us indicate the *last* one of them as STORE_P b. Moreover, the "use *vtg*" action implies one or more LOAD assembler instructions, performed by C, over the same shared block b. Let us indicate with LOAD_C b the *first* one of them. Assuming that b is used no more by P after $STORE_P$ b and that b is shared by P and C only, then, in a *basic-invalidation* solution, $LOAD_C$ b would generate a fault with a cache-to-cache block transfer between the nodes used to allocate P and C, i.e. PE_P and PE_C respectively (more precisely, by their cache controllers in charge of implementing the CC protocol). Moreover, if there was a $STORE_C$ b by the C process, then this would result in an invalidation request, i.e. explicit communication, from PE_C to PE_P and, on the reverse side, an invalidation reply/acknowledge from PE_P to PE_C. This fact is a source of possible contentions over cache controllers of processing modules with more than one symmetric, i.e. point-to-point, communication channel, e.g. the *emitter* of a *farm* parallel pattern, presented in Chapter 3, with every $worker_i$ that is home-node of the respective vtg_i blocks.

Conversely, if the home-flush and local/self-invalidation optimizations are exploited, and PE_C is the *home-node* of b (i.e. the cache controller of the PE_C cache is in charge of maintaining the directory partition for b and other blocks), then $STORE_P$ b causes a C2C transfer between PE_P and PE_C caches (home-flush) and the local invalidation, i.e. deallocation, of b in PE_P cache. In this way, no fault is generated by $LOAD_C$ b and no invalidation is needed in case of a $STORE_C$ b execution by C.

In conclusion, the home-flush and local/self-invalidation optimizations minimize possible contentions, and, if the data are useful for the home node computation, then they also minimizes the block reading latency (for an extensive explanation of the *home-flush* solution and other cache coherence solutions we refer to [1_Sec.20]).

4.2 PIM architectures specifications

Let us now consider *parametrized* versions of the abstract PIM architectures presented in previous chapter by explicitly distinguishing, again, the two possible variants: the single-host and the multi-host PIM architectures.

The two models detailed in the following sub-sections, with their respective parameters, will be then reused for successive base communication latencies cost models and, in Chapter 6, for the conclusive analysis of parallel program examples targeting PIM systems.

4.2.1 Single-host PIM architecture

The following figure represents a PIM architecture with a typical "halo"- like organization, consisting of a host processor, four *PIM-enabled* 3DMUs and two interconnections: a Host-to-3DMUs direct connection and a 3DMU-to-3DMU ring memory network.

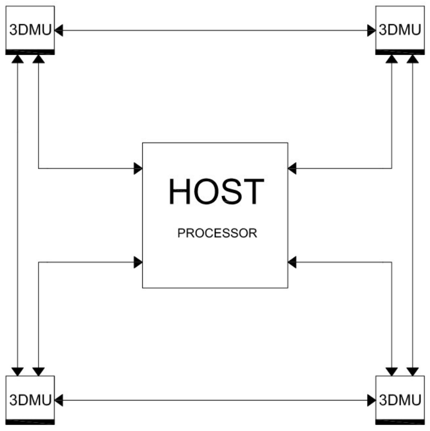

Figure 4.3: A single-host PIM architecture with a "halo"- like organization.

Let us study every system component in detail in order to provide precise architectural specifications:

- *Host Processor*

  A many-core processor with: $P_{host\text{-}PE}$ = 64 cores, each one with private, inclusive, on-demand two-level caching, where C1 has a capacity of 32 KB with primary cache block size σ_1 = 32 Bytes and C2 is large 512 KB with σ_2 = 64 Bytes, a two-dimensional mesh interconnect, with ten switch nodes for every dimension (10-ary 2-cube), four MINF (*Memory Interface*) units, directly attached to the four 3DMUs by dedicated bidirectional links, and two I/O-INF (*I/O-interface*) units.

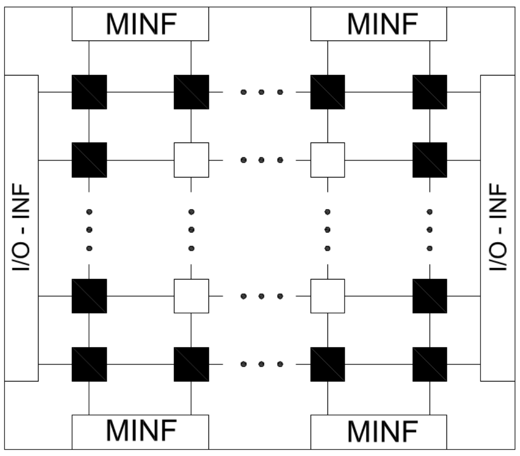

Figure 4.4: The host processor's internal structure.

The general organization of the host processor is graphically shown in the picture above; as it can be noted, the switch units at the extreme sides, coloured in black, are used only for MINF and I/O-INF connections. Therefore, the 64 host PEs, not shown in the figure, are attached to the internal switch units (the one left in white).
Last but not least, the clock frequency f of every core is equal to 1 GHz (therefore, the clock latency $\tau = 1$ nsec) and, furthermore, every PE/core is characterized by an internal *communication co-processor KP*, in addition to the main processor *IP*, dedicated/specialized to the execution of run-time support functionalities, in particular the *send* primitive. In this way, in a stream-based application, if the time spent to send a stream element is comparable to the time needed to compute the successive item, then it can be overlapped. To do this, when *IP* has to execute the *send* primitive over an *asynchronous* channel, it delegates this task to *KP* and continues its execution by computing the successive stream element.
Equivalently, if *KP* is not present, we can exploit a solution, detailed in [48], with a *communication thread* acting as *KP*, provided that the architecture of every PE/core is *multithreaded*.

- *PIM processor*

  A multi-core processor with $P_{pim\text{-}PE} = 16$ cores, each one with private, on-demand primary cache C1 with capacity 32 KB, a bidirectional crossbar interconnect, a single MINF unit that is directly connected to the logic-layer network of the *PIM-enabled* 3DMU in which the PIM processor resides. As for the host PEs, each PIM core has a clock frequency $f = 1$ GHz and a local communication co-processor *KP* (if feasible, otherwise, as said before, a communication thread).

- *Interconnection Networks*

  The data exchange over the interconnection networks exploit a *wormhole* flow-control strategy, with a size per flit of 4 Byte, and a packet switching minimal (deterministic or adaptive) routing. All the inter-chip links have a transmission latency of $T_{tr} = \tau$, thus $T_{hop} = 2\tau$, instead, intra-chip links have $T_{tr} \sim 0$ and, therefore, $T_{hop} = \tau$. Moreover, the inter-unit cooperation scheme is the *double-buffering* one.

  The external interconnection networks, i.e. the ones shown in figure 4.3, are made up of Ser/Des-like links, with 80 GB/sec sustainable bandwidth (i.e. for $T_{tr} = \tau$, as previously listed, then 20 flits at a time can be transmitted).

- *3D Memory Units*

  Without loss of generality, we will assume HMC-like 3DMUs; thus, the maximum offered bandwidth, the memory capacity, the links adopted (i.e. Ser/Des links) and other related aspects refer to what reported in the last HMC specification [42]. Therefore, each 3DMU has a capacity of 8 GB, 8 memory layers, one logic-layer and 32 memory-slices, each one with a capacity of 256 MB.

  Every memory slice has $m = 8$ interleaved memory modules, each one located in a different memory layer of the stack. Every memory module provides a bandwidth of one flit per access time τ_M; thus, considering all the m modules, the maximum offered bandwidth is σ_l/τ_M. Taking $\tau_M \sim 3\tau$ and, as previously listed, for σ_l equal to 32 Bytes, the maximum offered bandwidth per memory slice is about 10 GB/sec; instead, if we consider the whole 3DMU (32 memory slices), then it is 320 GB/sec. These performance values are compliant to what reported in the last HMC specifications [42].

  Finally, the logic-layer of each 3DMU is characterized by four interfaces I_{3DMU}, that allow to interconnect the 3DMU with external devices, e.g. host processor and/or other 3DMUs, etc. The logic-layer internal interconnect is a bidirectional crossbar.

Summarizing, we have: $P_{host\text{-}PE} = N_{host} = 64$, $P_{pim\text{-}PE} = 16$, $N_{3DMU} = P_{pim} = 4$, $N_{pim} = 64$ and, obviously, $P_{host} = 1$ (where, from Chapter 3, N_{host} and N_{pim} are, respectively, the total number of host cores and PIM cores in the system, P_{pim} is the total number of PIM processors, coinciding with the total number of 3DMUs, i.e. N_{3DMU}, in that they are all *PIM-enabled*, and P_{host} is the total number of host processors, here merely one).

4.2.2 Multi-host PIM architecture

The following figure represents a multi-host PIM architecture with four single-host logical sub-systems, each one with the same organization of the previous single-host PIM model.

Anyway, it should be noted that, in this case, we can distinguish three interconnections; in addition to the direct connection Host-to-3DMUs and the ring *local* memory network of each single-host sub-system, a global ring memory network of 16 3DMUs is also present.

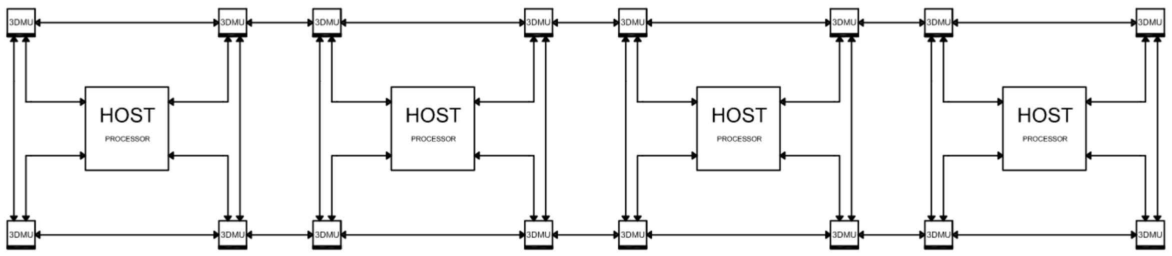

Figure 4.5: A multi-host PIM architecture.

Every system component has the same characteristics of the single-host PIM architecture; thus, we refer to the previous section for the specific description.

The only difference, in addition to the quadrupled number of 3DMUs, host and PIM processors, is that every host is a multi-core processor with $P_{host\text{-}PE}$ = 16 PEs and it is characterized by an internal two-dimensional mesh of six nodes per dimension (i.e. 6-ary 2-cube).

Summarizing, we have: $P_{host\text{-}PE} = 16$, $P_{pim\text{-}PE} = 16$, $N_{3DMU} = P_{pim} = 16$, $N_{host} = 64$, $N_{pim} = 256$ and $P_{host} = 4$.

4.3 Inter-unit communication latencies cost models

In the following sections, cost models associated to the inter-unit base communication latencies will be evaluated. In so doing, we will take into account only the cooperation that takes place between units of interest, i.e. cache-to-cache communications and memory blocks transfers, and we will exploit the architecture specification listed in previous sections (i.e. section 4.2.1 for the single-host PIM architecture, and section 4.2.2 for the multi-host one).

Notice that, the following numerical results have to be intended in no way as precise reference values; they are just used for acquiring a general knowledge of order of magnitudes, for comparing alternative solutions and for exemplifying the concrete application of low-level communication latencies cost models.

4.3.1 Single-host PIM architecture

As said before, data exchange over interconnection networks exploit a low-level/firmware packet switching protocol. Every packet is transmitted as a *stream of flits*, of which the first is the packet *header*, and the remaining are *value* flits. The header flit contains all the useful routing information, notably: source unit id, destination unit id, packet length, message type (e.g. read request, read reply, write request, etc.); instead, the value flits contain data that have to be exchanged, e.g. a primary cache block of size σ_1, and, in case of a read or a write request message, the physical address of the referenced information.

Assuming that a physical address is represented by 8 Bytes, i.e. it fits into 2 value flits, and, as listed before, knowing that a primary cache block is of size $\sigma_1 = 32$ Byte, i.e. 8 flits, then we are interested in evaluating the following base inter-unit communication latencies (notice that in the following sub-sections we will use $\sigma_1 = 8$ for computing base communication latencies).

Memory block reading base latency

Exploiting the pipelined communication latency formula of section 4.1, the latency for a C1-block transfer from memory is given by:

$$L_{read\text{-}C1}(\sigma_1) = L_{read\text{-}C1\text{-}req} + L_{read\text{-}C1\text{-}reply}(\sigma_1)$$

with:

- $L_{read\text{-}C1\text{-}req} = (s_{req} + d - 2)\ T_{hop}$
- $L_{read\text{-}C1\text{-}reply}(\sigma_1) = (s_{reply} + d - 2)\ T_{hop} + \tau_M = (\sigma_1 + d - 1)\ T_{hop} + \tau_M$

where τ_M is the memory access time, $s_{req} = 3$ (header + physical block address), d is the distance of the traversed path and s_{reply} is the response message length, i.e. $s_{reply} = 1 + \sigma_1$ (header + C1 requested block).

Distinguishing according to the kind of PE that performs the read request, i.e. PIM or host PE, we have the following base latency costs:

- *PIM core - memory block reading (within the same 3DMU in which it resides)*

  In this case, a PIM core performs a read request of a C1-block to a memory slice of the *PIM-enabled* 3DMU in which it is located. Thus, the following path of system units is traversed by the request and reply messages: C1, W, PIM-net-int (PIM processor internal network), MINF, ll-net-int (logic-layer internal network), I_M, M (i.e. destination memory slice).

Therefore, $d = 5 + d_{PIM\text{-}net\text{-}int} + d_{ll\text{-}net\text{-}int} = 7$, with $d_{PIM\text{-}net\text{-}int} = d_{ll\text{-}net\text{-}int} = 1$ since the related networks are crossbars, and $T_{hop} = \tau$ wit $T_{tr} \sim 0$ always (i.e. all intra-chip communications). Finally, we have:

$$L_{read\text{-}C1\text{-}PIM}(\sigma_1) = L_{read\text{-}C1\text{-}req} + L_{read\text{-}C1\text{-}reply}(\sigma_1) = 25\tau$$

- *Host core - memory block reading*

  The internal organization of the host processor, described and represented in section 4.2, can be logically thought as a NUMA one, in which each MINF is logically associated to a subset of host cores. Therefore, the whole 2D-mesh network (i.e. a 10-ary 2-cube network) can be logically split into four sub-meshes (5-ary 2-cube) such that, with good approximation, the large majority of accesses by a PE, located in a certain sub-network j, will be directed to MINF$_j$, with $j = 1\dots4$, without crossing other sub-meshes.

  The average sub-mesh distance traversed is $d_{host\text{-}net\text{-}int} = 5$ since, for a k-ary n-cube network, with k > n, the average distance is about: $n/2 \sqrt[n]{N}$ (in our case $N = 25$ and $n = 2$).

  The traversed path by request and reply messages will be: C1, C2, W, host-net-int, MINF, I_{3DMU}, ll-net-int, I_M, M.

  Thus, $d = 7 + d_{host\text{-}net\text{-}int} + d_{ll\text{-}net\text{-}int} = 13$, with $d_{ll\text{-}net\text{-}int} = 1$ since the logic-layer network is a crossbar, and $T_{hop} = 2\tau$ in that $T_{tr} = \tau$ (intra-chip communications are involved and, therefore, we must take the maximum T_{hop}).

  Finally, we have:

$$L_{read\text{-}C1\text{-}host}(\sigma_1) = L_{read\text{-}C1\text{-}req} + L_{read\text{-}C1\text{-}reply}(\sigma_1) = 71\tau$$

As it can be observed, the base latency for a memory block reading performed by a host core within the given single-host PIM architecture, whose specifications are reported in section 4.2.1, is about three-fold (3x) the one needed by a PIM core.

Obviously, the same obtained base latency values are valid for synchronous memory block writing operations. Conversely, in the case of an asynchronous memory write, only the write request latency must be considered (i.e. $L_{write\text{-}C1\text{-}req}(\sigma_1) = (s_{req} + d - 2)\, T_{hop} + \tau_M$ with $s_{req} = 3 + \sigma_1$, i.e. header + physical address + primary cache block).

Cache-to-Cache (C2C) block transfer base latency

In automatic cache-coherence architectures, proper low-level/firmware mechanisms should be provided for cache block transfers between the involved nodes (i.e. the ones for which caches must be coherent). Such mechanisms are also indicated as *cache-to-cache (C2C) communications*.

As we have seen in section 4.1, a C2C block transfer can occur after a LOAD with fault or after a STORE in the *home-flush* optimization (a proper assembler annotation is inserted by the compiler if the STORE implies a *flush* of the referred block).

Therefore, let us evaluate the C2C communication latency that take place between two nodes, the *home* and the *requestor* ones (assuming that they are distinct), after a LOAD with fault; obviously, the same result value is valid for a synchronous block flush operation (otherwise, if the flush is asynchronous, we consider only the latency for a cache block transfer request, with $s_{req} = 3 + \sigma_1$). Two main situations can be distinguished in the execution of a LOAD with fault of the referred block b:

1) the block b is in the home node cache → b is transferred from the home node to the requestor via C2C;
2) the block b is not allocated in the home node cache → the home node reads b from the main memory, with a base latency equal to $L_{read\text{-}C1}(\sigma_1)$, and sends it to the requestor node.

It goes without saying that the base latency associated to case 2) is the same of 1) with the addition of $L_{read\text{-}C1}(\sigma_1)$.

The general latency formula for a C2C block transfer is very similar to the one used for memory block transfer, except the fact that, in the first case, we do not consider any memory access time τ_M. Thus:

$$L_{C2C}(\sigma_1) = L_{C2C\text{-}req} + L_{C2C\text{-}reply}(\sigma_1)$$

with:

- $L_{C2C\text{-}req} = (s_{req} + d - 2)\ T_{hop}$
- $L_{C2C\text{-}reply}(\sigma_1) = (s_{reply} + d - 2)\ T_{hop} = (\sigma_1 + d - 1)\ T_{hop}$

Therefore, let us evaluate the C1-block transfer base latency of a C2C communication between two distinct PEs by distinguishing the following cases:

1) *PIM-to-PIM cores <u>local</u> C2C block transfer (i.e. within the <u>same</u> PIM processor)*

   The request message, with s_{req} = 3 (header + physical address), travels the path: $C1_{source}$, W_{source}, PIM-net-int (PIM processor internal network), W_{dest}, $C1_{dest}$.

   Thus $d = 4 + d_{PIM\text{-}net\text{-}int} = 5$, $T_{hop} = \tau$ because of $T_{tr} \sim 0$ always.

   Finally, we have:

   $$L_{C2C\text{-}PIM\text{-}local}(\sigma_1) = L_{C2C\text{-}req} + L_{C2C\text{-}reply}(\sigma_1) = 18\tau$$

2) *PIM-to-PIM cores <u>remote</u> C2C block transfer (i.e. <u>different</u> PIM processors)*

   In this case, the path is: $C1_{source}$, W_{source}, PIM-net-int, $MINF_{source}$, ll-net-int, I_{3DMU}, memory-net, I_{3DMU}, ll-net-int, $MINF_{dest}$, PIM-net-int, W_{dest}, $C1_{dest}$.

   Thus $d = 8 + 2d_{PIM\text{-}net\text{-}int} + 2d_{ll\text{-}net\text{-}int} + d_{mem\text{-}net} = 13$ and $T_{hop} = 2\tau$.

   Notice that $d_{mem\text{-}net} = 1$ since the 3DMU-to-3DMU memory network is a ring with 4 3DMU nodes (the average distance of a ring network topology is calculated as $N/4$, where N indicates the number of nodes).

   Finally, we have:

   $$L_{C2C\text{-}PIM\text{-}remote}(\sigma_1) = L_{C2C\text{-}req} + L_{C2C\text{-}reply}(\sigma_1) = 68\tau$$

3) *host-to-host cores C2C block transfer*

   When at least one host PE is involved, then we should also consider the secondary cache C2 in the traversed path. Therefore, the request and reply messages cross the following path of units: $C1_{source}$, $C2_{source}$, W_{source}, host-net-int (host processor internal network), W_{dest}, $C2_{dest}$,$C1_{dest}$.

   Thus, $d = 6 + d_{host\text{-}net\text{-}int} = 14$, where, this time, $d_{host\text{-}net\text{-}int} = 8$ in that we have to consider the whole sub-mesh whose switch nodes are connected to the host cores (i.e. it is a 8-ary 2-cube network, the one represented in figure 4.4 with internal nodes left in white).

   Obviously, $T_{hop} = \tau$ because of $T_{tr} \sim 0$ always (intra-chip communication).

   Therefore, we have:

$$L_{C2C\text{-}host}(\sigma_1) = L_{C2C\text{-}req} + L_{C2C\text{-}reply}(\sigma_1) = 36\tau$$

4) *host-to-PIM cores C2C block transfer*

In this case, the path is (assuming that the source is a PIM core): C1$_{source}$, W$_{source}$, PIM-net-int, MINF$_{source}$, ll-net-int, I$_{3DMU}$, MINF$_{dest}$, host-net-int, W$_{dest}$, C2$_{dest}$,C1$_{dest}$. Thus $d = 8 + d_{PIM\text{-}net\text{-}int} + d_{ll\text{-}net\text{-}int} + d_{host\text{-}net\text{-}int} = 18$ and $T_{hop} = 2\tau$.

Therefore, we have:

$$L_{C2C\text{-}host\text{-}pim}(\sigma_1) = L_{C2C\text{-}req} + L_{C2C\text{-}reply}(\sigma_1) = 88\tau$$

In conclusion, it is interesting to stress the great advantage of C2C communications when they are performed on-chip.

4.3.2 Multi-host PIM architecture

As said before, a multi-host PIM architecture is actually composed of many single-host logical sub-systems, each one with the same organization of the previous single-host PIM model. Thus, the base latencies of all the communications that take place within a certain single-host logical sub-system are, in principle, the same of the previous case. Actually, we should evaluate again the local communication latencies that involve the host processor in that the internal interconnect is now a 6-ary 2-cube one; therefore: $L_{read\text{-}C1\text{-}host\text{-}local}(\sigma_1) = 63\tau$, $L_{C2C\text{-}host\text{-}local}(\sigma_1) = 28\tau$ and $L_{C2C\text{-}host\text{-}pim\text{-}local}(\sigma_1) = 72\tau$.

Let us now briefly evaluate all the communication latencies, to which we are interested in, that involve the global memory network (i.e. a ring with 16 nodes) and that, therefore, depend on $d_{mem\text{-}net\text{-}global} = 4$.

- *Host core – remote memory block reading*

  When a host PE has to read (or write synchronously) a primary cache block from a remote 3DMU (i.e. of a different single-host logical sub-system), then the base access latency is: $L_{read\text{-}C1\text{-}host\text{-}remote}(\sigma_1) = 87\tau$, with $d = 11$ and $T_{hop} = 2\tau$.

- *PIM-to-PIM cores <u>remote</u> <u>system</u> C2C block transfer*

  For a C2C communication among two PIM PEs in different single-host logical sub-systems we have: $L_{C2C\text{-}pim\text{-}remote\text{-}system}(\sigma_1) = 80\tau$, with $d = 16$ and $T_{hop} = 2\tau$.

- *host-to-PIM cores remote system C2C block transfer*

  For a C2C communication among a host PE and a PIM PE in different single-host logical sub-systems we have: $L_{C2C\text{-}pim\text{-}host\text{-}remote\text{-}system}(\sigma_1) = 104\tau$, with a total path distance $d = 22$ and $T_{hop} = 2\tau$.

- *host-to-host cores remote C2C block transfer*

  For a C2C communication among two host cores in different host processors we have: $L_{C2C\text{-}host\text{-}remote}(\sigma_1) = 120\tau$, with a total path distance $d = 26$ and $T_{hop} = 2\tau$.

Summarizing, it is interesting to stress the fact that the host-to-host remote C2C communications, that involve two host cores in different host processors, are the costliest ones.

Chapter 5
Energy Modelling of PIM Architectures

In this section an energy modelling of PIM architectures presented in previous chapters will be derived. In so doing, we will concentrate on energy costs associated to data movement to/from memories and across interconnection networks since, as widely detailed, they take on a first-class importance in current computational trends and the minimization of them is crucial for the achievement of current and future computational goals. Structured parallel computing theory will be then widely exploited in order to ease the task of deriving data movement-based *energy cost models* for parallel programs targeting PIM architectures.

5.1 Estimating energy efficiency of a computational system

Measuring or estimating power/energy consumption of a computational system is one of the most important issues in large-scale computing infrastructure for enhancing energy efficiency and constructing an energy management policy. For this reason, different power/energy models have been and are now extensively studied. Anyway, deriving an energy model of a parallel program executed over a computational system is not a simple task; different parameters and factors are involved and heterogeneous aspects should be considered. These aspects are related not only to the usage and to the energy efficiency of particular physical components that make up the system, e.g. GPUs or energy-efficient processors, but also, and mainly, to the parallel program characteristics and how the program efficiently exploits the target architecture in terms of communications and "pure" computation data movements. As a matter of fact, the given parallel program could be characterized by: an irregular access pattern that does not allow to exploit caches effectively, a large amount of data transfer deriving from communications, additional data movements related to the inter-process cooperation run-time support, e.g. cache coherence traffic associated to shared data modifications, etc.

Different works trying to derive an energy modelling of a computational system exist and most of them focus mainly on multi/many-core processors; as an example, in [37] an high-level characterization based on the Energy-Per-Instruction (EPI) of the *Xeon Phi* processor is detailed, whereas in [36], a simple power model for a multi-core server system is proposed by taking into account only four parameters: operating frequency, number of active cores, number of cache accesses and number of last-level cache misses (accounting for main memory power consumption associated to cache blocks transfers).

As estimated in [32], the memory subsystem (memory chip, interfaces and links) consumes approximately 35% of the total system power budget and it is anticipated to consume more than 60% in future Exascale systems. Although 3D-stacked memories are able to provide less energy consumption per bit compared to current DDR4 DRAMs, off-chip memory accesses still cause high energy overhead [29]. For this reason, a data movement quantitative estimate should be taken into account when expressing algorithmic or energy complexity of a large-scale parallel computation, besides designing parallel applications that communicate as little as possible [15].

In the following sections a data movement-based analytical energy model will be derived for PIM architectures presented in previous chapters, i.e. single-host and multi-host PIM systems, and it will be used in relationship with different structured parallel program's cooperation mechanisms, such as: collective communications (e.g. scatter, multicast) and/or collective operations (e.g. reduce), and related mappings (i.e. over PIM or host PEs or both). Obviously, it is not an exact method that is able to precisely quantify the amount of data transferred, nor the exact number of communications that take place and the related energy consumption; anyway, it is able to provide a general idea of how good, or bad, is the energy efficiency of a given structured parallel program mapping, in relation to another one, by taking into account only (an order of magnitude of) the cache blocks transferred among system components.

5.2 Basic system components energy costs and parameters definition

Without loss of generality and as already specified in previous chapters, we will assume that the 3D-stacked memories of PIM architectures studied are HMC-like memories; therefore, memory networks are realized through HMCs chaining and point-to-point connections are realized by means of SerDes links, both for Host-to-3DMU and 3DMU-to-3DMU connections. As said before, we are interested in evaluating energy consumption of L1-cache block transfers, each one of size $\sigma_1 = 32$ Bytes.

Thus, the following energy consumption costs have been retrieved from the supporting literature, in particular [6, 35], and are below expressed in terms of energy per primary level cache block transfer (Joule-per-σ_1):

- *SerDes (Serializer/Deserializer) link energy per C1-block*

  Energy consumed to send a primary cache block over a SerDes link (used to realize the main point-to-point interconnections):

  $$E_{link}(\sigma_1) \sim 0.26 \text{ nJ}/\sigma_1$$

- *HMC's SerDes physical interfaces energy per C1-block*

  Energy consumed by SerDes physical interfaces in the logic-layer of a HMC:

  $$E_{3DMU\text{-}INF}(\sigma_1) \sim 1.28 \text{ nJ}/\sigma_1$$

- *HMC logic-layer components energy per C1-block*

  Energy consumed by the internal logic-layer logic except physical interfaces previously accounted for:

  $$E_{logic}(\sigma_1) \sim 0.46 \text{ nJ}/\sigma_1$$

- *3D-stacked memory layers energy per C1-block*

  Energy consumed by the DRAM layers logic of the 3D-stacked memory:

  $$E_{memory\text{-}layers}(\sigma_1) \sim 0.95 \text{ nJ}/\sigma_1$$

Therefore, for the whole HMC logic-layer we can derive the following energy cost:

$$E_{logic\text{-}layer}(\sigma_1) \sim E_{logic}(\sigma_1) + E_{3DMU\text{-}INF}(\sigma_1) = 1.74 \text{ nJ}/\sigma_1$$

Moreover, it could be convenient for subsequent energy modelling to define the energy consumed for a single “hop” step when data are transferred across 3DMUs networks. The single “hop” step consists of the subsystem pair (*3DMU logic-layer*, *output link*), thus we can define:

$$E_{hop}(\sigma_1) \sim E_{logic\text{-}layer}(\sigma_1) + E_{link}(\sigma_1) = 2 \text{ nJ}/\sigma_1$$

as the ***hop energy*** paid for switching a primary cache block within a 3DMUs network.

In addition to the previous parameters, the following ones will be also useful when deriving energy consumption modelling of specific parallel program mappings, collective communications and/or operations. Let n_w, or simply n, the number of worker modules (i.e. functional modules) of a given parallel program and n_{ps} the number of service modules (e.g. scatter, gather, multicast, etc.), if any, such that $n_\Sigma = n_w + n_{ps}$ is the total number of parallel program's processing modules, then it is possible to define:

- $n_{pim} = n$ as the number of PIM PEs used to allocate the whole set of worker modules, if a PIM cores mapping of functional processing modules is chosen, such that $n_{pim} \leq N_{pim}$ (where N_{pim} is the total number of PIM cores considering the whole architecture);

- $n_{host} = n$ as the number of host PEs exploited if, instead, a host cores mapping is preferred, such that $n_{host} \leq N_{host}$ (where N_{host} is the total number of host cores considering the whole architecture).

Service modules mapping is variable, i.e. it is possible to have a configuration in which a scatter is mapped over a host core while workers are mapped over PIM cores, and therefore it will be treated case-by-case; what has to be remarked is the fact that the following constraint must always be respected in order to guarantee an *exclusive-mapping* (one process per processor) and to avoid a performance degrading multiprogrammed execution: $n_\Sigma \leq N_{pim} + N_{host}$.
We define also p_{pim}, such that $p_{pim} \leq P_{pim}$, as the number of PIM processors exploited in the parallel computation, i.e. for which at least one PIM core is used to host a process. Moreover, from Chapter 3, we have: $P_{pim\text{-}PE}$ defined as the total number of PEs/cores of a single PIM processor, such that $P_{pim\text{-}PE} = N_{pim} / P_{pim}$, whereas P_{pim} is the total number of PIM processors. In the same way, we define p_{host}, such that $p_{host} \leq P_{host}$, as the number of host processors exploited in the parallel computation and, again, $P_{host\text{-}PE}$ is the total number of PEs/cores of a single host processor, such that $P_{host\text{-}PE} = N_{host} / P_{host}$, and P_{host} is the total number of host processors.

5.2.1 Assumptions and approximations

As previously anticipated, estimating energy consumption of a parallel application mapped over a PIM architecture, or a computational system in general, is a complex task because of the large number of architectural and application-related variants that are involved. As a matter of fact, for a sequential application it is possible to derive well-approximated data movement-based energy costs by simply taking into account the number of last-level cache (LLC) faults, indicated as $N_{fault\text{-}LLC}$, for which a cache blocks transfer from the main memory is required [29]. On the contrary, a parallel application involves not only the number of LLC faults of every processing modules, but also data movement associated to inter-process communications; furthermore, according to the specific implementation of the inter-process cooperation run-time support mechanisms, notably processor synchronization and cache coherence, a variable amount of data transfer could be required too.

What discussed so far implies an approximate energy estimation approach based on some assumptions, the most meaningful of which are:

- the energy cost models derived in the following sections will estimate data transfer energy consumption by taking into account only the number of cache blocks transferred after a LLC fault and during an inter-process communication. Therefore, data movement associated to all the inter-process cooperation run-time support actions except the message copy, e.g. sender-receiver synchronizations, low-level scheduling, etc., will be neglected. Although this introduces an error in the estimates, the I/O-based RDY/ACK solution and the *home-flush* cache-coherence technique, chosen for our system models, are able to minimize data movements associated to inter-process communications run-time support and, therefore, the error of energy consumption estimates;

- as previously anticipated, we avoid to treat the case of a mixed workers set mapping, i.e. exploiting both PIM and host cores for functional modules allocation, in that it is not meaningful from the performance point of view, at least when a structured parallel program with functional replication is executed. As a matter of fact, a PIM architecture provides such a great flexibility that it could be effectively exploited according to the characteristics of the parallel application: if it is compute-intensive and/or cache-friendly then an host core mapping can be exploited; conversely, if it is memory-intensive then a PIM cores mapping could be preferred. A similar reasoning can be performed by taking into account the energy consumption associated to data transfer. Therefore, when a structured parallel program with functional replication is executed, every worker has the same computational limitations of the others and, as such, a PIM or host cores mapping should be chosen. Conversely, if functional partitioning/decomposition is exploited and a given functional module is memory-intensive or, at the same way, if a given application contains a memory-intensive part that has to be *accelerated*, then a mixed mapping can be exploited as well. Anyway, in the following we will treat only structured parallel program characterized by functional replication which feature, among other things, a greater or equal bandwidth (with respect to functional partitioning) [1];

- when we want to estimate the $N_{fault\text{-}LLC}$ parameter, then we will consider only the number of faults for which a non-null probability of cache blocks transfer exists. As a matter of fact, if a data-structure is used by the program in a “write-only” mode and if the cache unit is designed with the *write-only* optimization then, in case of a cache fault verification, the block is directly allocated and written in cache without any transfer from the main memory.

- Data transfers associated to cache writing operations management (e.g. *Write-Back* or *Write-Through*) and cache replacement algorithms (e.g. LRU) are not considered in that they are architecture-dependent and not predictable respectively.

5.3 Energy Modelling of a single-host PIM architecture

Let us consider an abstract machine model for a single-host PIM architecture, as the one proposed in Chapter 3 and reported in the following figure, such that the relevant parameters that will be considered in the following energy cost models are the average distance of the Host-to-3DMUs network d_{net} and the average distance of the 3DMU-to-3DMU interconnection $d_{mem\text{-}net}$.

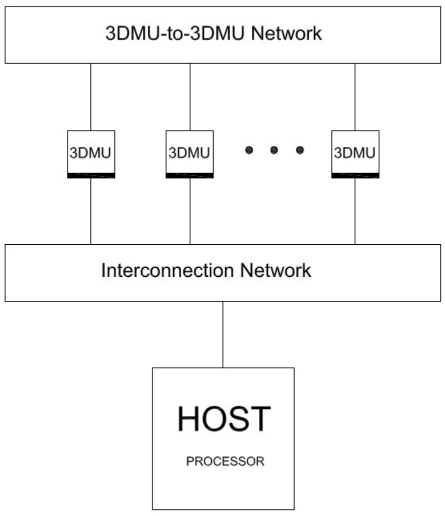

Figure 3.1: The general Abstract Machine Model for a single-host PIM architecture.

Therefore, it is possible to define the following energy costs per primary cache block transfer:

- *PIM intra-stack memory access*

  In this case we evaluate energy consumption cost for an internal cache block transfer between the memory stack and an underlying PIM core cache; therefore, only constant costs related to the internal logic-layer's logic and memory layers have to be considered:

  $$E_{pim\text{-}mem\text{-}acc}(\sigma_1) \sim E_{logic}(\sigma_1) + E_{memory\text{-}layers}(\sigma_1) = 1.41 \text{ nJ}/\sigma_1$$

- *Host memory access*

  Here we should take into account the hop energy costs of the Host-to-3DMU interconnection traversal and the final off-chip memory access energy cost:

  $$E_{host\text{-}mem\text{-}acc}(\sigma_1) \sim E_{hop}(\sigma_1)\, d_{net} + E_{memory\text{-}layers}(\sigma_1)$$

  In the case of a direct connection, as the one we had in the single-host PIM architecture of Chapter 4, then $d_{net} = 1$ and $E_{host\text{-}mem\text{-}acc}(\sigma_1) \sim 2.95 \text{ nJ}/\sigma_1$. Therefore it is interesting to note that, as confirmed by [34], the energy consumption paid for an external memory access is 2x more than an internal one. Nevertheless, it should be noted that it is a best-case result in that $E_{host\text{-}mem\text{-}acc}(\sigma_1)$ depends on d_{net} (that is equal to the minimum, i.e. 1, in the above estimate) while $E_{pim\text{-}mem\text{-}acc}(\sigma_1)$ remains constant.

- *PIM-to-PIM external C2C communication*

  In this case we evaluate energy consumption cost for an external C2C block transfer between PIM cores in different PIM processors. Therefore, we have:

  $$E_{pim\text{-}to\text{-}pim}(\sigma_1) \sim E_{hop}(\sigma_1)\ (d_{mem\text{-}net} - 1 + \delta_{unitary\text{-}dist}) + E_{logic\text{-}layer}(\sigma_1)$$

  where $(d_{mem\text{-}net} - 1 + \delta_{unitary\text{-}dist})$ is the average traversed distance across the memory network, considering the remote PIM core's 3DMU as the destination node "embedded" in the network itself and for which only $E_{logic\text{-}layer}(\sigma_1)$ has to be accounted for. The parameter $\delta_{unitary\text{-}dist}$ is a binary indicator variable such that it is 1 if $d_{mem\text{-}net} = 1$ and 0 otherwise; in this way the multiplication by 0 is avoided in case of a unitary average distance.

- *Host-to-PIM external C2C communication*

  In this case we evaluate energy consumption cost for an external C2C block transfer between a PIM core and a host core. Thus, we have simply:

  $$E_{host\text{-}to\text{-}pim}(\sigma_1) \sim E_{hop}(\sigma_1)\ d_{net}$$

  that is equal to the previous $E_{host\text{-}mem\text{-}acc}(\sigma_1)$ without considering the final memory access energy cost.

Finally, let us denote by E_1, E_2, E_3 and E_4 the energy costs per primary cache block listed above, and by Σ_1, Σ_2, Σ_3 and Σ_4 the total number of cache block transferred for every case, then the total amount of energy consumed by a parallel program mapped over a single-host PIM architecture, considering data movement costs *only*, is given by:

$$E_{data\text{-}mov} = \sum_{j=1}^{4} E_j\, \Sigma_j \qquad (1)$$

The above formula will be extensively used in the final examples and in the next chapter in order to derive energy costs of parallel programs variants and related mechanisms.

Introduction and preconditions to the examples

It is worth noting that, for everyone of the following examples, a performance modelling will be included as well in order to enhance the subsequent analysis in terms of energy-performance trade-off. In particular, we will be interested in the following metrics for performance evaluation of stream-based parallel applications: the *mean service time T_s* of a given processing module or a parallel computation as a whole, defined as the *average time interval between the beginning of the executions over two consecutive input stream items*, or, equivalently, its inverse $B = 1/T_s$ called *processing bandwidth* or *throughput*, and the *latency L* defined as the *mean time needed by a processing module or a parallel computation to process a single input stream item* (more details about performance cost models of structured parallel programs can be found in [1]).

As said before, we will assume to work with stream-based structured parallel computations, such that every processing module is typically characterized by the recurrent loop phases: *receive – compute – send*, and C2C communications will be largely exploited in order to involve main memory only when strictly needed. For this reason, we will assume that, in a typical producer to consumer interaction, the exchanged message, of length L bytes, is able to fit in primary cache when at least a PIM PE is involved in the communication (otherwise in secondary cache if only host PEs are involved). In this way, an amount of at most $3L/\sigma_1$ cache blocks transfer is avoided ($2L/\sigma_1$ are paid by the producer module to read the message from "its" memory and write/send the message to the "consumer" memory, and L/σ_1 are paid by the consumer to read the received message). Therefore, in the following single-host case examples only a PIM cores mapping of functional modules will be assumed, when collective cooperation are studied, owing to the previous assumption; as a matter of fact, the host cores mapping is not meaningful since no energy is consumed for communications thanks to the on-chip C2C exploitation. Obviously, this is not more valid when more than one host processor is involved in the computation (i.e. multi-host case).

Finally, the general architecture specifications and related parameters can be found in Chapters 3 and 4. The same is valid for multi-host PIM architecture examples at the end of this chapter.

5.3.1 Example 1: scatter collective communication

Let us assume to have a data structure A of size M Bytes that has to be scattered to a set of n processing modules, i.e. split into partitions of $g = M/n$ bytes, or $M/n\sigma_1$ blocks, and sent to the destination modules, then two main solutions can be studied: a centralized solution and a tree-structured solution.

Centralized Scatter

In this case, the implementation consists of n sequential point-to-point communications, each one directed to a different target module, executed by a centralized and distinct process. As previously assumed, a PIM cores mapping with $n_{pim} = n$ is exploited; therefore, the following two cases can be distinguished according to the scatter module mapping.

PIM mapping

Assuming that the data structure has to be read from the local memory stack by the scatter module and that M/σ_1 is the total number of blocks required to transfer A, then the total energy consumption cost, using (1), is:

$$E_{scatter\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1)\, M/\sigma_1 + E_{pim\text{-}to\text{-}pim}(\sigma_1)\, c_{ext}\, M/n\sigma_1$$

where the c_{ext} (which stands for *ext*ernal *c*ommunications) parameter is defined as:

$$c_{ext} = \begin{cases} 0 & if\; n \leq P_{pim-PE} - 1 \\ n - (P_{pim-PE} - 1) & if\; n \geq P_{pim-PE} \end{cases}$$

and indicates the number of data structure *partitions* that have to be sent *externally* from the 3DMU stack of the PIM processor into which the scatter module is mapped.

Notice that in the above formula it is assumed that the functional modules PIM cores mapping seeks to fill first the PIM processor in which the scatter module is mapped, thus supporting internal C2C communications with energy and performance benefits (i.e. the case $n \leq P_{pim\text{-}PE} - 1$). Instead, if the number of workers does not fit into a single PIM processor with a scatter module already allocated (i.e. the case expressed by the $n \geq P_{pim\text{-}PE}$ condition), then cores of other/s PIM processor/s have to be involved.

Host mapping

In this case the data structure has to be read from a given 3DMU and then scattered to n distinct modules mapped over PIM cores. Therefore:

$$E_{scatter\text{-}host} \sim E_{host\text{-}mem\text{-}acc}(\sigma_1)\, M/\sigma_1 + E_{host\text{-}to\text{-}pim}(\sigma_1)\, M/\sigma_1$$

Comparison

With reference to the single-host architecture specifications of Chapter 4, for which $d_{net} = 1$, $d_{mem\text{-}net} = 1$, $N_{pim} = 64$, $P_{pim} = 4$ and $P_{pim\text{-}PE} = 16$, and assuming that $n = n_{pim} = 63$, such that

p_{pim} = 4 processors are involved in the computation, then the energy consumption costs are: $E_{scatter\text{-}pim} \sim 4.26\ M/\sigma_1$ nJ and $E_{scatter\text{-}host} \sim 4.95\ M/\sigma_1$ nJ. Therefore, in the worst case, the two energy costs are comparable.

If instead, for $n = n_{pim} = P_{pim\text{-}PE} - 1$, only one PIM processor is involved (i.e. $p_{pim} = 1$) and internal C2C communications can be exploited at all, then $E_{scatter\text{-}pim} \sim 1.41\ M/\sigma_1$ nJ while $E_{scatter\text{-}host}$ does not change.

In conclusion, when a PIM PE is used to allocate the scatter module of a parallel computation, whose functional modules are mapped over PIM cores only, then, in the best case, an energy efficiency of 3.5x is obtained with respect to a host PE mapping.

Performance

When a centralized sequential solution is exploited, then service time and latency of the scatter module coincide and are linear in n:

$$T_{scatter} = L_{scatter} = n\ T_{send}(g) = n\ T_{setup} + M\ T_{transm}$$

Tree-structured Scatter

If a tree-structured scheme is exploited, then the data structure scattering is directly performed in parallel by the n functional modules. Anyway, apart from performance, the energy costs are the same since still $c_{ext}\ M/n\sigma_1$ blocks have to be sent involving inter-stack off-chip communications.

Performance

When a tree-structured parallel scheme is exploited, then service time has a modest improvement and is equal to the tree-root service time:

$$T_{scatter} = L_{scatter} = 2\ T_{send}(M/2) = 2\ T_{setup} + M\ T_{transm}$$

On the contrary, latency is sensibly improved and it is now logarithmic in n.

5.3.2 Example 2: multicast collective communication

As in the previous case, let us assume to have a data structure A of size M Bytes that has to be sent to a set of n processing modules with a multicast communication, then two main solutions can be studied: a centralized solution and a tree-structured solution.

Centralized Multicast

In this case, the implementation consists of *n* sequential point-to-point communications executed by a centralized and distinct module. The following two cases can be distinguished according to the multicast module mapping.

PIM mapping

Again, assuming that the data structure has to be read from the local memory stack by the multicast module, and that M/σ_1 is the total number of blocks required to transfer a copy of *A* to everyone of the *n* target modules, then the total energy consumption cost is:

$$E_{multicast\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1)\ M/\sigma_1 + E_{pim\text{-}to\text{-}pim}(\sigma_1)\ c_{ext}\ M/\sigma_1$$

where c_{ext} counts the number of data structure *copies* that have to be sent *externally* from the 3DMU to which the PIM processor hosting the multicast module belongs. The considerations made for the centralized scatter solution about PIM workers mapping policy can be applied in this case too.

Host mapping

In this case, the data structure has to be read from a given 3DMU and then sent in multicast to *n* distinct modules mapped over PIM cores. Therefore:

$$E_{multicast\text{-}host} \sim E_{host\text{-}mem\text{-}acc}(\sigma_1)\ M/\sigma_1 + E_{host\text{-}to\text{-}pim}(\sigma_1)\ n\ M/\sigma_1$$

Performance

When a centralized sequential solution is exploited, then service time and latency of the multicast module coincide and are proportional to *n*:

$$T_{multicast} = L_{multicast} = n\ T_{send}(M) = n\ (T_{setup} + M\ T_{transm})$$

Tree-structured Multicast

If a tree-structured scheme is exploited, with a worker-mapped implementation (i.e. every worker is a node of the logical multicast tree), then different multicast communication patterns can be recognized according to the chosen tree visit strategy (e.g. *depth-first*). Anyway, when the workers set is mapped over PIM cores, then in the best case at least one external communication per PIM processor involved in the computation (in total p_{pim} - 1) is needed. Therefore, the total energy costs, considering also the data structure reading from main memory by the root worker, can be evaluated as:

$$E_{tree\text{-}multicast\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1)\ M/\sigma_1 + E_{pim\text{-}to\text{-}pim}(\sigma_1)\ (p_{pim} - 1)\ M/\sigma_1$$

Performance

As for the scatter case, when a tree-structured parallel solution is exploited, then service time is equal to:

$$T_{multicast} = 2\ T_{send}(M) = 2\ (T_{setup} + M\ T_{transm})$$

Again, latency is sharply improved and it is now logarithmic in n.

Comparison

Let us refer again to the single-host architecture specifications of Chapter 4 (reported also in the previous scatter case comparison), and assuming $n = 63$ as in the scatter case, then $E_{multicast\text{-}pim} \sim 180.93\ M/\sigma_1$ nJ and $E_{multicast\text{-}host} \sim 128.95\ M/\sigma_1$ nJ. Thus, in the worst case, the host PE multicast module mapping is about 1.4x more efficient than the PIM core mapping. With the same specifications, the tree-structured multicast is able to efficiently consume $E_{tree\text{-}multicast\text{-}pim} \sim 12.63\ M/\sigma_1$ nJ, which is a notable improvement.

Conversely, if only one PIM processor is involved and internal C2C can be exploited at all, e.g. for $n = P_{pim\text{-}PE} - 1 = 15$, then $E_{multicast\text{-}pim} = E_{tree\text{-}multicast\text{-}pim} \sim 1.41\ M/\sigma_1$ nJ and $E_{multicast\text{-}host} \sim 32.95\ M/\sigma_1$ nJ, with PIM PE multicast module mapping and tree-structured solution which are 23.4x more efficient than the host PE mapping.

5.3.3 Example 3: reduce collective operation

Given a computation that performs the sum of the vector columns of a given integers matrix *A[R][M]* and puts the result in a vector *B[M]*, i.e. :

$$B = \sum_{j=0}^{M-1} A^j$$

then, it can be equivalently re-written as:

$$B = reduce\ (A^j, +) \quad \text{with}\ j = 0\ ...\ M-1$$

where A^j is the j-th matrix column, + is the associative operator involved (i.e. a sum), and *reduce* is the well-known summary operation.

Let us assume that the given matrix *A* has been previously scattered by blocks of columns to the *n* workers and that, after a local reduce over the received partition, each worker takes part to the global reduce by communicating its partial results vector of size R; then three main solutions can be studied: a centralized solution, a tree-structured solution and a tree + centralized solution.

Centralized Reduce

In this case, the implementation consists of n point-to-point communications from every worker to a centralized and distinct module that performs the global reduce locally over the n received partial results vectors. Again, a PIM cores mapping is assumed with $n_{pim} = n$. Two cases can thus be distinguished according to the centralized module mapping.

PIM mapping

When a PIM PE mapping is exploited, then the total energy consumption cost is:

$$E_{centralized\text{-}reduce\text{-}pim} \sim E_{pim\text{-}to\text{-}pim}(\sigma_1)\ c_{ext}\ R/\sigma_1$$

assuming, again, that the centralized module is always allocated in a PIM processor that is used first when the workers set mapping has to be performed. Moreover, this time the parameter c_{ext} indicates the number of partial results vectors that are received from external PIM processors (with respect to the one in which the centralized module is allocated).

Host mapping

In this case we have simply:

$$E_{centralized\text{-}reduce\text{-}host} \sim E_{host\text{-}to\text{-}pim}(\sigma_1)\ n\ R/\sigma_1$$

in that all the n workers mapped over PIM cores send their partial results vectors to the centralized module.

Performance

When a centralized solution is exploited then the global reduce latency is equal to:

$$T_{reduce} = (T_G + T_{send})\ n$$

where T_G is the calculation time of the associative operator/function adopted.

The above time could be potentially masked in stream-based parallel computation, by exploiting the so called *pipeline-effect*, provided that the n proportionality and $T_G + T_{send}$ time allow it.

Tree-structured Reduce

In this case, the global reduce is performed in parallel, following a tree-structured scheme, by all the workers. Since they are mapped over PIM cores, then some communications are performed intra-stack, i.e. considering the tree levels close to the leaves, while other involve inter-stack communications, i.e. moving towards the tree root. Thus, if we consider all the p_{pim} PIM processors involved in the global reduce as leaves of another logical binary tree, then it is easy to convince ourself that the total number of external communications is equal to the number of internal nodes in the tree, i.e. (p_{pim} - 1). Therefore, the energy cost can be evaluated as:

$$E_{tree\text{-}reduce\text{-}pim} \sim E_{pim\text{-}to\text{-}pim}(\sigma_1)\ (p_{pim} - 1)\ R/\sigma_1$$

Performance

When a tree-structured solution is exploited then the global reduce latency is equal to:

$$T_{reduce} = (T_G + T_{send})\ log_2\, n$$

Tree-structured + Centralized Reduce

A notable reduce variant discussed in the supporting literature, in particular in [21], and that seems to take advantages from PIM architectures is the tree-structured + centralized reduce, also indicated as *in-memory reduction trees*.

In this solution there is not a single but different reduction trees, as far as the number of PIM processors is concerned, such that workers belonging to everyone of them work in parallel exploiting C2C intra-stack communications only. At the end, all the final results, one for every tree, are collected by a centralized module mapped over a host core such that the final reduce can be performed.

Therefore, the energy cost associated to this solution is equal to:

$$E_{tree+centralized} \sim E_{host\text{-}to\text{-}pim}(\sigma_1)\ p_{pim}\ R/\sigma_1$$

Performance

When a tree-structured + centralized solution is exploited then the global reduce latency is equal to:

$$T_{reduce} = (T_G + T_{send})\ log_2\,(n/p_{pim}) + (T_G + T_{send})\ p_{pim}$$

As for the centralized solution, the time related to the centralized reduce phase could be potentially masked in a stream-based parallel computation.

Even more so, this time it depends on p_{pim} and not to n; we can confirm that, in general, the condition $p_{pim} << n$ is always true in large-scale applications and architectures.

Comparison

Let us refer again to the single-host architecture specifications of Chapter 4 (reported also in previous comparisons), and assuming $n = 63$, then the following results can be achieved: $E_{centralized\text{-}reduce\text{-}pim} \sim 179.52\ M/\sigma_1$ nJ and $E_{centralized\text{-}reduce\text{-}host} \sim 126\ M/\sigma_1$ nJ.

The tree-structured solution achieves an energy cost equal to $E_{tree\text{-}reduce\text{-}pim} \sim 11.22\ M/\sigma_1$ nJ, while the tree-structured + centralized solution achieves $E_{tree+centralized} \sim 8\ M/\sigma_1$ nJ. The latter is 1.4x better than the former solution and sharply better than the centralized solutions.

In conclusion, the better energy-performance trade-off can be achieved with the tree + centralized reduce solution; even more, if the centralized phase latency can be masked and, furthermore, $p_{pim} > 2$ and $d_{net} \leq d_{mem\text{-}net}$ (most common cases), then it is the best solution taking into account both energy and performance improvements.

5.4 Energy Modelling of a multi-host PIM architecture

Using the same approach of the single-host case, let us consider an abstract machine model for a multi-host PIM architecture, as the one proposed in Chapter 3 and reported in the following figure, such that relevant parameters that will be considered in the following are the average distance of the Host-to-3DMUs network d_{net} and the average distances of the 3DMU-to-3DMU local interconnection $d_{mem\text{-}net\text{-}local}$ and global interconnection $d_{mem\text{-}net\text{-}global}$.

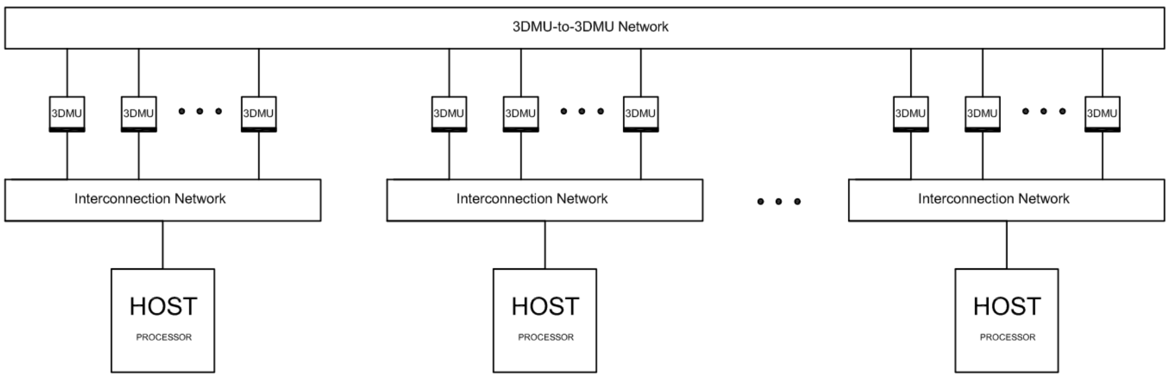

Figure 3.2: The general Abstract Machine Model for a multi-host PIM architecture.

In the multi-host case, different energy costs per primary cache block transfer can be

inherited from the previous single-host energy costs study; as a matter of fact, the only difference resides in the fact that now a local and a global memory network average distance has to be considered. As a consequence, every energy cost model that in the single-host case depends on $d_{mem\text{-}net}$ now has to be studied for both $d_{mem\text{-}net\text{-}local}$ (i.e. $d_{mem\text{-}net}$) and $d_{mem\text{-}net\text{-}global}$. At the same way, for every kind of communication between system components that involves the global memory network a new energy cost model has to be derived as well.
Therefore, we can proceed as follows:

- *PIM intra-stack memory access*

  The same of the single-host case, i.e. :

  $$E_{pim\text{-}mem\text{-}acc}(\sigma_1) \sim E_{logic}(\sigma_1) + E_{memory\text{-}layers}(\sigma_1) = 1.41 \text{ nJ}/\sigma_1$$

- *Host memory access* (*local and remote*)

  When the memory access is *local*, then we have the same cost of the single-host case memory access, i.e. :

  $$E_{host\text{-}mem\text{-}acc\text{-}local}(\sigma_1) \sim E_{host\text{-}mem\text{-}acc}(\sigma_1)$$

  in that the global 3DMU-to-3DMU network is not involved.
  On the contrary, in the case of a *remote* memory access then the energy cost per cache block is:

  $$E_{host\text{-}mem\text{-}acc\text{-}remote}(\sigma_1) \sim E_{hop}(\sigma_1)\ (d_{net} + d_{mem\text{-}net\text{-}global} - 1) + E_{memory\text{-}layers}(\sigma_1)$$

  where d_{net} is the average number of hops that has to be performed in order to reach the global memory network while, instead, $d_{mem\text{-}net\text{-}global} - 1$ is the average number of steps to reach the destination 3DMU through the global memory network. Notice that a single unit step has been subtracted since the source 3DMU in the global memory network, which is also the destination 3DMU in the Host-to-3DMUs interconnection, has to be considered only one time.

- *PIM-to-PIM external C2C communication* (*local and remote*)

  When the external C2C block transfer is *local* (i.e. same logical single-host subsystem), then we have the same cost of the single-host C2C block transfer between PIM PEs in different PIM processors, i.e. :

$$E_{pim\text{-}to\text{-}pim\text{-}local}(\sigma_1) \sim E_{pim\text{-}to\text{-}pim}(\sigma_1)$$

On the contrary, in the case of a *remote* C2C block transfer the energy cost can be evaluated as:

$$E_{pim\text{-}to\text{-}pim\text{-}remote}(\sigma_1) \sim E_{hop}(\sigma_1)\ (d_{mem\text{-}net\text{-}global} - 1 + \delta_{unitary\text{-}dist}) + E_{logic\text{-}layer}(\sigma_1)$$

where $(d_{mem\text{-}net\text{-}global} - 1 + \delta_{unitary\text{-}dist})$ is the average traversed distance across the global memory network and, again, $\delta_{unitary\text{-}dist}$ is a binary indicator variable such that it is 1 if $d_{mem\text{-}net\text{-}global} = 1$ and 0 otherwise. It is interesting to note that the above formula is correct if the source PIM processor belongs to a 3DMU that, in turn, belongs to the global 3DMU-to-3DMU interconnection network. On the contrary, if the source 3DMU has to reach the global memory network by traversing first the local one, then the term $E_{hop}(\sigma_1)\ (d_{mem\text{-}net\text{-}local} - 1 + \delta_{unitary\text{-}dist})$ has to be added in the above formula too. The same reasoning is valid for the destination 3DMU.

- *Host-to-PIM external C2C communication* (*local and remote*)

  Again, when the external C2C block transfer is *local* then:

  $$E_{host\text{-}to\text{-}pim\text{-}local}(\sigma_1) \sim E_{host\text{-}to\text{-}pim}(\sigma_1)$$

  On the contrary, if the external C2C block transfer, between a PIM and a host PE, is *remote* then:

  $$E_{host\text{-}to\text{-}pim\text{-}remote}(\sigma_1) \sim E_{hop}(\sigma_1)\ (d_{net} + d_{mem\text{-}net\text{-}global} - 1)$$

  which is equal to the previous $E_{host\text{-}mem\text{-}acc\text{-}remote}(\sigma_1)$ without considering the final memory access energy cost.

- *Host-to-Host external C2C communication*

  Obviously, this kind of interaction occurs only in a multi-host PIM architecture and, for this reason, a new cost model has to be derived. Anyway, it is very similar to the previous one except that now also the destination 3DMU in the global memory network, which is also the source 3DMU in the Host-to-3DMUs interconnection, has to be considered only one time.

  Therefore, the energy cost model for a Host-to-Host C2C communication is:

  $$E_{host\text{-}to\text{-}host}(\sigma_1) \sim E_{hop}(\sigma_1)\ (2d_{net} + d_{mem\text{-}net\text{-}global} - 2)$$

  where $2d_{net}$ is due to the fact that this time two Host-to-3DMUs networks are involved. Moreover, the condition $d_{mem\text{-}net\text{-}global} \geq 2$, which is always true for a large network, has to be satisfied in order to avoid wrong results.

Again, let us denote by E_1, E_2, E_3, E_4 and E_5 the energy costs per primary cache block listed above, and with Σ_1, Σ_2, Σ_3, Σ_4 and Σ_5 the total number of cache block transferred for every case. Remembering the fact that for E_2, E_3, E_4 we must take into account both local and remote energy communication costs, then the total amount of energy consumed by a parallel program mapped over a multi-host PIM architecture is given by:

$$E_{data\text{-}mov} = \sum_{j=1}^{5} E_j \Sigma_j \qquad (2)$$

5.4.1 Example: producer-consumer pattern

Let us assume to have a data structure A of size M Bytes that has to be sent from a producer to a consumer module. Then, according to the mapping of the two processes, three interesting variants can be studied: host mapping in different host processors, PIM mapping in different PIM processors belonging to the same and to different single-host sub-systems.

PIM mapping

Assuming that the producer process maintains the data structure in its L1-cache, then the energy costs obtained using (2) are:

- S*ame single-host subsystem* (*local case*)

  In this case, a local inter-stack communication is exploited; therefore:

  $$E_{prod\text{-}cons\text{-}pim\text{-}local} \sim E_{pim\text{-}to\text{-}pim\text{-}local}(\sigma_1)\ M/\sigma_1$$

- *Different single-host subsystems* (*remote case*)

  This time, a remote inter-stack communication is performed; thus:

  $$E_{prod\text{-}cons\text{-}pim\text{-}remote} \sim E_{pim\text{-}to\text{-}pim\text{-}remote}(\sigma_1)\ M/\sigma_1$$

Host mapping

In this case a C2C communication is performed among two different host processors with an energy cost evaluated as:

$$E_{prod\text{-}cons\text{-}host} \sim E_{host\text{-}to\text{-}host}(\sigma_1)\, M/\sigma_1$$

Comparison

With reference to the multi-host architecture specifications of Chapter 4, for which $d_{net} = 1$, $d_{mem\text{-}net\text{-}local} = 1$, $d_{mem\text{-}net\text{-}remote} = 4$, then the energy consumption costs are:

- $E_{prod\text{-}cons\text{-}pim\text{-}local} \sim 3.74\ M/\sigma_1$ nJ
- $E_{prod\text{-}cons\text{-}pim\text{-}remote} \sim 7.74\ M/\sigma_1$ nJ
- $E_{prod\text{-}cons\text{-}pim\text{-}host} \sim 8\ M/\sigma_1$ nJ

In conclusion, the above results show how, in a multi-host environment, computation locality into the same single-host logical sub-system has to be exploited; in this way, the energy consumption per communication is halved with respect to a host-to-host and a PIM-to-PIM remote cooperation case.

5.5 Final considerations

In this chapter, a simple yet formal study of energy related concepts in relationship to PIM architectures has been performed.

Results show how communications minimization and communications locality, intra-stack or within a single-host logical subsystem, play an important role in the energy efficiency of a PIM system. Therefore, the parallel program characteristics, in terms of data access and communication patterns, and the program mapping choice are determinant in order to minimize data movements and the associated energy consumption.

In the next chapter, the presented cost models will be extensively used in order to derive the energy efficiency of real application examples mapped over PIM architectures.

Chapter 6

Comparative Study and Evaluation of PIM Architectures with Parallel Program Examples

In this chapter a comparative study of PIM systems in relationship to structured parallel program variants, mappings, communication patterns and other related aspects will be carried out. In so doing, different parallel versions of the well-known *Count-Min Sketch* algorithm, largely used in *real-time analytics* applications, will be proposed in order to enhance the final results evaluation. The *Count-Min Sketch* algorithm characteristics, mainly in relation to the irregular data access pattern, seem to make it a suitable benchmark for PIM architectures comparison. Therefore, the final goal of this chapter is neither to find the *killer application* for Processing-in-Memory nor to aim at the best parallelization of the proposed algorithm; the real objective is to analytically provide an evaluation of how good or bad a PIM system acts when executing a memory-intensive application.

In the following sections, the *Count-Min Sketch* algorithm, its property and the application context in which it applies will be briefly described. Then, a formal analysis of its parallel versions targeting PIM architectures will be performed, for both the single and multi-host variants, exploiting an analytical approach based on energy and performance cost models. Thus, we will use all the numeric results obtained in previous chapters such that, in the light of the current state of the art about PIM technologies, they can be considered correct with good approximation.

Finally, a concluding parametric study will be carried out in order to summarize the obtained results and provide an idea of the PIM architectures potential.

6.1 Brief description of real-time analytics and sketching techniques

The term "Big Data analytics" refers to the process of examining a huge amount of data in order to discover hidden data patterns, unknown data correlations, extract knowledge and to provide useful business information.

In many modern web and Big Data applications data arrives in a streaming fashion and needs to be processed on the fly. Therefore, unlike traditional off-line or batch analytics, e.g. exploiting MapReduce processing, on-line or real-time analytics should be able to detect events and trends as they happen due to the stringent latency/time constraints. To meet these requirements, real-time analytics applications store the entire or the large majority of the dataset in main memory, rather than traditional secondary storage media, so that data can be quickly accessed and queried. Moreover, this kind of applications are typically characterized by abundant parallelism and a huge amount of data streaming with low locality, making on-chip caches ineffective and, nonetheless, offering opportunity for PIM acceleration [21].

As reported in [53], it is very common in modern analytics application to answer to statistical queries about a given set of items; common queries could be related to the membership of a given item to the set (set-membership query), to the total number of different items seen so far (cardinality or size estimation query), or how much frequently a given item occurred (frequency estimation query).

When the cardinality of the given set is small, then no storage problem arises and specialized data-structures, allowing to maintain the set of items in memory for real-time updates and queries, exist. However, when it becomes large (i.e. there are many distinct items) then the set storage becomes problematic in that it is linear in n, where n is the set cardinality.

To tackle this problem, *sketch* data structures and algorithms have been developed and proposed. A sketch is an approximate data structure which represents a summary of a given dataset and is used, by a related sketch algorithm, to deliver approximated query results. Sketch algorithms are characterized by three aspects that make them attractive: constant time updates of the data, sub-linear storage space for the sketch data structure, and at worst linear querying time. Obviously, all these desirable properties can be achieved at the cost of introducing errors into the reported results.

Among the existent sketch-based techniques, the popular Count-Min Sketch algorithm will be briefly introduced in the following sub-section and then widely exploited for subsequent analysis.

6.1.1 Count-Min (CM) Sketch algorithm

The Count-Min, or CM, Sketch algorithm, proposed by G. Cormode and S. Muthukrishnan in [41], is one of the most popular sketch-based algorithm that has found wide applications from IP traffic monitoring, machine learning, distributed computing, signal processing and beyond [40].

In its common usage, the CM sketch provides an efficient and approximated solution to the *Count Tracking* problem: given a set with a large number of items and a frequency estimate associated to each one of them, when a query for item x arrives, the answer to the query is the current frequency of x. A simple example could be a popular website, e.g. Google, which wants to keep track of statistics on the search queries; more specifically, it could be interested in maintaining the *top-K* list of frequent search queries or the list of queries with a frequency higher than some predetermined threshold (the so-called *Heavy-Hitters* list).

Let us formally define now the scenario into which the CM sketch is involved, as detailed in [39, 41], with the following preliminary set-up:

- $A_t[1, n]$ is a vector of n items whose state changes with time t. Thus, its current state at a given time t' is defined as $A_{t'} = [a_1(t'), a_2(t'),\dots, a_i(t'),\dots, a_n(t')]$.
- The updates of an individual entry of A_t at time t consists of a pair (i_t, c_t) of numbers, such that:
  - $A_{t+1}[i_t] = A_t[i_t] + c_t$, with c_t value that could be strictly positive in some cases (*cash register* case), unitary (i.e. equal to 1) or also negative (*turnstile* case);
  - $A_{t+1}[i'] = A_t[i']$, if $i' \neq i_t$.
- At any time t, a query arrives and asks for computing some simple or more complex function over A_t; in the basic case a *point query*, denoted by *query*(i), asks for the approximation of $A_t[i]$.

Goal: achieve sub-linear space in n, fast update and query but with answers that need to be inevitably (ε, δ)-approximated, meaning that the error in the returned frequency estimate for a given item i is within a factor of ε with probability δ.

Data Structure: a Count-Min sketch data structure with parameters ε and δ is represented by:

- a two-dimensional array of counters $CM[d, w]$ with $d = \lceil \ln \frac{1}{\delta} \rceil$ rows, which stands for *depth*, and $w = \lceil \frac{e}{\varepsilon} \rceil$ columns, which stands for *width*;
- a set of different hash functions $h_1,\dots, h_d$: $\{1,\dots, n\} \rightarrow \{1,\dots, w\}$, chosen uniformly

at random from a *pairwise independent* family, hence: $P(h_i(x) = h_j(x)) = 1/w$, where each i-th hash function is of the form $h_i(x) = ((ax + b) \bmod p) \bmod w)$ with p prime.

The sketch data structure can accurately summarize arbitrary set of streaming items with compact and fixed memory footprint that in some cases and applications could fit in cache, i.e. order of kilobytes to some megabyte, while in others in main memory, i.e. orders of mega to some gigabyte [40].

Update and Point Query: every item i has one estimator/counter for each row, i.e. :

$$CM\,[1, h_1(i)], \ldots, CM\,[d, h_d(i)]$$

When an update (i_t, c_t) arrives, this means that item i_t has to be updated by a quantity of c_t; therefore, each counter of i is incremented by c_t. Formally:

$$update(i_t, c_t)\text{: for } j = 1,\ldots,d \text{ do } CM\,[j, h_j(i_t)] \mathrel{+}= c_t$$

The following figure graphically summarizes what discussed so far about the update procedure.

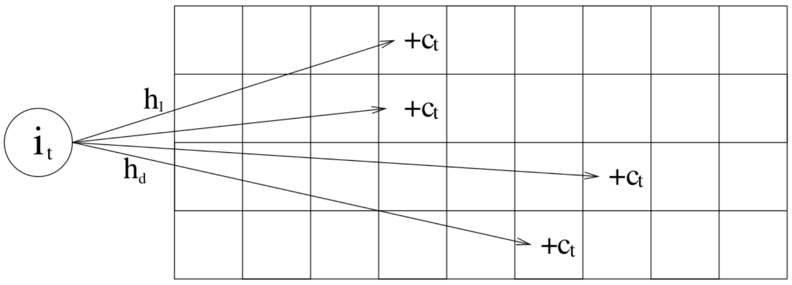

Figure 6.1: Schematic of CM sketch update procedure (*taken from* [40]).

When, instead, a point query for item i arrives, then an estimate $\hat{A}[i]$ equal to the minimum among all the counters values of i is returned. Formally:

$$query(i)\text{: return } \hat{A}[i] = \min\,\{CM\,[j, h_j(i_t)], \text{ for } j = 1,\ldots,d\}$$

It should be noted that the returned estimate is different with respect to the real frequency value of item i, i.e. $A[i]$, as stated in the following theorem from [41]:

Theorem: The estimate for item i, with $i = 1,\ldots, n$, is such that:

- $\hat{A}[i] \geq A[i]$
- and, with probability of at least 1- δ, it is: $\hat{A}[i] \leq A[i] + \varepsilon \|A\|_1$

(with $\|A\|_1 = \sum_{j=1}^{n} |A[j]|$)

Time and Space Costs: queries and updates take $O(d) = O(\ln \frac{1}{\delta})$ time, whereas space occupancy is $O(dw) = O(\frac{e}{\varepsilon} \ln \frac{1}{\delta})$ and, therefore, sub-linear in n.

6.2 Analysis and comparison of CM Sketch parallel version variants over PIM

6.2.1 Introduction and preliminary considerations

In the following sections, a formal analysis of different parallel versions of the CM sketch algorithm targeting PIM architectures will be performed, by explicitly distinguishing the update and query procedures. In so doing, we will consider the following cases.

- single and multi-host PIM target architectures;
- host and PIM cores mapping of functional processing modules;
- performance and energy costs evaluation for every parallel program variant studied.

Therefore, the architectures specifications presented in Chapter 3 and 4, as well as the cache block transfer base latencies at the end of Chapter 4, and the energy and performance cost models of Chapter 5 will be extensively exploited.

Before starting with the formal analysis, the following preliminary considerations are due:

- the goal for every parallel version variant, i.e. a stream-based parallel application, is to maximize the *offered* bandwidth that the parallel application, executed over a given PIM architecture and exploiting a given mapping, is able to achieve.

  Thus, when evaluating the optimal parallelism degree n_{opt}, i.e. the optimal number of functional modules of a given parallel program, the general formula adopted will be:

  $$n_{opt} = \lceil \frac{T_S}{T_{DD}} \rceil$$

  where T_s is the mean service time of the sequential module and and T_{DD} is the mean time needed to perform the distribution of the input data according to a certain scheme.

To be thorough, if we would have been interested to the *requested* bandwidth, then:

$$n_{opt} = \lceil \frac{T_S}{T_A} \rceil$$

where T_A is the mean inter-arrival time among consecutive input stream elements.

- The strategy adopted for parallelizing the Count-Min sketch algorithm is similar to the one proposed in [38]. At initialization time, the sketch data structure, that represents the state of the computation, and the set of d hash functions are defined and replicated in every worker of the workers set.
  In the update/ingestion phase, a data distribution module (e.g. a scatter or an emitter module), or the same workers according to the communication scheme adopted, assigns disjoint parts of the input stream to each worker that, in turn, updates its local sketch. Obviously, this strategy introduces an error since sketches/states are not coherent one another; anyway, this problem will be solved at query time, as detailed in section 6.2.4.
  Once input data stream has been analysed and sketches have been constructed, they can be asked to answer point queries. Therefore, when a query for an item x arrives, it is sent in multicast to every worker that, in turn, queries its local sketch to compute its partial results. Local results are then collected from each workers, according to a distributed and/or centralized reduce scheme, and the final global estimate for item x is returned (i.e. *multicast+map+reduce* structured computation).
  It should be clear that, with the above parallelization strategy, only the ingestion phase can be speeded up since the query processing phase is characterized by work replication across participating functional modules; thus, in the best case, the parallel query procedure bandwidth will be equal to the sequential one (in compliance with what did in [38]). For this reason, the optimal parallelism degree n_{opt} will be computed in the update phase and <u>reused</u> in the querying one.

- The values chosen for the setting parameters of the sketch data structure are:
  - $d = 40$
  - $w = 2^{20}$

  with very low values of ε and δ and a CM sketch size of about 168 MB (with 4 byte integer counters). The large value for *d* has been chosen because the service time of the sequential computation is proportional to it; therefore, for a low value of *d*, the computation would have been too fine-grained and not interesting to be studied, due to the resulting low n_{opt} value.

Again, we should remark that in no way we aim at finding the best CM sketch settings in order to solve a given problem. Our goal is only to evaluate how a PIM architecture acts when executing a memory-intensive parallel program, hopefully with interesting n_{opt} values for our purposes.

- The following study has been realized by taking into account base communication latencies and, therefore, neglecting possible contentions that could happen when executing a parallel program in a parallel architecture. Examples of contentions could be related to the concurrent accesses to a certain memory macro-module, to the conflicts on the network switch units and links, to the cache coherence protocol interactions on the home nodes cache controllers and so on.

  Anyway, the chosen architectural settings, mainly in relation to the crossbar internal interconnections as well as the high bandwidth links and interface units, the used parallel program communication forms, i.e. only symmetric (point-to-point) channels, and the implementation exploited for the run-time support mechanisms, e.g. the I/O-based RDY/ACK communications solution and the *home-flush* cache coherence technique, are such that possible contentions are minimized.

6.2.2 Sequential analysis

Let us consider a sequential processing module *Q* that implements the Count-Min sketch algorithm update and query procedures. The *Q* module encapsulates a CM sketch data structure, represented by a two-dimensional array of integer counters *CM* [*D*][*W*], receives an integer type item *x* from a given input stream channel with asynchrony degree $k > 0$, and, in the update procedure case, it updates the counters of *x* with unitary increments (i.e. $c_t = +1$); in the query procedure case, it simply returns the estimate for *x*.

Therefore, the *Q* computation can be described with the following pseudo-code:

```
Q:: int C[D][W], x, m; channel_in input_stream(k);
```

```
update(x)::
    while (true) do {
        receive(input_stream, x);
        for j = 1...D - 1: CM[j][$h_j$(x)] ++;
    }

query(x)::
    while (true) do {
        receive(input_stream, x);
        m = max_int;
        for j = 0...D - 1: m = MIN(m, CM[j][$h_j$(x)]);
    }
```
where *max_int* is a constant equal to the maximum admissible value for an integer type, whereas MIN is a function that returns the minimum among the two numbers passed in input.

Architecture specifications

Let us report the main architecture parameters defined in Chapter 4, by distinguishing the single and the multi-host case, in that they will be widely used in subsequent analysis.

- *Single-host PIM architecture*

  The single-host architecture reported in the following figure is characterized by:

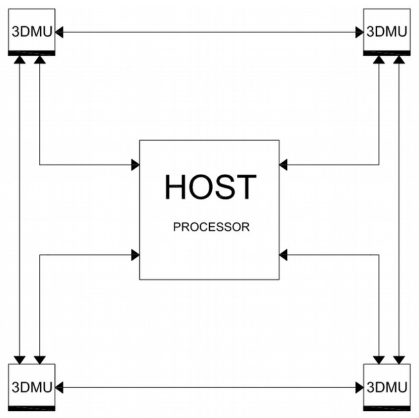

Figure 6.2: The single-host PIM architecture.

1. all the missing specifications defined in Chapter 4;
2. one host processor, i.e. P_{host} = 1, composed of $P_{host\text{-}PE}$ = 64 PEs/cores and 4 MINFs. Each MINF is logically connected to a group of 16 PEs and is physically connected to a given 3DMU through a high bandwidth SerDes link (thus, d_{net} = 1). The internal interconnect is a two-dimensional mesh with an average distance $d_{host\text{-}net\text{-}int}$ = 8;
3. four 3D-stacked memories, each one with an underlying PIM processor such that P_{pim} = N_{3DMU} = 4. The 3DMUs are chained one another forming a ring memory network that surrounds the host ("halo"-like organization). The 4-node ring average distance is $d_{mem\text{-}net}$ = 1.

   Each 3DMU has 32 memory slices and, in turn, each memory slice is composed of 8 interleaved memory modules and has a capacity of 256 MB;
4. Each one of the four PIM processors is composed of $P_{pim\text{-}PE}$ = 16 PIM PEs, one MINF and a crossbar internal interconnection.

   Each PIM PE is logically assigned to a single memory slice of the 3DMU in which it is located. The same assignment is repeated for every host PE; as a matter of fact, every host PE group, logically assigned to a given MINF, is closer to a certain 3DMU with respect to the others. As said before, each 3DMU has 32 memory slices of which 16 have been logically assigned to the underlying PIM cores, whereas the other 16 memory slices are assigned to the group of 16 host PEs ("closer" to the given 3DMU).

   Therefore, the whole architecture has a logical NUMA organization due to the one-PE-per-memory slice mapping;
5. cache coherence is automatic, directory-based, with *home-flush* and local invalidation optimization mechanisms. Home nodes are chosen statically;
6. process run-time support is with exclusive-mapping, therefore the maximum number of PEs that can be exploited is N_{pim} = 64 and N_{host} = 64 for PIM and host cores respectively, and I/O-based RDY/ACK communications.

- *Multi-host PIM architecture*

  The multi-host architecture reported in the following figure is characterized by:

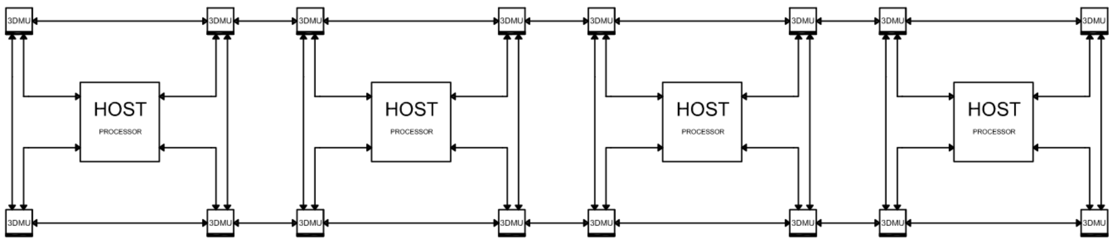

Figure 6.3: The multi-host PIM architecture.

1. all the missing specifications detailed in Chapter 4;
2. P_{host} = 4 host processors, each one composed of $P_{host\text{-}PE}$ = 16 PEs/cores and 4 MINFs. Each MINF is logically connected to a group of 4 PEs and is physically connected to a given 3DMU through a high bandwidth SerDes link (thus, again d_{net} = 1). The internal interconnect is a two-dimensional mesh with an average distance $d_{host\text{-}net\text{-}int}$ = 4;
3. each one of the four single-host logical sub-system has the same organization and <u>local</u> parameters value of the previous case; global parameters can be defined considering: the total number of 3DMU and PIM processors, i.e. P_{pim} = N_{3DMU} = 16, the total number of host and PIM PEs, i.e. N_{pim} = 256 and N_{host} = 64, and the average distance of the global memory network, i.e. a ring with 16 nodes and $d_{mem\text{-}net\text{-}global}$ = 4;
4. The not mentioned parameters and further considerations are the same of the previous single-host case.

Q analysis

Let us assume that our PIM architecture, both in the case of a single or multi-host, constitutes a single node of a cluster of workstations; therefore, input stream items are received from the external cluster network and are written in memory by an intelligent NIC through the efficient *RDMA* (Remote Direct Memory Access) facility. In this case, the intelligent NIC computation acts as an "external" producer process that cooperates with *Q*.

A RDY/ACK communication model among the NIC computation, i.e. basically a *send* primitive, and *Q* can be implemented as well exploiting interrupts (NIC to *Q*'s PE) and Memory-Mapped I/Os (*Q*'s PE to NIC) for sender-receiver synchronization.

Thus, every integer type item *x* is written in a target variable of size σ_I (padding is properly added due to the low integer type size) and is read from memory by *Q* that starts its computation.

The *Q* computation can be compiled using the D-RISC ([1]), i.e. RISC-like, assembler formalism according to the logical scheme reported in the following.

We should highlight that the goal of this part is to evaluate the "pure" code calculation time of *Q* when it is executed over a pipelined scalar CPU (i.e. over a PIM or host PE).

```
START:          < receive x >
                < compute x >    // it could be  < update x > or < query x >
                < set_ack >      // set ACK = 1
                GOTO START
```

The *receive-set_ack* assembler code is provided in [1] and has a low calculation time, i.e. $T_{setup} \sim 10\ \tau$ (notice that *set_ack* refers to the RDY/ACK cooperation model of section 4.1).

Let us compile the compute part, by distinguishing the update and query procedures, in order to derive its calculation time:

```
< compute x >

     < update x >
               CLEAR     Rj
     LOOP:     CALL      Rh_j, Rret
                         // input stream element value is in Rx from < receive x > and output is put in Rt
               LOAD      RCM, Rt, Rcm
               INCR      Rcm
               STORE     RCM, Rt, Rcm
               ADD       RCM,RD,RCM
               INCR      Rj
               IF <      Rj, RD, LOOP


     < query x >
               CLEAR     Rj
               MOV       Rmax_int, Rm     // Rm is set with max_int value in Rmax_int
     LOOP:     CALL      Rh_j, Rret
                         // input stream element value is in Rx from < receive x > and output is put in Rt
               LOAD      RCM, Rt, Rcm
```

```
            IF >      Rcm, Rm, SKIP
            MOV       Rcm, Rm   // at end the estimate for x is in Rm
  SKIP:     INCR      Rj
            IF <      Rj, RD, LOOP
```

The above code has been optimized, according to the methodology reported in [1] concerning the pipelined-CPU compilation optimizations, and the resulting “pure” code calculation times are:

- $T_{calc\text{-}0\text{-}update} = (T_{hj} + 8\tau)\, D = 920\ \tau$

- $T_{calc\text{-}0\text{-}query} = (T_{hj} + 6.6\tau)\, D = 864\ \tau$

with $D = 40$ and the calculation time for a single hash function T_{hj} equal to 15 τ; as suggested in [40], it has been calculated using a bitmasking operation in place of the more time consuming mod w instruction exploiting the fact that w is a power of 2.

Let us now evaluate the number of LLC faults $N_{fault\text{-}LLC}$ in order to derive the effective internal calculation time of Q module. Although we could potentially define $N_{fault\text{-}LLC\text{-}host}$ and $N_{fault\text{-}LLC\text{-}PIM}$, with $N_{fault\text{-}LLC\text{-}host}$ lower than $N_{fault\text{-}LLC\text{-}PIM}$ due to the host PE multi-level cache hierarchy and, therefore, the higher probability to exploit spatial and/or temporal locality, we simply define $N_{fault\text{-}LLC}$ because caches cannot be exploited at all with the given CM sketch computation (thus: $N_{fault\text{-}LLC\text{-}host} = N_{fault\text{-}LLC\text{-}PIM} = N_{fault\text{-}LLC}$).

As a matter of fact, the CM sketch size is about 168 MB, thus much greater than any cache capacity, and accesses are performed, one for every row, completely random with no prefetching opportunities due to the too fine-grained computation (the mean time among two consecutive row random accesses is very short) and to the difficulties to prefetch effectively hashing data structures ([49]).

Therefore, the LLC fault number is equal to D and it occurs for every input stream item (neglecting one fault for every input stream item reading access and instruction faults that occur only for the first element).

The internal calculation time for Q can be evaluated as:

$$T_{calc} = T_{calc\text{-}0} + T_{fault} = T_{calc\text{-}0} + N_{fault\text{-}LLC}\ T_{transf}(\sigma_1)$$

where $T_{transf}(\sigma_1)$ is the latency needed to transfer a primary cache block from memory.

The above formula is very general and it can be used to evaluate the computational characteristics of a given processing module; more specifically, we will classify a general process computation as compute-intensive if $T_{calc-0} >> T_{fault}$ and, on the contrary, as memory-intensive if $T_{calc-0} << T_{fault}$. This classification can be applied to every structured parallel application considering every single processing module of which it is made up. Although this seems a complex task, we would be generally interested in evaluating computational characteristics of functional modules. Moreover, if a parallel pattern with functional replication is exploited (more probable case), then every processing module has the same computational characteristics of the others, such that only a "one-for-all" module evaluation is needed.

Once we know the computational characteristics of a single processing module, or parallel application, we can then think to a host cores mapping, if it is compute-intensive, or PIM cores mapping if, on the contrary, is memory-intensive.

Thus, applying what said so far to the Q computation, by distinguishing according to the single and multi-host architectures, we can proceed as follows:

- *Single-host PIM architecture*

  Let us assume to map the Q module over a certain host PE; then, from Chapter 4, we have $T_{transf}(\sigma_1) = L_{read-C1-host} = 71\tau$. Thus, for $N_{fault-LLC} = D = 40$:

  - $T_{calc-update} = T_{calc\text{-}0\text{-}update} + D\ T_{transf}(\sigma_1) = 920\ \tau + 2840\ \tau = 3760\ \tau$

  - $T_{calc-query} = T_{calc\text{-}0\text{-}query} + D\ T_{transf}(\sigma_1) = 864\ \tau + 2840\ \tau = 3704\ \tau$

  therefore, for both phases, the time needed to retrieve data from memory is more than three times greater with respect to the "pure" code computation time.

  Let us try with a PIM mapping, for which $T_{transf}(\sigma_1) = L_{read-C1-PIM} = 25\ \tau$:

  - $T_{calc-update} = T_{calc\text{-}0\text{-}update} + D\ T_{transf}(\sigma_1) = 920\ \tau + 1000\ \tau = 1920\ \tau$

- $T_{calc\text{-}query} = T_{calc\text{-}0\text{-}query} + D\, T_{transf}(\sigma_1) = 864\,\tau + 1000\,\tau = 1864\,\tau$

thus, for both phases, the time needed to transfer primary cache blocks from memory is comparable to the "pure" code computation time.
Actually this result is not new since, as reported in [31], different scientific computing kernels that are memory-intensive over a general processor execution, become compute-intensive over a PIM execution (or compute and memory activities are comparable as in our case).

- *Multi-host PIM architecture*

  Let us assume to map the *Q* module over a certain host PE of a given single-host logical sub-system.
  From Chapter 4 we have: $T_{transf}(\sigma_1) = L_{read\text{-}C1\text{-}local\text{-}host} = 63\tau$. Thus:

  - $T_{calc\text{-}update} = T_{calc\text{-}0\text{-}update} + D\, T_{transf}(\sigma_1) = 920\,\tau + 2520\,\tau = 3440\,\tau$

  - $T_{calc\text{-}query} = T_{calc\text{-}0\text{-}query} + D\, T_{transf}(\sigma_1) = 864\,\tau + 2520\,\tau = 3384\,\tau$

  Instead with a PIM mapping, we have the same cost of the single-host case with similar considerations.

Thus, we can conclude that the ideal service time of a single module *Q,* taking into account all the variants detailed so far, is:

- *Single-host PIM architecture*

$$T_{Q\text{-}id\text{-}update} = \begin{cases} 1920\,\tau & \textit{PIM mapping} \\ 3760\,\tau & \textit{host mapping} \end{cases}$$

$$T_{Q\text{-}id\text{-}query} = \begin{cases} 1864\,\tau & \textit{PIM mapping} \\ 3704\,\tau & \textit{host mapping} \end{cases}$$

- *Multi-host PIM architecture*

$$T_{Q\text{-}id\text{-}update} = \begin{cases} 1920\,\tau & PIM\ mapping \\ 3440\,\tau & host\ mapping \end{cases}$$

$$T_{Q\text{-}id\text{-}query} = \begin{cases} 1864\,\tau & PIM\ mapping \\ 3384\,\tau & host\ mapping \end{cases}$$

Let us finally evaluate the energy costs of the *Q* sequential computation exploiting cost models presented in Chapter 5.

Here we should distinguish two cases merely according to the kind of mapping chosen; as a matter of fact, the *Q* allocation is performed considering only a single-host logical subsystem when targeting a multi-host PIM architecture; thus, energy costs for the single and multi-host variants coincide (for a sequential module allocation).

Therefore, we have:

- *host mapping*

  In this case, the host PE accesses an external directly connected 3DMU, in which the CM sketch is stored, consuming $E_{host\text{-}mem\text{-}acc}(\sigma_1) \sim 2.95$ nJ, with $d_{net} = 1$, for every LLC fault; thus:

$$E_{seq\text{-}host} \sim E_{host\text{-}mem\text{-}acc}(\sigma_1)\ N_{fault\text{-}LLC} = 118 \text{ nJ}$$

- *PIM mapping*

  In this case, the PIM core accesses a local 3DMU memory slice, in which the CM sketch is stored, consuming $E_{pim\text{-}mem\text{-}acc}(\sigma_1) \sim 1.41$ nJ for every LLC fault, thus:

$$E_{seq\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1)\ N_{fault\text{-}LLC} = 56.4 \text{ nJ}$$

Summarizing, we can conclude that, for every studied variant, when a PIM core mapping is exploited for the sequential *Q* module allocation, then 50% reduction both in execution time and energy consumption is achieved with respect to a host core allocation choice.

6.2.3 Parallel analysis with a single-host PIM architecture – CM Sketch Update

As previously anticipated, the parallel strategy chosen for the CM sketch algorithm, both for the update and query procedures, is characterized by the functional replication of the sequential module *Q*. Thus, assuming that the optimal parallelism degree is given, the energy and performance analysis related to the set of independent functional modules is straightforward; what should be accurately studied is how to effectively distribute/partition the input data stream in order to maximize the offered bandwidth.

In the following, different parallelization variants will be proposed for the update procedure computation, each one characterized by a different input data partitioning scheme and, consequently, a different parallelism degree resulting from the effectiveness of the strategy adopted. As a matter of fact, we have previously shown that the optimal parallelism degree n_{opt} is inversely proportional to the mean time needed to perform the input data distribution T_{DD}.

Moreover, we will assume that each *worker*$_i$ node (namely the cache controller of the PE in which it is mapped) is *home* node of the shared target variables blocks associated to the channel *DD-worker*$_i$, used to connect each worker to a data distribution module (if any). In this way, contentions are minimized and the *home-flush* optimization provides important benefits in that no other communications except the one for the pure message copy are needed.

Centralized Scatter (map parallel pattern)

This solution is very similar to the one proposed in [38] and it consists of a centralized sequential scatter module that implements a *count-based sliding window* in order to collect a certain amount of input stream items and then scatter them to the set of workers (i.e. it is basically a *map* parallel pattern with no data collecting module).

The count-based sliding-window implementation consists of *M* consecutive *receive* primitives over the input stream elements which are then scattered into *n* partitions of *M/n* items each. Notice that each item is written in a single target variable of size σ_I = 32 bytes (padding is properly added due to the low integer type size), such that the size of the sliding-window is $M\sigma_I$ Bytes and every partition is g = $M\sigma_I/n$ Bytes, with $M \geq n$.

Thus, according to what said so far, the maximum offered bandwidth can be achieved if the following equality is satisfied:

$$T_{DD} = \frac{T_{Q-id-update}}{n}$$

such that $n \geq 1$. Thus, we can solve it as follows:

$$M\, T_{receive} + T_{scatter}(n, M) = \frac{T_{Q-id-update}}{n}$$

$$M\, T_{setup} + n\, T_{setup} + M\, T_{transm}(\sigma_1) = \frac{T_{Q-id-update}}{n}$$

$$M = \frac{T_{Q-id-update} - n^2 T_{setup}}{n\left(T_{setup} + T_{transm}(\sigma_1)\right)} \qquad (1)$$

since $M \geq n$, then the following inequality is also valid:

$$n \leq \frac{T_{Q-id-update} - n^2 T_{setup}}{n\left(T_{setup} + T_{transm}(\sigma_1)\right)}$$

Therefore, we can solve it (second-degree inequality) and then, considering only the upper integer part of the greater resulting root, we obtain:

$$n \leq \left\lceil \sqrt{\frac{T_{Q-id-update}}{2T_{setup} + T_{transm}(\sigma_1)}} \right\rceil \qquad (2)$$

from which n_{opt} can be taken as:

$$n_{opt} = \left\lceil \sqrt{\frac{T_{Q-id-update}}{2T_{setup} + T_{transm}(\sigma_1)}} \right\rceil$$

We can now substitute the resulting n_{opt} value to the lower integer part of inequality (1), such that $M \geq \lfloor(1)\rfloor$, and find the minimum value of M which satisfies both $M \geq \lfloor(1)\rfloor$

and $M \geq n$.

Before going on, it should be noted that the value of $T_{transm}(\sigma_1)$ changes according to the kind of mapping chosen for the workers set. Moreover, since input stream elements (one per block) are read from memory (where they are written by the NIC as in the sequential case), then a pipelined effect can be exploited when reading an input stream element from memory and sending it via C2C to a target worker. Thus, we can define $T_{transm}(\sigma_1)$ as the maximum value among the memory reading latency and the C2C block transfer; formally:

$$T_{transm}(\sigma_1) = max(L_{read\text{-}C1}(\sigma_1), L_{C2C}(\sigma_1))$$

where, again, $L_{read\text{-}C1}(\sigma_1)$ and $L_{C2C}(\sigma_1)$ vary according to the chosen mapping variant. Therefore, we should distinguish two cases, according to the host or PIM cores mapping, in order to find the optimal parallelism degree value:

- *PIM mapping* ($n = n_{pim}$)

  When a PIM mapping for functional modules is exploited, then we have to take into account the fact that inter-stack communications with a larger latency can take place when $n \geq P_{pim\text{-}PE}$, assuming that the scatter module is mapped over a PIM core. Thus, in this case the $T_{transm}(\sigma_1)$ value is equal to:

$$T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}PIM}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, p_{loc}))$$

  where $L_{C2C\text{-}avg}(\sigma_1, p_{loc})$ is defined as:

$$L_{C2C\text{-}avg}(\sigma_1, p_{loc}) = L_{C2C\text{-}PIM\text{-}local}(\sigma_1)\, p_{loc} + L_{C2C\text{-}PIM\text{-}remote}(\sigma_1)\, (1 - p_{loc})$$

  and, in turn, the probability of having an intra-stack/local communication is defined as:

$$p_{loc} = \begin{cases} 1 & if\, n \leq P_{pim-PE} - 1 \\ \dfrac{(P_{pim-PE} - 1)}{n} & if\, n \geq P_{pim-PE} \end{cases}$$

As for the parameter c_{ext} in Chapter 5, here we are assuming that the workers set mapping is performed starting from the PIM processor in which the scatter module is mapped; this justifies the chosen extremes of the p_{loc} definition.

As can be noted, the actual value of p_{loc} depends on n, thus also $L_{C2C\text{-}avg}(\sigma_1, p_{loc})$ and $T_{transm}(\sigma_1)$ depend on n. Consequently, an equation like $n = f\,(n)$ has to be solved iteratively.

From Chapter 4, the communication latencies costs that we need to evaluate $T_{transm}(\sigma_1)$ are: $L_{C2C\text{-}PIM\text{-}local}(\sigma_1) = 18\tau$, $L_{C2C\text{-}PIM\text{-}remote}(\sigma_1) = 68\tau$ and $L_{read\text{-}C1\text{-}PIM}(\sigma_1) = 25\tau$. Moreover, from above, we have also $T_{setup} \sim 10\tau$ and $T_{Q\text{-}id\text{-}update} = 1920\tau$.

By iteratively substituting increasing values of n that satisfy inequality (2), it can be found that the best value for n, i.e. n_{opt}, is 7. The other parameters values exploited are $p_{loc} = 1$ and $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}PIM}(\sigma_1), L_{C2C\text{-}PIM\text{-}local}(\sigma_1)) = L_{read\text{-}C1\text{-}PIM}(\sigma_1) = 25\tau$.

Thus, substituting n_{opt} to $\lfloor(1)\rfloor$ we found that also $M = 7$ (since: $M \geq \lfloor(1)\rfloor = 5$ but also $M \geq n_{opt} = 7$).

Indicating with Σ a general parallel version of the Count-Min Sketch algorithm, the resulting service time and offered bandwidth of this parallel solution, exploiting a PIM cores mapping, are:

- $T_{\Sigma\text{-}id\text{-}update} = \dfrac{T_{Q-id-update}}{n_{opt}} \sim 274.3\tau$
- $B_{\Sigma\text{-}id\text{-}update} \sim 3.64\ 10^6$ *items/sec*

From the point of view of energy consumption, the costs paid are:

- $M\,E_{pim\text{-}mem\text{-}acc}(\sigma_1) = 9.87$ nJ, for input stream elements memory reading;
- $n_{opt}\,E_{seq\text{-}pim} = 394.8$ nJ, for functional modules computations;
- no inter-stack communication is involved (i.e. $E_{scatter\text{-}pim} \sim M\,E_{pim\text{-}mem\text{-}acc}(\sigma_1)$).

Thus, we can conclude that the total energy consumption cost is $E_{par\text{-}pim} = 404.67$ nJ.

Obviously, with a host PE allocation of the scatter module both performance and energy costs would be subjected to degradations due to the external/off-chip communications.

- *Host mapping* ($n = n_{host}$)

  When a host mapping for functional modules is exploited in a single-host PIM architecture, then all the communications are performed via on-chip C2C.
  Thus, the $T_{transm}(\sigma_1)$ value in this case is equal to:

$$T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host}(\sigma_1), L_{C2C\text{-}host}(\sigma_1)) = L_{read\text{-}C1\text{-}host}(\sigma_1) = 71\tau$$

  since, from Chapter 4, $L_{C2C\text{-}host}(\sigma_1) = 36\tau$.
  In this case, $T_{transm}(\sigma_1)$ does not depend on n; hence, n_{opt} can be directly derived using the above equation from which we obtain $n_{opt} = 7$ (with $T_{Q\text{-}id\text{-}update} = 3760\tau$).
  Using now the same approach of the previous PIM mapping case, we can substitute n_{opt} to $\lfloor(1)\rfloor$ and, again, we obtain $M = 7$.
  Therefore, the resulting service time and offered bandwidth of this parallel solution, exploiting a host cores mapping, are:

  - $T_{\Sigma\text{-}id\text{-}update} = \dfrac{T_{Q-id-update}}{n_{opt}} \sim 537.15\tau$
  - $B_{\Sigma\text{-}id\text{-}update} \sim 1{,}86\ 10^6$ *items/sec*

  From the point of view of energy consumption, the costs paid are:

  - $M\, E_{host\text{-}mem\text{-}acc}(\sigma_1) = 20.65$ nJ, for input stream elements memory reading;
  - $n_{opt}\, E_{seq\text{-}host} = 826$ nJ, for functional modules computations.

  Thus, we can conclude that the total energy consumption cost is $E_{par\text{-}host} = 846.65$ nJ.

In conclusion, a PIM mapping of the parallel CM sketch update procedure, that exploits a map parallel pattern with a sequential and centralized scatter, is able to provide, as for the sequential case, about 50% reduction both in execution time (i.e. 2x speed-up and doubled offered bandwidth) and energy consumption with respect to a host mapping.
Moreover, the optimal parallelism degree exploited by both mapping is $n_{opt} = 7$. Obviously, this is a best case in that, when a PIM mapping is exploited, then no inter-stack communications are involved for $n_{opt} < P_{pim\text{-}PE}$.

Anyway, in the following analysis we will study a solution for which $n_{opt} > P_{pim\text{-}PE}$ and, therefore, inter-stack cooperation is mandatory.

Notice that we could also think to a tree-structured scatter solution mapped directly over the workers set; anyway, as seen in Chapter 5, the tree scatter solution is able to provide a small performance improvement and no energy benefits. Thus, it could be studied when we are interested to the *requested* bandwidth and the sequential scatter solution results a bottleneck.

Master-Worker parallel pattern

In this solution we will exploit the *master-worker* parallel pattern (or equivalently a *farm* pattern with no collector module) with no sliding-window implementation.

Therefore, an input stream item is read from memory by the master module and is then scheduled to a given worker according to a certain scheduling policy (e.g. round-robin, on-demand, etc.).

In its general usage, this parallel pattern is used for *state-less* parallel computation. Anyway as said above, in our case the coherency of the replicated sketch/state will be solved at the querying time. The following figure shows a graphical representation of the master-worker parallel pattern with an input stream channel.

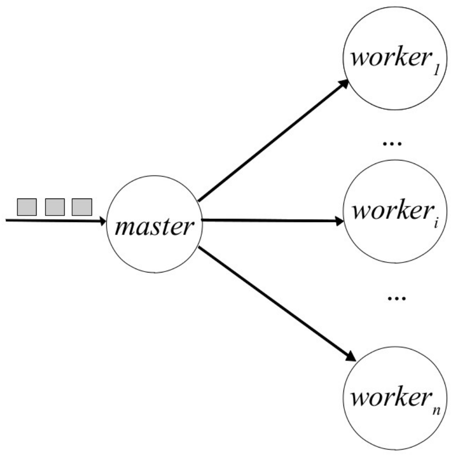

Figure 6.4: Representation of the master-worker parallel patter.

Exploiting the same approach of the previous case, the maximum offered bandwidth can be achieved when the following equality is satisfied:

$$T_{DD} = \frac{T_{Q-id-update}}{n}$$

Therefore, remembering that each item is inserted in a block of size σ_1, we can proceed as follows:

$$T_{receive} + T_{master}(1) = \frac{T_{Q-id-update}}{n}$$

$$T_{setup} + T_{setup} + T_{transm}(\sigma_1) = \frac{T_{Q-id-update}}{n}$$

we can now take n_{opt} as:

$$n_{opt} = \left\lceil \frac{T_{Q-id-update}}{2T_{setup} + T_{transm}(\sigma_1)} \right\rceil \qquad (3)$$

Let us distinguish again two cases according to the host or PIM cores mapping:

- *PIM mapping* ($n = n_{pim}$)

  Again, assuming that also the master module is mapped over a PIM core, in this case we should take into account possible inter-stack communications that can take place for $n \geq P_{pim\text{-}PE}$ with probability $(1 - p_{loc})$.

  Thus, the $T_{transm}(\sigma_1)$ value is again defined as:

  $$T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}PIM}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, p_{loc}))$$

  and it depends on n that, in turn, depends on $T_{transm}(\sigma_1)$; the same equation $n = f(n)$ has to be solved again.

  Therefore, exploiting an iterative method that assigns incremental values to n, such that the following inequality is satisfied:

  $$n \leq \left\lceil \frac{T_{Q-id-update}}{2T_{setup} + T_{transm}(\sigma_1)} \right\rceil$$

  the optimal parallelism degree obtained is $n_{opt} = 31$. The other parameters values are $p_{loc} = 15/31$ and $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}PIM}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, p_{loc})) = L_{C2C\text{-}avg}(\sigma_1, p_{loc}) \sim 43.8\tau$ with $L_{C2C\text{-}avg}(\sigma_1, p_{loc}) = L_{C2C\text{-}PIM\text{-}local}(\sigma_1)\, p_{loc} + L_{C2C\text{-}PIM\text{-}remote}(\sigma_1)\,(1 - p_{loc})$.

The resulting performance parameters for this parallel solution, exploiting a PIM cores mapping, are:

- $T_{\Sigma\text{-}id\text{-}update} = \dfrac{T_{Q-id-update}}{n_{opt}} \sim 61.94\tau$
- $B_{\Sigma\text{-}id\text{-}update} \sim 16.14\ 10^6$ *items/sec*

From the point of view of energy consumption, the costs paid for a <u>single</u> input stream item computation are:

- $E_{pim\text{-}mem\text{-}acc}(\sigma_1) \sim 1.41$ nJ, for reading the input stream element from memory;
- $E_{seq\text{-}pim} = 56.4$ nJ, for the computation of the worker that receives the item;
- $(1 - p_{loc})\ E_{pim\text{-}to\text{-}pim}(\sigma_1) \sim 1.93$ nJ, for sending the item to an external/off-stack target worker with probability $(1 - p_{loc})$.

In conclusion, the average energy consumed is $E_{par\text{-}pim} = 59.74$ nJ.
Again, with a host allocation of the master module both performance and energy costs would be subjected to degradations due to the off-chip communications among the master process and each worker in the workers set.

- *Host mapping* ($n = n_{host}$)

When a host mapping for functional modules is exploited, then all the PE-to-PE communications are performed via on-chip C2C; thus, $T_{transm}(\sigma_1)$ is again equal to:

$$T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host}(\sigma_1), L_{C2C\text{-}host}(\sigma_1)) = L_{read\text{-}C1\text{-}host}(\sigma_1) = 71\tau$$

As it can be noted, in this case $T_{transm}(\sigma_1)$ does not depend on n and we can find the optimal value for the parallelism degree by simply solving the above equation (3) and obtaining $n_{opt} = 42$.
The resulting service time and offered bandwidth of this parallel solution, exploiting a host cores mapping, are:

- $T_{\Sigma\text{-}id\text{-}update} = \dfrac{T_{Q-id-update}}{n_{opt}} \sim 89.53\tau$

- $B_{\Sigma\text{-}id\text{-}update} \sim 11.16\ 10^6$ *items/sec*

Energy consumption costs for a <u>single</u> input stream item computation are:

- $E_{host\text{-}mem\text{-}acc}(\sigma_1) \sim 2.95$ nJ, for input stream element memory reading;
- $E_{seq\text{-}host} = 118$ nJ, for the scheduled worker computation.

Thus, we can conclude that the total energy consumption cost is $E_{par\text{-}host} = 120.95$ nJ.

Comparison

In conclusion, a PIM mapping of the master-worker parallel pattern, computing the CM sketch update procedure, is able to speed-up the computation execution time of about 1.44x with respect to the host mapping (i.e. about 31% of service time reduction and, equivalently, about 44% of offered bandwidth increase). Moreover, the optimal parallelism degree exploited by the PIM mapping (i.e. 31) is lower than the one exploited by the host mapping (which is 42).

The performance gap is reduced with respect to the previous parallel solution because of the degrading inter-stack communications that take place in the PIM mapping variant. Anyway, a more fair comparison will be performed in section 6.2.5 for the multi-host case.

From the point of view of energy, the PIM mapping choice is still able to provide about 50% reduction of energy consumption.

6.2.4 Parallel analysis with a single-host PIM architecture – CM Sketch Query

As previously said, when a query for an item x arrives, it is sent in multicast to every functional module that, in turn, queries its local sketch to compute its partial results. All the partial results from each worker are then collected exploiting a certain reduce scheme and the final query result is returned.

Since this phase cannot be speeded up with respect to the sequential computation, due to the work replication, we will use the best optimal parallelism degrees found in the previous update procedure solutions; i.e. $n_{opt\text{-}pim} = 31$ for the PIM mapping and $n_{opt\text{-}host} = 42$ for the host mapping variant.

Therefore, the parallel query procedure computation over an item x consists of the following phases:

1. receive(*x*) and multicast(*x*): the item is read from memory by the data distribution module, or by the root worker if a tree-structured multicast is exploited, and it is sent in multicast to every worker in the workers set;

2. compute(x): every worker performs its computation, by querying its local sketch, and then constructs a vector of size *D* containing all its partial results (i.e. all the *D* counters values found computing the *D* hash functions over *x*);

3. reduce(+): every worker participates to the reduce collective operation exchanging its partial results vector according to the reduction scheme adopted, and, at the end, a global results vector is computed as the sum of all the partial results vectors;

4. calculation of the minimum: given the global results vector, then the minimum among its entry values, which is actually the final result of the query, is returned. The calculation of the minimum is performed in centralized manner according to the reduction scheme exploited; i.e. if a centralized solution is exploited then the centralized module calculates the minimum, otherwise, if a tree-structured solution is exploited, then the tree root worker computes it.

As it can be noted, although this approach implies a computational overhead, i.e. global reduce(+) computation, the coherence problem introduced by the adopted parallel strategy for the update procedure is finally solved.

As previously said, the best that we can do is to achieve the sequential *Q* module bandwidth and service time; thus, we should be able to overlap the multicast(*x*) and the reduce(+) phases in order to obtain an offered bandwidth comparable to the sequential one.

First of all, let us define the following calculation times obtained exploiting the same pipelined-CPU evaluation methodology (detailed in [1]) of the sequential case:

- $T_{sum\text{-}vector} = 280\tau$ is the time needed to perform the sum of two vectors of size *D*;

- $T_{min} = 250\tau$ is the time needed to perform the calculation of the minimum among the values of a vector of size D;

- $T_{w\text{-}PIM} = 1840\tau$ and $T_{w\text{-}host} = 3680\tau$ are the internal calculation time of every worker mapped over a PIM or host PE respectively (without considering the computation of the minimum that we had in the sequential case but considering the partial results vector construction).

We can now study every phase in detail by distinguishing among the PIM and host cores mapping of the workers set:

- *PIM mapping* ($n_{opt} = n_{opt\text{-}pim} = 31$)
  Multicast
  As seen in Chapter 5, to multicast an item of size σ_1 we can exploit a centralized or a tree-structured solution mapped over the same workers.
  In the first case, we would have (if the multicast module is mapped over a PIM core):

$$T_{receive} + T_{multicast}(1) = T_{setup} + n\ T_{send}(1) = T_{setup} + n\ (T_{setup} + T_{transm}(\sigma_1)) = 1677.8\tau$$

again with $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}PIM}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, p_{loc})) = L_{C2C\text{-}avg}(\sigma_1, p_{loc}) \sim 43.8\tau$, $p_{loc} = 15/31$ and $T_{setup} \sim 10\tau$. Thus, it can be potentially masked exploiting the pipeline effect with the workers set stage (an equivalent host mapping of the multicast module would not be masked since we would have $T_{multicast}(1) = 3048\tau$ due to the higher $T_{transm}(\sigma_1)$ value which is equal to $L_{C2C\text{-}host\text{-}pim}(\sigma_1) = 88\tau$).
Instead the tree-structured solution, has a service time equal to:

$$T_{receive} + T_{multicast}(1) = T_{setup} + 2\ T_{send}(1) = T_{setup} + 2\ (T_{setup} + T_{transm}(\sigma_1)) = 166\tau$$

with $T_{transm}(\sigma_1)$ that, in the worst case, is equal to the external PIM-to-PIM communication latency, i.e. $L_{C2C\text{-}PIM\text{-}remote}(\sigma_1) = 68\tau$.
Also this solution can be masked but this time we exploit the communication processor facility (or, equivalently, communication thread) of every PIM PE to

offload and overlap the two *send* latencies with the functional computation.
Although, in principle, both solutions can be exploited, we prefer the tree-structured one for energy efficiency reasons.
As a matter of fact, from Chapter 5 we have:

- $E_{multicast\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1) + E_{pim\text{-}to\text{-}pim}(\sigma_1)\, c_{ext} = 61.25$ nJ, with $c_{ext} = 16$;
- $E_{tree\text{-}multicast\text{-}pim} \sim E_{pim\text{-}mem\text{-}acc}(\sigma_1) + E_{pim\text{-}to\text{-}pim}(\sigma_1)\,(p_{pim} - 1) = 5.15$ nJ, with $p_{pim} = 2$.

Reduce
Although the centralized reduce solution can be potentially masked with respect to the tree-structured one, in our case it results a bottleneck; in fact:

$$T_{reduce} = n(T_{sum\text{-}vector} + T_{send}(D/\sigma_1)) = n(T_{sum\text{-}vector} + T_{setup} + D/\sigma_1\, T_{transm}(\sigma_1)) = 15779\tau$$

with $T_{transm}(\sigma_1) = L_{C2C\text{-}avg}(\sigma_1, p_{loc}) \sim 43.8\tau$ and $p_{loc} = 15/31$. We have assumed a PIM core mapping of the centralized reduce module because, with a host mapping, we would have obtained even a larger value.
With a tree-structured solution, and the same parameters value of the centralized one, we have:

$$T_{reduce} = log_2\, n(T_{sum\text{-}vector} + T_{send}(D/\sigma_1)) \sim 1941.66\tau$$

with $T_{transm}(\sigma_1) = \frac{n - p_{pim}}{n-1} L_{C2C\text{-}PIM\text{-}local} + \frac{p_{pim}-1}{n-1} L_{C2C\text{-}PIM\text{-}remote}(\sigma_1) \sim 19.66\tau$, since in the tree-structured scheme only p_{pim} -1 out of n - 1 communications are external.
The energy consumption cost is: $E_{tree\text{-}reduce\text{-}pim} \sim E_{pim\text{-}to\text{-}pim}(\sigma_1)\,(p_{pim} - 1)\, D/\sigma_1 = 18.7$ nJ.
Obviously, the above time has to be added to the workers set service time.
Finally, if a tree-structured + centralized solution is exploited, with independent intra-stack reduction trees that work directly in memory and only at the end return their final results (see Chapter 5), we would have:

$$T_{reduce} = log_2\,(n/p_{pim})(T_{sum\text{-}vector} + T_{send\text{-}int}(D/\sigma_1)) + p_{pim}(T_{sum\text{-}vector} + T_{send\text{-}ext}(D/\sigma_1))$$

where $T_{send\text{-}int}$ (D/σ_1) refers to the internal communication among two PIM cores within the same PIM processor, thus we have $T_{transm}(\sigma_1)$ = $L_{C2C\text{-}PIM\text{-}local}$ (σ_1) = 18τ, whereas $T_{send\text{-}ext}$ (D/σ_1) refers to the external communication among a PIM core and the host core in which the centralized reduce module is mapped, i.e. $T_{transm}(\sigma_1)$ = $L_{C2C\text{-}PIM\text{-}host}(\sigma_1)$ = 88τ.
Therefore, the first member is equal to 1520τ and it has to be added to the service time of the workers set stage; the second member is equal to 1460τ and it could be potentially masked.
If this solution is chosen, then the centralized reduce module has to compute also the local minimum with an increment of its service time to 1710τ which, anyway, does not constitute a bottleneck (as a matter of fact, if a tree-structured solution is exploited, then the local minimum calculation is performed by the tree-root worker with an increase of the reduce latency and, therefore, of the workers set service time).
Energy consumption costs associated to the tree + centralized reduce solution are: $E_{tree+centralized} \sim E_{host\text{-}to\text{-}pim}(\sigma_1)\, p_{pim}\, D/\sigma_1 = 20$ nJ. Thus, it is a bit more than the tree-structured solution but this is true only for $p_{pim} \leq 2$, i.e. rare case in a large-scale or highly-parallel application.
In conclusion, the better energy-performance trade-off is provided by the tree-structured + centralized solution and, therefore, it will be exploited for our parallel solution of the query procedure.
The service time of each worker becomes $T_{w\text{-}PIM}$ = 1840τ + 1520τ = 3360τ.
The resulting service time and offered bandwidth of this parallel solution, exploiting a PIM cores mapping, are:

- $T_{\Sigma\text{-}id\text{-}query} = T_{w\text{-}PIM} = 3360\tau$
- $B_{\Sigma\text{-}id\text{-}query} \sim 297.62\ 10^3$ *items/sec*

The time increase with respect to the sequential case is of about 1.8x times, i.e. 80% of increment and 44% degradation of the offered bandwidth.
The total energy consumption cost is:

$$E_{tree\text{-}multicast\text{-}pim} + n_{opt\text{-}PIM} E_{seq\text{-}PIM} + E_{tree+centralized} = 1773.55 \text{ nJ}$$

- *Host mapping* ($n_{opt} = n_{opt\text{-}host} = 42$)

  Multicast

  Let us adopt the same tree-structured multicast solution of the previous case with a service time equal to:

$$T_{receive} + T_{multicast}(1) = T_{setup} + 2\ T_{send}(1) = T_{setup} + 2\ (T_{setup} + T_{transm}(\sigma_1)) = 172\tau$$

  with $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host}(\sigma_1), L_{C2C\text{-}host}(\sigma_1)) = L_{read\text{-}C1\text{-}host}(\sigma_1) = 71\tau$. Again it can be overlapped with the functional computation exploiting a communication processor or thread facility.

  No energy costs are paid, except $E_{host\text{-}mem\text{-}acc}(\sigma_1)$ for reading the input stream item from memory, since on-chip C2C communications are exploited at all.

  Reduce

  Furthermore, we use an efficient tree-structured reduce for our parallel solution with latency:

$$T_{reduce} = log_2\, n(T_{sum\text{-}vector} + T_{send}(D/\sigma_1)) \sim 2534.39\tau$$

  and no energy consumption costs, thanks to the on-chip C2C communications.

  The reduce latency time and the calculation of the minimum, performed by the tree-root worker, have to be added to the service time of the workers set stage that becomes equal to: $T_{w\text{-}host} = 3680\tau + 2534.39\tau + 250\tau = 6464.39\tau$.

  The resulting service time and offered bandwidth of this parallel solution, exploiting a host cores mapping, are:

  - $T_{\Sigma\text{-}id\text{-}query} = T_{w\text{-}host} = 6464.39\tau$
  - $B_{\Sigma\text{-}id\text{-}query} \sim 154.69\ 10^3$ *items/sec*

  The increasing in time with respect to the sequential case is of about 1.74x, i.e. 74% of increment and 42% degradation of the offered bandwidth (slightly less with respect to PIM mapping case thanks to the on-chip communications exploitation).

  The total energy consumption cost is:

$$E_{host\text{-}mem\text{-}acc}(\sigma_1) + n_{opt\text{-}host} E_{seq\text{-}host} = 4958.95 \text{ nJ}$$

Comparison

In conclusion, a PIM mapping of the query procedure parallel program, which consists of a multicast + map + reduce structured computation, is able to provide a better offered bandwidth and energy efficiency with respect to the host mapping exploitation.

In particular, the offered bandwidth is about 1.92x greater than the one achieved with the host mapping and, even more, energy consumption is reduced of about 2.8 times exploiting a PIM mapping (i.e. about 64% reduction).

Thus, positive results have been achieved with a PIM cores exploitation although, in the host mapping case, efficient on-chip C2C communications can be adopted.

6.2.5 Parallel analysis with a multi-host PIM architecture – CM Sketch Update

As previously detailed, the sequential service time cost does not change in the PIM mapping case but it changes for the host mapping one due to the different architectural specifications (see section 6.2.2). Thus, we have:

$$T_{Q\text{-}id\text{-}update} = \begin{cases} 1920\,\tau & \textit{PIM mapping} \\ 3440\,\tau & \textit{host mapping} \end{cases}$$

with a lower gap (i.e. about 1.8x time reduction with a PIM mapping exploitation) with respect to the sequential execution over a single-host PIM architecture (i.e. were we had about 2x time reduction on favour of PIM mapping).

Let us adopt again the master-worker parallel pattern in order to study the parallel update procedure in the multi-host case; it is able to provide higher values for the optimal parallelism degree with respect to the other experimented map pattern solution.

Exploiting the same approach of section 6.2.3, we can find the optimal parallelism degree with the following equation:

$$n_{opt} = \left\lceil \frac{T_{Q-id-update}}{2\,T_{setup} + T_{transm}(\sigma_1)} \right\rceil$$

We should now distinguish again different cases according to the kind of mapping that can be exploited. Anyway, when a PIM mapping is exploited involving only a single-host logical sub-system, then we obtain the same $n_{opt\text{-}pim}$ value (i.e. 31) and the same results of section 6.2.3.

A formal study of the PIM mapping variant involving different single-host logical sub-system should be performed for values of $n_{opt\text{-}pim}$ that are larger than the number of PIM cores available in a single-host logical sub-system.

Therefore, we should now study only a host mapping of the functional modules that this time involves more than one host (as a matter of fact the number of host processors is now $P_{host} = 4$, each one with $P_{host\text{-}PE} = 16$ cores).

- *Host mapping* ($n = n_{host}$)

  When a host mapping of the workers set is exploited in a multi-host PIM architecture, then external/off-chip communications among host PEs with a larger latency can take place for $n \geq P_{host\text{-}PE}$, assuming that the master module is mapped over a host core.

  Thus, the $T_{transm}(\sigma_1)$ value should be evaluated as:

$$T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host\text{-}local}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, q_{loc}))$$

  where $L_{C2C\text{-}avg}(\sigma_1, q_{loc})$ is defined as:

$$L_{C2C\text{-}avg}(\sigma_1, q_{loc}) = L_{C2C\text{-}host\text{-}local}(\sigma_1)\, q_{loc} + L_{C2C\text{-}host\text{-}remote}(\sigma_1)\,(1 - q_{loc})$$

  and, in turn, the probability of having an off-chip/external communication among two PEs in different host processors is defined as:

$$q_{loc} = \begin{cases} 1 & if\, n \leq P_{host-PE} - 1 \\ \dfrac{(P_{host-PE} - 1)}{n} & if\, n \geq P_{host-PE} \end{cases}$$

As for parameters c_{ext} in Chapter 5 and p_{loc} in section 6.2.3, here we are assuming that the workers set mapping is performed starting from the host processor in which the master module is mapped; this justifies the chosen extremes above.
Again, the actual value of q_{loc} depends on n and an equation like $n = f(n)$ has to be solved iteratively, such that the following inequality is satisfied:

$$n \leq \left\lceil \frac{T_{Q-id-update}}{2T_{setup} + T_{transm}(\sigma_1)} \right\rceil$$

The optimal parallelism degree found is $n_{opt} = 34$ (thus the number of host processors involved in the computation is $p_{host} = 3$). The other parameters values are $q_{loc} = 15/34$, $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host\text{-}local}(\sigma_1), L_{C2C\text{-}avg}(\sigma_1, q_{loc})) = L_{C2C\text{-}avg}(\sigma_1, q_{loc}) \sim 79.42\tau$ and, from Chapter 4, $L_{C2C\text{-}host\text{-}remote}(\sigma_1) = 120\tau$ and $L_{C2C\text{-}host\text{-}local}(\sigma_1) = 28\tau$.
The resulting performance parameters are:

- $T_{\Sigma\text{-}id\text{-}update} = \dfrac{T_{Q-id-update}}{n_{opt}} \sim 101.18\tau$
- $B_{\Sigma\text{-}id\text{-}update} \sim 9.88\ 10^6$ *items/sec*

From the point of view of energy consumption, the costs paid for a single input stream item computation are:

- $E_{host\text{-}mem\text{-}acc}(\sigma_1) \sim 2.95$ nJ, for reading the input stream element from memory;
- $E_{seq\text{-}host} = 118$ nJ, for the computation of the worker that receives the item;
- $(1 - q_{loc})\ E_{host\text{-}to\text{-}host}(\sigma_1) \sim 4.47$ nJ, for sending the item to an external/off-chip target worker with probability $(1 - q_{loc})$.

In conclusion, the average energy consumed is $E_{par\text{-}host} = 125.42$ nJ.

Comparison

To sum up, a PIM mapping of the master-worker pattern, that exploits a single-host logical sub-system with performance and energy results available in 6.2.3, is able to speed-up the computation execution time of about 1.6x with respect to a host mapping that involves $p_{host} = 3$ host processors (i.e. about 38% of service time reduction and, equivalently, about 60% of offered bandwidth increase). Moreover, the optimal parallelism degree exploited by the PIM mapping (i.e. 31) is slightly lower than the one exploited by the host mapping (which is 34).

From the point of view of energy, the PIM mapping choice is still able to provide more than 50% reduction of energy consumption.

6.2.6 Parallel analysis with a multi-host PIM architecture – CM Sketch Query

Let us refer to section 6.2.4 for the explanation of the strategy adopted to parallelize the query procedure and for all the parameters values, such as: $T_{sum\text{-}vector} = 280\tau$ and $T_{min} = 250\tau$, except $T_{w\text{-}host}$ that is equal to 3360τ due to the different architectural specifications.

As specified in the previous section, when a PIM mapping is exploited involving only a single-host logical sub-system, then we obtain the same $n_{opt\text{-}pim}$ value (i.e. 31) and the same results of section 6.2.4. Thus, in the following, we will study only the host mapping case exploiting the $n_{opt\text{-}host}$ value found in the previous section (i.e. $n_{opt\text{-}host} = 34$).

- *Host mapping* ($n_{opt} = n_{opt\text{-}host} = 34$ and $p_{host} = 3$)

  Multicast

  Let us adopt the same tree-structured multicast solution of section 6.2.4 with a service time equal to:

$$T_{receive} + T_{multicast}(1) = T_{setup} + 2\ T_{send}(1) = T_{setup} + 2\ (T_{setup} + T_{transm}(\sigma_1)) = 290\tau$$

  with $T_{transm}(\sigma_1) = max(L_{read\text{-}C1\text{-}host\text{-}local}(\sigma_1), L_{C2C\text{-}host\text{-}remote}(\sigma_1)) = L_{C2C\text{-}host\text{-}remote}(\sigma_1) = 120\ \tau$. It can be overlapped with the functional computation exploiting the communication processor facility (or communication thread) available for each host (and PIM) PE. The total energy cost associated to the tree-structured multicast is:

  - $E_{tree\text{-}multicast\text{-}host} \sim E_{host\text{-}mem\text{-}acc}(\sigma_1) + E_{host\text{-}to\text{-}host}(\sigma_1)\ (p_{host} - 1) = 18.95$ nJ

  Reduce

  Again, we use an efficient tree-structured reduce with latency:

$$T_{reduce} = log_2\, n(T_{sum\text{-}vector} + T_{send}(D/\sigma_1)) \sim 2520.58\tau$$

and $T_{transm}(\sigma_1) = \dfrac{n - p_{host}}{n-1} L_{C2C\text{-}host\text{-}local} + \dfrac{p_{host}-1}{n-1} L_{C2C\text{-}host\text{-}remote}(\sigma_1) \sim 41.09\ \tau$; again, in the tree-structured scheme only p_{host} - 1 out of n - 1 communications are external. The energy consumption cost is $E_{tree\text{-}reduce\text{-}host} \sim E_{host\text{-}to\text{-}host}(\sigma_1)\,(p_{host} - 1)\, D/\sigma_1 = 80$ nJ. The reduce latency time and the calculation of the minimum, performed by the tree-root worker, have to be added to the service time of the workers set stage that becomes equal to: $T_{w\text{-}host} = 3360\tau + 2520.58\tau + 250\tau = 6130.58\tau$.

The resulting service time and offered bandwidth of this parallel solution, exploiting a host cores mapping, are:

- $T_{\Sigma\text{-}id\text{-}query} = T_{w\text{-}host} = 6130.58\tau$
- $B_{\Sigma\text{-}id\text{-}query} \sim 163.11\ 10^3$ *items/sec*

The increasing in time with respect to the sequential case is of about 1.8x.

The total energy consumption cost is:

$$E_{tree\text{-}multicast\text{-}host} + n_{opt\text{-}host} E_{seq\text{-}host} + E_{tree\text{-}reduce\text{-}host} = 4110.95 \text{ nJ}$$

Comparison

In conclusion, a PIM mapping of the query procedure parallel program, that exploits a single-host logical sub-system with performance and energy results available in 6.2.4, is able to provide a better offered bandwidth and energy efficiency with respect to the multiple host mapping exploitation.

In particular, the offered bandwidth is about 1.8x greater than the one achieved with the host mapping and energy consumption is reduced of about 2.32 times exploiting a PIM mapping.

It should be noted that we achieved shorter performance and energy gap with respect to the results of section 6.2.4 since the number of host cores exploited is lower (i.e. lower reduce latency that impacts on the service time) as well as the service time of each worker allocated over a host PE (due to the multi-host architecture specifications that are characterized by a lower latency to transfer a cache block from memory after a LLC fault).

6.3 Brief summary and conclusive parametric study

In this chapter, Processing-in-Memory architecture variants have been evaluated and compared in terms of how good or bad they act when executing structured parallel program examples. In so doing, all the concepts, techniques, results and cost models presented in previous chapters have been extensively used.

Results obtained show how a sequential or parallel memory-intensive application benefits from an execution that exploits PIM cores underlying main memory units, from the point of view of both energy and performance. In particular, when executing a stream-based structured parallel application, as the one proposed in this chapter, that exploits a PIM cores mapping for its functional modules, then the obtained results show that the offered bandwidth is speeded up from 1.4x to 2x with respect to a host cores mapping. In the same way, energy consumption is reduced from 50% to about 64% exploiting a PIM cores execution.

Anyway, it should be remarked that in all the previous examples *base* memory access and communication latencies have been considered, i.e. neglecting possible conflicts, although the architectural specifications and run-time support choices are such that different kinds of contentions are minimized. Thus, a deeper and more complex analysis should be performed in order to evaluate the *under-load* latency values and provide more accurate results.

In the following, a conclusive parametric study will be performed in order to provide a general idea of the PIM architectures potential in relationship to structured parallel programs executions.

Conclusive parametric study

Let us consider a functional module of a stream-based computation graph with internal calculation time T_{calc} defined as: $T_{calc} = T_{calc\text{-}0} + T_{fault} = T_{calc\text{-}0} + N_{fault\text{-}LLC}\, T_{transf}(\sigma_1)$.

Let us assume now that this functional module is executed over a PIM and a host core of a given PIM architecture. Thus, the calculation time differs in the two executions with non null probability, such that $T_{calc\text{-}pim}$ and $T_{calc\text{-}host}$ can be identified.

Assuming that the two calculation times coincide with the ideal service times of the functional module, then their components can be compared as follows:

- $T_{calc\text{-}0\text{-}\mathrm{pim}} \geq T_{calc\text{-}0\text{-}host}$: in that it is highly probable that the host processor is characterized by faster cores, with a greater switching frequency, powerful functional units, superscalar CPUs, hardware multithreading, etc.
  As a matter of fact, we have highlighted many times that a host cores mapping should be preferred for compute-intensive parallel applications;

- $N_{fault\text{-}LLC\text{-}pim} \geq N_{fault\text{-}LLC\text{-}pim}$: in that, as said in section 6.2.2, the host cores are characterized by a multi-level cache hierarchy and, therefore, by a higher probability to exploit spatial and/or temporal locality;

- $T_{transf\text{-}pim}(\sigma_1) < T_{transf\text{-}host}(\sigma_1)$ at least as far as the base memory access latency is concerned.

In a memory-intensive application, as the one studied in this chapter, it could be the case that $N_{fault\text{-}LLC\text{-}pim} \sim N_{fault\text{-}LLC\text{-}host}$, let us use simply $N_{fault\text{-}LLC}$; even more, when the memory activities are such that the "pure" calculation time is negligible, i.e. $T_{calc\text{-}0} << T_{fault}$, then the following inequality holds: $T_{calc\text{-}pim} < T_{calc\text{-}host}$.

As said above, base access latencies are considered; thus, we emphasize this fact by writing $T_{calc\text{-}pim}(R_{Q0}) < T_{calc\text{-}0\text{-}host}(R_{Q0})$, and equivalently $T_{transf\text{-}pim}(\sigma_1, R_{Q0}) < T_{transf\text{-}host}(\sigma_1, R_{Q0})$, where R_{Q0} indicates that we used base latencies to evaluate cache blocks transfers from memory and, by consequence, the internal calculation times (conversely, R_Q would indicate *under-load* latencies). It goes without saying that: $T_{transf\text{-}pim}(\sigma_1, R_{Q0}) = L_{read\text{-}C1\text{-}pim}(\sigma_1)$, whereas $T_{transf\text{-}host}(\sigma_1, R_{Q0}) = L_{read\text{-}C1\text{-}host}(\sigma_1)$.

We can now parallelize our functional module exploiting a structured parallel paradigm characterized by functional replication with independent workers, e.g. farm, map, master-worker, and derive two different case studies according to the requested bandwidth demanded to, or the offered bandwidth achieved by, our parallel application.

- *Requested Bandwidth*
  Let T_A the mean interarrival time, such that $T_A < T_{calc\text{-}pim}(R_{Q0})$, $T_A < T_{calc\text{-}host}(R_{Q0})$ and the data distribution computation does not result a bottleneck i.e. $T_A > T_{DD}$.
  Thus, the optimal parallelism degrees $n_{opt\text{-}pim}$ and $n_{opt\text{-}host}$ are equal to:

$$n_{opt\text{-}pim} = \left\lceil \frac{T_{calc-pim}(R_{Q_0})}{T_A} \right\rceil \qquad n_{opt\text{-}host} = \left\lceil \frac{T_{calc-host}(R_{Q_0})}{T_A} \right\rceil$$

From the considerations above, we have that $n_{opt\text{-}pim} < n_{opt\text{-}host}$. Therefore, the same requested bandwidth $1/T_A$ can be achieved by both PIM and host mapping but with different parallelism degrees.

The following figure summarizes what detailed so far:

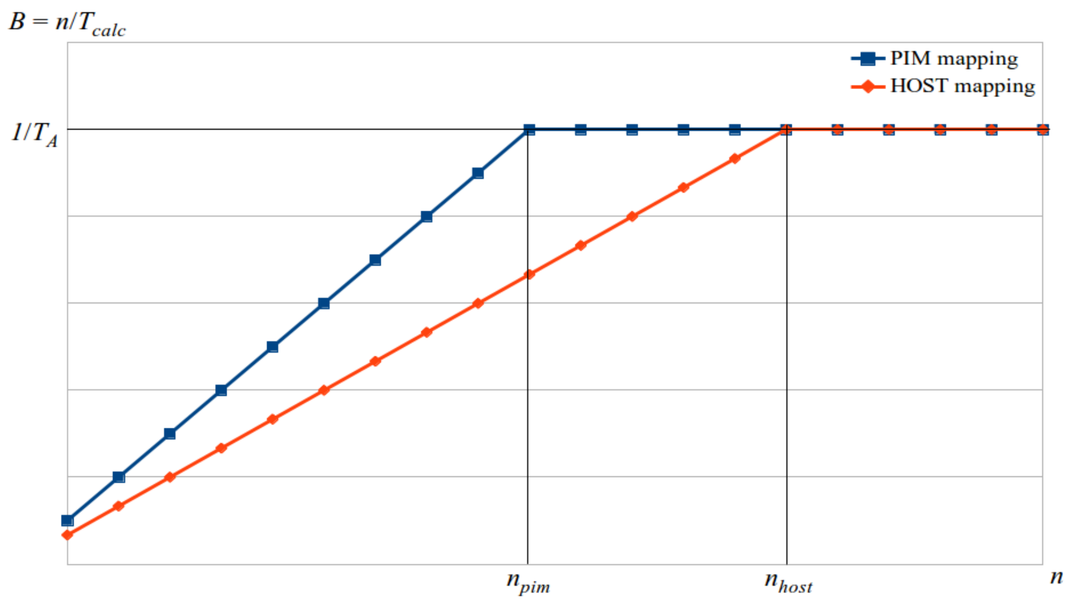

Figure 6.5: Bandwidth graph in function of the parallelism degree n.

The ratio between $n_{opt\text{-}host}$ and $n_{opt\text{-}pim}$ is equal to the ratio between $T_{calc\text{-}host}(R_{Q0})$ and $T_{calc\text{-}pim}(R_{Q0})$. It is interesting to note that, for our parallel application, i.e. a memory-intensive one with no cache exploitation, if we take the limit for $N_{fault\text{-}LLC}$ tending to infinite of the ratio between $T_{calc\text{-}host}(R_{Q0})$ and $T_{calc\text{-}pim}(R_{Q0})$, then:

$$\lim_{N_{fault-LLC} \to \infty} \frac{T_{calc-host}(R_{Q_0})}{T_{calc-pim}(R_{Q_0})} = \frac{T_{transf-host}(\sigma_1, R_{Q_0})}{T_{transf-pim}(\sigma_1, R_{Q_0})}$$

Thus, without considering conflicts and for very memory-intensive applications, the ratio between $n_{opt\text{-}host}$ and $n_{opt\text{-}pim}$ is equal to the ratio between the base memory access latencies $L_{read\text{-}C1\text{-}host}(\sigma_1)$ and $L_{read\text{-}C1\text{-}pim}(\sigma_1)$.

- *Offered Bandwidth*

  Let T_{DD} the mean time needed to distribute or schedule the input stream items, such that it results a bottleneck, i.e. $T_A < T_{DD}$.

  As we have seen in previous sections, the data distribution time changes with a PIM or host cores mapping in that $T_{transf}(\sigma_1)$ is different, according to the C2C communication latencies, and depends on n when a multi-chip mapping is involved (i.e. PIM cores mapping with more than one PIM processors and host cores mapping with multiple host processors).

  Thus, we should distinguish among $T_{DD\text{-}pim}$ and $T_{DD\text{-}host}$ as the time needed to distribute data to $n_{opt\text{-}pim}$ and $n_{opt\text{-}host}$ functional modules respectively.

  The optimal parallelism degrees $n_{opt\text{-}pim}$ and $n_{opt\text{-}host}$ are equal to:

$$n_{opt\text{-}pim} = \left\lceil \frac{T_{calc-pim}\left(R_{Q_0}\right)}{T_{DD-pim}} \right\rceil \qquad n_{opt\text{-}host} = \left\lceil \frac{T_{calc-host}\left(R_{Q_0}\right)}{T_{DD-host}} \right\rceil$$

  and should be derived exploiting an iterative method that assigns incremental values to n_{pim} and n_{host} such that the following inequalities are satisfied:

$$n_{pim} \leq \left\lceil \frac{T_{calc-pim}\left(R_{Q_0}\right)}{T_{DD-pim}} \right\rceil \qquad n_{host} \leq \left\lceil \frac{T_{calc-host}\left(R_{Q_0}\right)}{T_{DD-host}} \right\rceil$$

  Deriving a comparison between $n_{opt\text{-}pim}$ and $n_{opt\text{-}host}$ is more difficult in this case; therefore, we can reason as follows.

  Let:

$$\alpha = \frac{T_{calc-host}\left(R_{Q_0}\right)}{T_{calc-pim}\left(R_{Q_0}\right)}$$

  then we can recognize different cases according to the possible values of the data distribution times $T_{DD\text{-}pim}$ and $T_{DD\text{-}host}$:

  - if $T_{DD\text{-}pim} \sim T_{DD\text{-}host}$ then this means that $n_{opt\text{-}pim} < n_{opt\text{-}host}$, and, in particular, we can write: $n_{opt\text{-}pim} \sim n_{opt\text{-}host}/\alpha$.

Thus, as for the requested bandwidth case, the same bandwidth $1/T_{DD}$ can be achieved by both mapping choices but the PIM cores allocation allows to exploit an optimal parallelism degree which is α times lower than the one obtained with a host mapping.

- if $T_{DD\text{-}pim} > T_{DD\text{-}host}$ then this means that $n_{opt\text{-}pim} < n_{opt\text{-}host}$ but the host mapping achieves a larger bandwidth equal to $1/T_{DD\text{-}host}$.

  More specifically, let:

$$\beta = \frac{T_{DD-pim}}{T_{DD-host}}$$

  then we can write: $n_{opt\text{-}host} = \alpha\, \beta\, n_{opt\text{-}pim}$ with $\beta > 1$ and also $\alpha > 1$.

- if $T_{DD\text{-}pim} < T_{DD\text{-}host}$ then this means that a PIM mapping is able to achieve a larger bandwidth, equal to $1/T_{DD\text{-}pim}$, for $n_{opt\text{-}pim} > n_{opt\text{-}host}/\alpha$.

  More specifically, let $\gamma = 1/\beta$, then:

$$n_{opt\text{-}pim} = \frac{\gamma}{\alpha}\ n_{opt\text{-}host}$$

  In conclusion, we distinguish again three different cases according to the possible values of α and γ:

  - $\gamma > \alpha \longrightarrow n_{opt\text{-}pim} > n_{opt\text{-}host}$;
  - $\gamma \sim \alpha \longrightarrow n_{opt\text{-}pim} = n_{opt\text{-}host}$;
  - $\gamma < \alpha \longrightarrow n_{opt\text{-}pim} < n_{opt\text{-}host}$;

  The validity of this parametric results can be proved directly with specific numbers of previous sections examples.

Conclusion

Summing up, we can conclude with the following significant result:

- In the majority of cases, when a PIM cores mapping is exploited for allocating *independent* workers of a *memory-intensive* structured parallel application, characterized by functional replication, we are able to achieve the *best or the same* bandwidth with a *lower or equal* parallelism degree with respect to a host core mapping.

This result is very significant if we are interested in exploiting an exclusive mapping in order to maximize the application bandwidth and minimize energy consumption in relationship to the number of active cores exploited for the parallel computation.

Chapter 7

Conclusion and Future Works

The aim of this thesis was to study Processing-in-Memory architectures exploiting a structured approach and a methodology able to capture all the useful details to derive a performance and energy characterization of parallel programs executed over PIM systems. For this purpose, we laid emphasis on architectural aspects, structured parallel computations, performance and energy cost models. In so doing, we were able to perform an analytical treatment of parallel program benchmarks, executed over target PIM architectures, focusing on energy and performance aspects.

Thus, we started by collecting all the possible information, provided by the supporting literature, that could be relevant for our interests; then, we applied our methodology deriving *Abstract Machine Models* for PIM architectures. Exploiting the abstract model, we studied two different kinds of PIM architecture variants: the single-host and the multi-host one.

In order to lay the foundation for the subsequent quantitative analysis about parallel program examples targeting PIM architectures, we started deriving numeric values for communication latencies among system components. For this purpose, we exploited parametrized versions of the abstract architectures previously described. Thus, we defined specific network topologies, a specific number of PIM cores per PIM processor, as well as the number of host cores per host processor, links bandwidth, cache blocks size, memory and processor clock times, number of 3D memories, etc.; in so doing, we were able to calculate base communication latencies for every inter-unit cooperation of interest.

Moreover, we defined novel energy cost models able to estimate the energy consumed by a parallel program in terms of the amount of cache blocks transfers among system components it requires. We proposed different examples and communication pattern variants; among all, the reduce scheme with intra-stack or in-memory reduction trees, taking advantages from the single-host PIM architecture organization, has been analytically demonstrated to be the best solution among the one proposed, considering both energy and performance factors.

Finally, a comparative study and an evaluation of PIM architectures with parallel program examples has been carried out. For this purpose, we exploited parallel version variants of the *Count-Min Sketch* algorithm working on a sketch data structure, not fitting in cache, with a highly irregular data access pattern. Furthermore, in the analysis we distinguished among single and multi-host PIM architecture alternatives.

Results show how, in a single-host PIM architecture, a PIM cores mapping of the parallel application's functional modules is able, on average, to speed-up the computation ranging from 1.4x, for medium-high parallelism, to 2x, for low parallelism degree, with respect to the host cores mapping. The degradation is due to the inter-stack communications that take place between PIM cores for higher values of the parallelism degree. Instead, in the multi-host PIM architecture case, a PIM cores mapping of the functional modules is able, on average, to speed-up the computation of about 1.6x for a medium parallelism degree with respect to the host cores mapping. It is interesting to note that, in all the above mentioned cases, the PIM cores mapping provides an higher bandwidth always exploiting a lower parallelism degree. This factor is very significant if we are interested in exploiting an exclusive mapping (at most one process per processor) in order to maximize the parallel application bandwidth and minimize energy consumption in relationship to the number of active cores.

As for the energy characterization, numerical results show that a PIM cores mapping is able to reduce energy consumption of at least 50% to about 64% with respect to a host mapping exploitation.

At the end of this work, we carried out a formal parametric study that quantitatively provides an idea of PIM architectures benefits when executing stream-based and memory-intensive structured parallel applications. Results show that a PIM cores mapping of the parallel program's functional modules, with respect to a host cores one, is able to satisfy the same requested bandwidth with a lower parallelism degree. Instead, in relationship to the offered bandwidth, distinct cases have been distinguished with different and sometimes opposite results. Anyway, in the majority of cases, the analytical model shows that a PIM cores mapping is able to achieve, for a structured parallel application characterized by functional replication with independent workers, the best or the same bandwidth with a lower or equal parallelism degree.

7.1 Future works and open research problems

The materials presented in this work have been intended as starting point for further research and future refinements. Among all the topics treated, two of them deserve a more accurate and formal study:

- The run-time support of parallel programs targeting PIM architecture should be formally and accurately studied in future works, also in relationship to the advancements performed by the research world.
  It should be remarked that literature information are missing about this aspect. For this reason, although it is widely confirmed that PIM architectures provide benefits to sequential and/or parallel applications that do not exploit caches effectively, at least for the functional computation per se, a more detailed treatment should be performed about the parallel program run-time support. As a matter of fact, it could be the case that run-time support data structures are characterized by spatial and/or temporal locality, such that the cache hierarchy exploitation still provide a great advantage.

- A formal under-load analysis evaluating contention issues in PIM architectures and in parallel program run-time supports should be carried out. In this way, all the *base* communication latencies detailed in previous chapters can be refined by taking into account possible conflicts that could happen when executing a parallel program over a parallel architecture. The concluding parametric study, at the end of Chapter 6, would mainly benefit from this kind of study in that the results, provided by the exploitation of *under-load* communication latencies, would be even more accurate.

In addition to the future improvements to this particular work, it is worth noting that there are many open research problems and many challenges that should be faced before PIM adoption can become reality.

Apart from the physical/hardware level issues, such as choosing kind and complexity of the processing logic to be used for PIM exploitation, at the higher-levels of the system structure, existing programming models and run-time systems should be adapted, or new ones designed, and algorithms should be restructured in order to make them aware of the data location.

Thus, a software-hardware co-design should be advocated in order to realize parallel architectures, programming models and run-time systems that enforce the computation locality concept which, in this new environment, results in distributing functional computations into the memory hierarchy, closer to where data reside [2].

Bibliography

[1] M. Vanneschi, *High Performance Computing. Parallel Processing Models and Architectures*, Pisa University Press, 2014.

[2] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insights from a Micro-46 Workshop", *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul./Aug. 2014.

[3] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, M. Ignatowski, "A processing in memory taxonomy and a case for studying fixed-function pim," in *1st Workshop on Near-Data Processing (WoNDP)*, Dec. 2013.

[4] D. P. Zhang, N. Jayasena, A. Lyashevsky et al., "A new perspective on processing-in-memory architecture design," in P*roceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '13. New York, NY, USA: ACM, 2013, pp. 7:1–7:3.

[5] M. Islam, M. Scrbak, K. M. Kavi, "Improving Node-level MapReduce Performance using Processing-in-Memory Technologies", in *Workshop on Unconventional High Performance Computing*, 2014.

[6] D. Zhang, N. Jayasena, A. Lyashevsky et al., "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC' 14. New York, NY, USA: ACM, 2014, pp. 85–98.

[7] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors", in *International Symposium on Computer Architecture*, IEEE, 2008.

[8] J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System. A Retrospective Paper", in *International Conference on Computer Design*, IEEE, 2012.

[9] M. L. Chu, N. Jayasena, D. P. Zhang, M. Ignatowski, "High-level programming model abstractions for processing in memory," in *Workshop on Near-Data Processing (WoNDP)*, Dec 2013.

[10] P. Ranganathan, "From Microprocessors to Nanostores: Rethinking Data-Centric Systems," *Computer*, vol. 44, no. 1, 2011, pp. 39-48.

[11] S. H. Pugsley, J. Jestes, H. Zhang, et al., "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads", in *International Symposium on Performance Analysis of Systems and Software*, 2014.

[12] M. Ferdman, A. Adileh, O. Kocberber, et al., "A Case for Specialized Processors for Scale-Out Workloads", in *Micro*, pp. 31-42. IEEE, 2014.

[13] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," *SC'99*, November 1999.

[14] B. R. Gaeke, P. Husbands, H. J. Kym, X. S. Li, H.J. Moon, L. Oliker, K. A. Yelick, and R. Biswas, "Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines", in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS),* Ft. Lauderdale, FL. April, 2002.

[15] D. A. Reed, J. Dongarra, "Exascale Computing and Big Data", in *Communications of the ACM*, Vol. 58 No. 7, Pages 56-68, July 2015.

[16] R. Nair, S. F. Antao, C. Bertolli, P. Bose, et al., "Active Memory Cube: A processing in-memory architecture for exascale systems", IBM J. Res. & Dev., vol. 59, No. 2/3, Paper 17, March/May 2015.

[17] M. Gokhale, "Near Data Processing: are we there yet?", in *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[18] E. Azarkhish, D. Rossi, I. Loi, L. Benini, "A Logic-base Interconnect for Supporting Near Memory Computation in the Hybrid Memory Cube", in *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[19] G. Kim, J. Kim, J. Ho Ahn, and Y. Kwon, "Memory Network: Enabling Technology for Scalable Near-Data Computing" in *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[20] G. Kim et al., "Memory-centric system interconnect design with hybrid memory cubes," in PACT'13.

[21] Z. Guz, M. Awasthi, V. Balakrishnan, M. Ghosh, A. Shayesteh, and T. Suri "Realtime-Analytics is the Killer Application for Processing-In-Memory", *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[22] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design", *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[23] Y. Eckert, N. Jayasena, and G. Loh, "Thermal Feasibility of Die-Stacked Processing in Memory", *2nd Workshop on Near Data Processing (WoNDP)*, December 2014.

[24] S. Pughsley, J. Jestes, R. Balasubramonian, et al., "Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads" in *Micro, IEEE* (Volume:34 , Issue: 4), Pag. 44 – 52, June 2014.

[25] J.A. Ang, R.F. Barrett, R.E. Benner, D. Burke, C. Chan, J. Cook, et al., "Abstract Machine Models and Proxy Architectures for Exascale Computing", in *Proceedings of the 1st International Workshop on Hardware-Software Co-design*

for High Performance Computing (Co-HPC 2014), pp. 25-32, IEEE Press, Piscataway, May 2014.

[26] R. Nair, J. Moreno, D. Joseph, “Augmenting Memory Capabilities for Exascale Systems”, in *NNSA ASC Conference*, IBM Corporation, February 2014.

[27] P. M. Kogge, “PIM & Memory: The Need for a Revolution in Architecture”, in *ATPESC*, July 13.

[28] A. Choudhary, J. G. Searle, “Big Data + Extreme-scale. Time to Compute → Actionable Insights”, 2013.

[29] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, “Processing-in Memory: Exploring the Design Space”, *28th International Conference on the Architecture of Computer Systems (ARCS-2015)*, March 2015.

[30] G. Almasi, “PGAS languages in the exascale era”, IBM Research.

[31] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, et al., “Exascale Workload Characterization and Architecture Implications”, in *Performance Analysis of Systems and Software (ISPASS)*, IEEE, April 2013.

[32] A. Gara, “Energy Efficiency Challenges for Exascale Computing”, IBM Corporation, 2008.

[33] S. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, “Beyond the Wall: Near-Data Processing for Databases”, in *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2015.

[34] N. S. Mirzadeh, O. Kocberber, B. Falsafi, B. Grot, “Sort vs. Hash Join Revisited for Near-Memory Execution”, in *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, June 2015.

[35] L. Fiorin, E. Vermij, R. Jongerius, J. V. Lunteren, C. Hagleitner, “An energy efficient custom architecture for the SKA1-Low central signal processor”, in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, Article No. 5, 2015.

[36] M. Kim, Y. Ju, J. Chae and M. Park, “A Simple Model for Estimating Power Consumption of a Multicore Server System”, in *International Journal of Multimedia and Ubiquitous Engineering*, Vol.9, No.2, pp.153-160, 2014.

[37] Y. S. Shao and D. Brooks, “Energy Characterization and Instruction-Level Energy Model of Intel’s Xeon Phi Processor”, in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, Pages 389-394, IEEE Press, 2013.

[38] D. Thomas, R. Bordawekar, C.C. Aggarwal, P.S. Yu, “On Efficient Query Processing of Stream Counts on the Cell Processor”, in IEEE International Conference on Data Engineering, 2009.

[39] P. Ferragina, et al., “Bloom Filters and Count-Min Sketches with some of their applications”. http://www.di.unipi.it/~ferragin/Teach/Copie_Vecchie_Pagine/IR0607.html

[40] G. Gormode, S. Muthukrishnan, “Approximating Data with Count-Min Data Structure”, 2011.

[41] G. Gormode, S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications”, in *Journal of Algorithms,* Volume 55 Issue 1, Pages 58-75, April 2005.

[42] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.0. November 2014.

[43] T. Farrel, “Hybrid Memory Cube (HMC)”, Micron Technology, May 2012.

[44] D. Reed, “Exascale Computing and Big Data: Time To Reunite”, June 2015. http://cacm.acm.org/blogs/blog-cacm/188773-exascale-computing-and-big-data-time-to-reunite/fulltext

[45] J. Hruska, “Beyond DDR4: The differences between Wide I/O, HBM, and Hybrid Memory Cube”, January 2015. http://www.extremetech.com/computing/197720-beyond-ddr4-understand-the-differences-between-wide-io-hbm-and-hybrid-memory-cube

[46] P. F. Baumeister, H. Boettiger, J. R. Brunheroto, T. Hater, T. Maurer, A. Nobile, D. Pleiter, “Accelerating LBM and LQCD Application Kernels by In-Memory Processing”, in *30th International Conference, ISC High Performance 2015*, pp 96-112, July 2015.

[47] M. Danelutto, “Distributed System: Paradigms and Models”, Teaching material - master degree in Computer Science and Networking, version September 2014.

[48] D. Buono, T. De Matteis, G. Mencagli, and M. Vanneschi. “Optimizing message-passing on multicore architectures using hardware multi-threading.” In *Parallel, Distributed and Network-Based Processing (PDP)*, 22nd Euromicro International Conference, pages 262-270, Feb 2014.

[49] J. Lee, H. Kim, R. Vuduc, “When Prefetching Works, When It Doesn’t, and Why”, in *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 9, Issue 1, Article No. 2, March 2012.

[50] K. Kang , L. Benini , G. De Micheli, “A High-throughput and Low-Latency Interconnection Network for Multi- Core Clusters with 3-D Stacked L2 Tightly Coupled Data Memory”, in *IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, 2012.

[51] P.M. Kogge, “EXECUBE - A New Architecture for Scaleable MPPs”, in

International Conference on Parallel Processing, Vol. 1, 1994.

[52] C.E. Kozyrakis, S. Perissakis, D.A. Patterson, et al., "Scalable Processors in the Billion Transistor Era: IRAM", in *Computer* 30 (9) pp. 75–78, 1997.

[53] B. Ellis, *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*, Wiley, 2014.