Structured Parallel Programming and Cache Coherence in Multicore Architectures

Silvia Lametti

Supervisor
Marco Vanneschi

Chair
Pierpaolo Degano

October 4, 2015

Dipartimento di Informatica, Largo B. Pontecorvo, 3, I-56127 Pisa, Italy
SETTORE SCIENTIFICO DISCIPLINARE INF/01
It is clear that multicore processors have become the building blocks of today’s high-performance computing platforms. The advent of massively parallel single-chip microprocessors further emphasizes the gap that exists between parallel architectures and parallel programming maturity. Our research group, starting from the experiences on distributed and shared memory multiprocessor, was one of the first to propose a Structured Parallel Programming approach to bridge this gap. In this scenario, one of the biggest problems is that an application’s performance is often affected by the sharing pattern of data and its impact on Cache Coherence. Currently multicore platforms rely on hardware or automatic cache coherence techniques that allow programmers to develop programs without taking into account the problem. It is well known that standard coherency protocols are inefficient for certain data communication patterns and these inefficiencies will be amplified by the increased core number and the complex memory hierarchies.

Following a structured parallelism approach, our methodology to attack these problems is based on two interrelated issues: structured parallelism paradigms and cost models (or performance models).

Evaluating the performance of a program, although widely studied, is still an open problem in the research community and, notably, specific cost models to describe multicores are missing. For this reason in this thesis, we define an abstract model for cache coherent architectures which is able to capture the essential elements and the qualitative behaviors of multicore-based systems. Furthermore, we show how this abstract model combined with well known performance modelling techniques, such as analytical modelling (e.g., queueing models and stochastic process algebras) or simulations, provide an application- and architecture-dependent cost model to predict structured parallel applications performances.

Starting out from the behavior and performance predictability of structured parallelism schemes, in this thesis we address the issue of cache coherence in multicore architectures, following an algorithm-dependent approach, a particular kind of soft-
ware cache coherence solution characterized by explicit cache management strategies, which are specific of the algorithm to be executed. Notably, we ensure parallel correctness by exploiting architecture-specific mechanisms and by defining proper data structures in order to “emulate” cache coherence solutions in an efficient way for each computation. Algorithm-dependent cache coherence can be efficiently implemented at the support level of structured parallelism paradigms, with absolute transparency with respect to the application programmer. Moreover, by using the cost model, in this thesis we study and compare different algorithm-dependent implementations, such as those based on automatic cache coherence with respect to an original, non-automatic and lock-free solution based on interprocessor communications. Notably, with this latter implementation, in some cases, we are able to reduce the number of memory accesses, cache transfers and synchronizations and increasing computation parallelism with respect to the use of automatic cache coherence.

Current architectures do not usually allow disabling automatic cache coherence. However, the emergence of many-core architectures somewhat changed the scenario, so that some architectures, such as the Tilera TilePro64, allow to control and disable the automatic cache coherence facilities. For this reason, in this thesis we finally apply our methodology to TilePro64 platform in order provide a further validation of the results obtained by our cost model.
Contents

I Introduction 1

1 Introduction 3

1.1 The Cache Coherence Problem in the ManyCore Era 4
1.2 Parallel Paradigms and Cache Coherence 5
1.3 Our Starting Point 6
1.4 Contribution of the Thesis 7
1.5 Outline of the Thesis 8
1.6 Current Publications by the author 9

2 Background 11

2.1 CMP Architectures 11
2.2 Cache Coherence 16

2.2.1 Automatic Cache Coherence 16
2.2.2 Software or non-Automatic Cache Coherence 20
2.3 Existing Evaluations of Cache Coherence Solutions 21
2.4 Parallel Programming on CMPs 22

2.4.1 Programming Languages 23
2.4.2 Libraries 24
2.5 Summary 25

3 Our Methodology: Programming and Cost Models 27

3.1 Structured Parallel Programming 28

3.1.1 Stream Parallelism 29
3.1.2 Data Parallelism 30
3.2 Parallelization Methodology and Cost Model 33

3.2.1 Performance Modeling with Queueing Networks 36
3.2.2 Performance evaluations of modules and graph computations 40
3.2.3 Parallelism forms and cost models ........................................... 43
3.2.4 Evaluating the model parameters ............................................. 47
3.3 Summary ................................................................................. 49

II Applying Our Methodology to the Cache Coherence Problem ............... 51

4 Modelling Cache Coherent Architectures ........................................... 53
  4.1 SM Organization in CMP-based Architectures ............................. 53
    4.1.1 Single-CMP ..................................................................... 54
    4.1.2 Multiple-CMP .................................................................. 56
  4.2 An Abstract Model for CC-Architectures ................................. 56
    4.2.1 A Hierarchy-based Classification ....................................... 58
  4.3 Base Memory Access Latencies .................................................. 60
    4.3.1 Reading Operations .......................................................... 60
    4.3.2 Writing Operations .......................................................... 63
    4.3.3 Reading and Writing Operations in Multiple-CMP Architectures 68
  4.4 Benchmarks for Reading and Writing Latencies ......................... 70
  4.5 Summary ................................................................................. 72

5 Parallel Paradigms and Cache Coherence ........................................... 75
  5.1 Recognizing CC Patterns in Parallel Paradigms ......................... 76
    5.1.1 Farm .............................................................................. 76
    5.1.2 Data-Parallel: Map ........................................................... 80
    5.1.3 Data-Parallel with Stencil .................................................. 81
  5.2 Synchronizations in Shared Memory Systems ............................. 85
    5.2.1 Mutual Exclusion ............................................................. 85
    5.2.2 Global (barrier) event synchronizations ............................... 89
    5.2.3 Memory Ordering and Memory Consistency Models ............... 90
  5.3 Lock-Free Data Structure for Comm. Mechanisms ...................... 92
  5.4 Summary ................................................................................. 97

6 Cost Models for CMP-based Architectures ......................................... 99
  6.1 Cost Model for Under Load Memory Latencies ........................... 100
    6.1.1 Model Resolution ............................................................. 102
    6.1.2 Complexity vs Approximation of the model ......................... 102
    6.1.3 Memory access latency ..................................................... 104
    6.1.4 On-chip cache-to-cache transfers ...................................... 104
  6.2 On parallel program mapping and under-load evaluation .............. 105
  6.3 PEPA: Process Algebra for Quantitative Analysis ....................... 108
    6.3.1 PEPA Language ............................................................... 108
    6.3.2 PEPA for graphs ............................................................. 111
List of Figures

2.1 Examples of current CMP-based architectures .......................... 12
2.2 A first comparison of automatic and non-automatic techniques ........ 22

3.1 Pipeline parallelism form .................................................. 29
3.2 Farm parallelism form ...................................................... 30
3.3 Data-parallel with stencil parallelism form ................................. 31
3.4 Map parallelism form ...................................................... 32
3.5 MapReduce parallel paradigms ............................................. 32
3.6 The “compilation workflow” in our programming environment ........ 35
3.7 An example module graph ................................................ 36
3.8 A computation module modeled as a queueing system .................... 37
3.9 An example module graph and its queueing network representation ....... 38
3.10 A generic acyclic graph computation Σ .................................. 41
3.11 Stream-oriented pipeline modeling of task farm .......................... 44
3.12 An example of an abstract architecture with n Processing Elements
connected to the main memory through the interconnection structure ....... 48

4.1 SMP vs NUMA characterization of multiprocessor architectures ........ 54
4.2 Main memory subsystem in single-CMP architectures .................... 55
4.3 SMP vs NUMA characterization of multiple-CMP architectures ........ 56
4.4 Abstract model for a cache coherent architecture with N processing
   elements ................................................................. 57
4.5 GC implementation ......................................................... 59

5.1 Farm implementation strategies ........................................... 77
5.2 Pseudo-code for stencil computation in shared-variable implementation .... 82
5.3 Pseudo-code for stencil computation in message-passing implementation .... 83
5.4 Average latency times for the FastFlow queues on Tilera TilePro64
   and Intel processors varying the buffer size ............................. 97
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Client-server models with request-reply behaviour used to model multiprocessor systems</td>
<td>100</td>
</tr>
<tr>
<td>6.2</td>
<td>$R_Q/R_{Q_0}$ for main memory with $T_M = 3\tau$</td>
<td>101</td>
</tr>
<tr>
<td>6.3</td>
<td>$R_Q/R_{Q_0}$ for on-chip cache-to-cache transfers with $T_{PrC} = 3\tau$</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>Structured Operational Semantic of PEPA</td>
<td>109</td>
</tr>
<tr>
<td>6.5</td>
<td>Co-operating modules graph example</td>
<td>111</td>
</tr>
<tr>
<td>6.6</td>
<td>Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p = 2$) vs $p = nw + 1$ in Single-CMP with GC implemented at the shared cache level</td>
<td>118</td>
</tr>
<tr>
<td>6.7</td>
<td>Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs</td>
<td>119</td>
</tr>
<tr>
<td>6.8</td>
<td>Comparison with PEPA of $R_Q/R_{Q_0}$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in Single-CMP with GC implemented at the shared cache level</td>
<td>119</td>
</tr>
<tr>
<td>6.9</td>
<td>Comparison with PEPA of $R_Q/R_{Q_0}$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs</td>
<td>119</td>
</tr>
<tr>
<td>6.10</td>
<td>Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in Single-CMP, multiple-MINF, with a shared level cache that acts as GC</td>
<td>120</td>
</tr>
<tr>
<td>7.1</td>
<td>Abstract definition of the send-receive operations in the rdy-ack communication model</td>
<td>129</td>
</tr>
<tr>
<td>7.2</td>
<td>Algorithms of the send-receive operations in the rdy-ack communication model based on shared memory synchronizations</td>
<td>130</td>
</tr>
<tr>
<td>7.3</td>
<td>Algorithm of the zero-copy receive in the rdy-ack communication model based on shared memory synchronizations and an example of its use</td>
<td>132</td>
</tr>
<tr>
<td>7.4</td>
<td>Rdy-Ack communication data structures for asynchrony degree $k &gt; 1$</td>
<td>132</td>
</tr>
<tr>
<td>7.5</td>
<td>Rdy-Ack channel structure for the message-passing solution based on interprocessor communications</td>
<td>139</td>
</tr>
<tr>
<td>7.6</td>
<td>Pseudo-code of send, receive and set_ack operations for Rdy-Ack channel structure for the message-passing solution based on interprocessor communications</td>
<td>139</td>
</tr>
<tr>
<td>7.7</td>
<td>Pseudo-code of wait, notify and interrupt-handler operations for Rdy-Ack channel structure for the message-passing solution based on interprocessor communications</td>
<td>140</td>
</tr>
<tr>
<td>7.8</td>
<td>Rdy-Ack channel structure for the pointer-passing solution based on interprocessor communications</td>
<td>140</td>
</tr>
<tr>
<td>7.9</td>
<td>Tilera TilePro64 automatic cache coherence protocols</td>
<td>145</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>7.10</td>
<td>FastFlow stream matrix multiplication (AixBi) using different cache coherence strategies: Hash Home Node (HHN), No Home Node (NHN) and Fixed Home Node (FHN)</td>
<td>147</td>
</tr>
<tr>
<td>7.11</td>
<td>Rdy-Ack based on Shared Memory synchronizations with home-flush optimization on Tilera TilePro64</td>
<td>149</td>
</tr>
<tr>
<td>7.12</td>
<td>Asymmetric Rdy-Ack based on Shared Memory synchronizations with home-flush optimization on Tilera TilePro64</td>
<td>150</td>
</tr>
<tr>
<td>7.13</td>
<td>Rdy-Ack based on interprocessor communications on Tilera TilePro64</td>
<td>150</td>
</tr>
<tr>
<td>7.14</td>
<td>Rdy-Ack symmetric communication latency for passing-pointer solution evaluated with the ping-pong micro-benchmarks</td>
<td>151</td>
</tr>
<tr>
<td>7.15</td>
<td>Rdy-Ack many-to-one communication latency for passing-pointer solution evaluated with the ping-pong micro-benchmarks</td>
<td>152</td>
</tr>
<tr>
<td>7.16</td>
<td>A 3-stage pipeline computation example</td>
<td>153</td>
</tr>
<tr>
<td>7.17</td>
<td>Speedup of the farm computation with the various run-time supports: integer values matrices with N=32/64/128 Tilera TilePro64</td>
<td>159</td>
</tr>
<tr>
<td>7.18</td>
<td>Speedup of the farm computation with the various run-time supports: float matrices with N=32/64/128 Tilera TilePro64</td>
<td>160</td>
</tr>
</tbody>
</table>
List of Tables

4.1 GC and Network Latencies ........................................... 62
4.2 Reading Operations Latencies ....................................... 63
4.3 Writing Operations Latencies ........................................ 67
4.4 Memory read latencies for core 0, depending on the cache line state, for Intel SandyBridge processors. Times in clock cycles ............. 71
4.5 Memory read latencies for core 0, on the Tilera TilePro64 architecture, with or without cache coherence. Times in clock cycles ......... 71
4.6 Memory write latencies for core 0, depending on the cache line state, for Intel SandyBridge processors. Times in clock cycles ............. 72
4.7 Memory write latencies for core 0, depending on the cache line state, for Tilera TilePro64 processors. Times in clock cycles ............. 73
6.1 PEPA steady-state resolution of a multiple-source computation graph ......................................................... 112
7.1 Reading and writing activities on the VTG fields .................. 148
7.2 Reading and Writing Operations Latencies in Tilera TilePro64 .... 155
List of Algorithms

5.1 Lock and unlock implementation with the test-and-set instruction . . 86
5.2 Lock and unlock implementation with LL/SC instructions ......... 88
5.3 Global barrier algorithm with mutual exclusion ................... 90
5.4 Use of a memory fence instruction inside the unlock operation to ensure the correct memory ordering (Total Store Ordering) ........ 92
5.5 Locking queue implementation .................................. 93
5.6 ⊥-based lock-free queue implementation ....................... 95
7.1 Pseudo-code of Stage 2 in a 3-stage pipeline computation .......... 155
Part I

Introduction
CHAPTER 1

Introduction

Continuous advances in microprocessor design have been reflected in high-performance computing platforms that increasingly rely on multicores as a basic building block. Off-the-shelf multicores, also called Chip MultiProcessors (CMPs) in the research world, are at the moment commonly built with a relatively small number of cores (4 to 12). However, the trend in hardware technology is clear: the “Moore law” is expected to be applied to the number of cores and researchers expect architectures with 128 to 1024 cores on a single chip in few years [67, 96], up to the point that the term many-cores has been introduced, to indicate the large amount of core per chip of some solutions. All that accompanied by a corresponding evolution of on-chip interconnection networks and by increasingly complex memory hierarchy and more scalable cache coherence solutions. Some examples of current highly parallel multicore platforms are the Tilera TilePro64 [19] processor (64 cores), the IBM PowerEN [50] (16 4-way Simultaneous Multithreading cores each processor for a maximum of 64 cores and 256 thread in 4-chip configurations), the AMD Opteron [34] with 48 cores in a single machine, the IBM Power7 [64] servers, with up to 32 8-core 4-way SMT processors per machine, for a total of 1024 threads, or the Intel Xeon Phi, an accelerator composed of 60 4-way SMT cores [62].

This “revolution” only further emphasizes the gap that exists between parallel architectures and parallel programming maturity. “The aggressive goal of the parallel revolution is to make it as easy to write programs that are as efficient, portable, and correct (and that scale as the number of cores per microprocessor increases biennially) as it has been to write programs for sequential computers.” [16].

The historical difficulties that characterize parallel programming continue to influence the choices related to the development of parallel programming environments: the achievement of the trade-off “productivity-performance-programmability” is still characterized by the contrast between the search for high performance through low-
level tools and the use of high-level languages for easy programmability. Applications should be designed at the highest level by means of formalisms and tools that are fully independent from the machine architecture and from the mechanisms at the process level.

Our research group, starting from the experiences on distributed and shared memory multiprocessor \cite{109, 11, 10, 21}, proposes a structured parallelism approach to bridge this gap. Structured parallelism paradigms aim to provide standard and effective rules for composing parallel computations in a machine independent manner. Cost models are defined for performance evaluation and prediction, and are a fundamental tool for reducing the complexity in parallel software design.

General results for multiprocessor architectures are therefore valid for CMPs. At the same time, however, there are some important features and capabilities that are not available in multiprocessor and must be further investigated. The number and complexity of cores, the interconnection network among cores and towards the outer memory, cache hierarchies and cache coherence solutions make each CMP different. Regarding the parallel programming, this forces the programmer to write specific low-level code to reach good performance. Currently, the most common situation consists in developing parallel applications directly at the process level through programming languages or extensions of existing sequential languages (e.g., OpenMP \cite{38}, Cilk \cite{71}) or through libraries (e.g., MPI \cite{102}, Intel TBB \cite{92}, Skandium \cite{72}, Fast Flow \cite{15}). These approaches are independent of the underlying architecture, but have not sufficient expressive power to support high-level development of complex applications and “performance portability”. In addition, a cost model is still missing, that is there is no way to predict the behaviour in terms of performance of your program until it is run on a specific platform, thus making different parallelizations of a program incomparable in a formal or generalizable way, but only by execution times.

1.1 The Cache Coherence Problem in the Many-Core Era

The cache coherence problem arises from the possibility that more than one cache of the system may maintain a copy of the same memory location. If different processors transfer into their cache the same data, it is necessary to ensure that copies remain consistent with each other and against the copy in the memory hierarchy. Current solutions consist in preventing incoherence through the use of hardware mechanisms that ensures that each cache holds the current value of a memory location. These mechanisms (commonly called cache coherence protocols) rely on hardware or automatic cache coherence techniques that allow programmers to develop programs without taking into account this problem. In fact, no explicit coherence operations must be inserted in the program.
In a generic cache coherence protocol each line in a cache has a state associated, that represents the availability and use of that cache line inside the system. From a logical point of view, the state is global (i.e. the same for each cache of the architecture). A read/write operation to a cache line changes its state, and may prompt some communications between the caches to ensure that none of them holds a stale value. To do that two main techniques can be used:

- **invalidation** when a line is modified on a private cache, all the other caches remove (if present) the old value;

- **update** each modification of a cache line is communicated (broadcasted) to all other caches.

In general shared memory programming models an automatic cache coherence protocol is proven to offer the best performance [86]. When the problem of cache coherence was introduced, several studies tried to introduce and compare hardware-based and software-based cache coherence protocols. There are several works that highlight how cache coherence may be ensured at a software-level to obtain performance improvements [3]. In fact, it is well known that standard coherence protocols are inefficient for certain data communication patterns (e.g., producer-consumer), and these inefficiencies will be amplified by the increased core number and the complex memory hierarchies.

This makes it again necessary to study the cache coherence mechanisms with particular attention to the disadvantages of automatic techniques compared to, what we call, an algorithm-dependent approach a particular kind of software (or non-automatic) cache coherence solution characterized by explicit cache management strategies, which are specific to the algorithm to be executed.

### 1.2 Parallel Paradigms and Cache Coherence

Understanding the impact of automatic cache coherence solutions in parallel program performances is too complex if studied independently of the parallel program characteristics. Following a structured parallelism approach, our methodology to attack these problems is based on two interrelated issues: structured parallelism paradigms and cost models (or performance models).

Performance prediction of a program, although widely studied, is still an open problem in the research community. Cost models in the world of parallel programming are usually proposed to asymptotically study algorithms, like PRAM [17], and BSP [106] models. A first step towards a more “detailed” model is LogP [36], and its successive enhancements. However all these models are kept as simple as possible to let programmers easily compare algorithms. We are not interested in this kind of model. We are looking for a more realistic model that takes into account every important property of the parallel architecture and of the parallel program. Unfortunately, as
of today, there does not exist a way to precisely estimate the completion time of a general program on current architectures, mainly because of their complexity and dynamicity. The idea of specific performance models for parallel pattern is not really new, as it was introduced in $P^3L$ [17]. These, however, modeled the performance of the implementations by taking the sequential code as a “black box”, with specific, immutable, characteristics.

We extend the original concepts by defining a performance cost model in association with a simplified view of the concrete architecture, an abstract architecture or abstract model for cache coherent architectures. The abstract model is a simplified view of the concrete architecture able to describe the essential performance properties and abstract from all the others that are useless. It aims to throw away details belonging to different concrete architectures and emphasizes all the most important and general ones (e.g., the actions required by cache coherence protocols to maintain data consistent).

A cost model, associated to the abstract architecture, has to sum up all the features of the concrete architecture (e.g., how cache coherency state affects the memory and cache access latencies), the inter-process communication run-time support (e.g., in order to evaluate communication performances of an algorithm-dependent solution) and the impact of the parallel application (e.g., showing the possible correlation between parallel paradigms and cache coherence). Further, we strongly advocated that a cost model should be easy to use and conceptually simple to understand.

The aim is to use cost models to perform optimizations for parallel applications and to study and compare the different implementations of cache coherent solutions for each pattern.

1.3 Our Starting Point

Our research group has a quite long history in structured parallel programming, starting with the $P^3L$ skeleton language in 1992, and culminating with ASSIST [109] in the last years. We never, however, really focused our effort in multicore and shared memory architectures in general. Our experiments with FastFlow [15] demonstrated the need, and the possibility, of multicore-specific optimizations in a skeleton-based library. A skeleton library, however, does not allow us to fully exploit the benefits of structured parallel programming, because it does not (entirely) allow code restructuring and transformations. The long-term project of our research group is ASSISTANT [21], the extension and adaptation of ASSIST for the current world of parallel computing, composed of multicores, pervasive grids and clouds. Many of the principles introduced in ASSIST are inherited and extended, in order to provide a significant leap forward in the world of multicore-oriented parallel programming.

Respecting the basic ASSIST principles, a parallel program will be described as a generic graph of stream-connected parallel modules. Each module will be constituted by a parallel pattern, and the programmer will be able to write the algorithm code
by means of the most used sequential languages (C/C++, Java, Matlab, and so on). A first step toward multicore technologies has been taken with Daniele Buono’s PhD thesis \cite{buono2008} where we start targeting multicore architectures, showing the feasibility of the cost model approach, by defining an architectural model for a specific many-core architecture (the Tilera TilePro64), and applying it on well known parallel pattern implementations to evaluate specific memory-related optimizations introduced in the thesis.

1.4 Contribution of the Thesis

With this thesis we start from the experiences gained so far in our research group and we approach the cache coherence problem with our methodology. Notably, we define an architectural cost model for cache coherent CMP-based architectures and apply it to evaluate specific cache coherence-related optimizations for well known parallel paradigms.

The fundamental contributions are the following:

- The definition of an abstract model for cache coherent CMP-based architectures, which is able to summarize the characteristics of the state-of-the-art of multicore technologies (e.g., memory and cache hierarchy) in relation to possible cache coherence solutions. The model provides a first result toward the evaluation of the impact of automatic cache coherence on parallel program performances, by analytically defining the base memory and cache access latencies of reading and writing operations in terms of the coherency protocol adopted.

- An extensive study on the parallel paradigm implementations with the focus on the identification of specific cache coherence patterns in order to evaluate how and when coherency protocols are effectively used and with which effects on parallel program performances. In this way, we are able to define specific optimizations by exploiting the knowledge of the structure of each parallelism scheme, such as the interactions between the parallel modules and their data access patterns.

- A queuing network-based model for cache coherent CMP-based architectures that, starting from the abstract model of the architecture, shows how standard automatic cache coherence affects the under load memory and cache access latencies. Notably, by combining the abstract model with well known performance modelling techniques, such as queuing models and stochastic process algebras (i.e., PEPA \cite{pepa}), we provide an application- and architecture-dependent cost model to predict structured parallel application performances. This cost model is fundamental in the definition of the parallel paradigms run-time support, showing for example how a specific mapping strategy can improve performances by minimizing the under load latencies.
1. INTRODUCTION

• An optimized run-time support for structured parallel applications and a demonstration of the use of the cost model to compare different solutions based on automatic or non-automatic cache coherence, lock-free and based on interprocessor communications. Notably, we show that with the non-automatic and lock-free solutions based on interprocessor communications we are able to reduce the number of memory accesses, cache transfers and synchronizations, and increasing computation parallelism with respect to the use of the automatic cache coherence alternative. Finally, the implementations of these solutions on Tilera TilePro64 processors confirm the results estimated by the cost model.

1.5 Outline of the Thesis

The thesis is organized in three main parts:

Part I: Introduction in which we introduce the reader to the state of the art of cache coherence in CMP-based architectures and to our methodology based on structured parallel programming and performance models, establishing the basis to understand the second part. Notably, we have:

• Chapter 2 that reviews the current state of cache coherence solutions and of parallel programming for multicores. We describe the features of current architectures and the evolution trend that is likely to be followed. Then, a brief overview of hardware-based and software-based cache coherence solutions and the existing evaluations of these approaches. Finally we introduce the current tools for programming CMPs.

• Chapter 3 is focused on the introduction of the methodology adopted in the second part of this thesis. We introduce the reader to the Structured Parallel Programming. After that, we introduce the conceptual framework of ASSIST and its pervasive evolution ASSISTANT and the general approach to performance models.

Part II: Applying Our Methodology to the Cache Coherence Problem where we define an application- and architecture-dependent cost model to predict structured parallel applications performances and to compare automatic vs non-automatic cache coherence solutions. Notably, we have:

• Chapter 4 in which we define an abstract model for cache coherence CMP-based architectures in order to analytically define base memory access latency in terms of the cache coherence protocol. We also show interesting benchmark results that validate the abstract model results.
1.6. CURRENT PUBLICATIONS BY THE AUTHOR

- **Chapter 5** analyzes different parallel paradigms implementations (i.e., shared-memory vs message-passing) to understand the recognize eventually cache coherence patterns in order to study the possible optimizations for the support of each parallelism form.

- **Chapter 6** starting from the abstract model defined in Chapter 4 and the analysis of Chapter 5, defines an analytic cost model for under load memory latencies based on queuing model and discusses the effects of specific parallel program mapping with different cache coherence solutions. Finally, a brief introduction to the PEPA process algebra shows to the reader how this tool could be useful for performance evaluation of computation graphs and under load memory latency.

### Part III: Evaluation of the Proposed Methodology

In which we show the result obtained from the experiments used to evaluate the considerations done in the previous part of the thesis. Notably, we have:

- **Chapter 7** Proposes different implementations for the support of parallel paradigms, in order to compare automatic and non-automatic solutions with respect to a lock-free solution based on interprocessor communications. The comparison is done by using the cost model defined in the previous part of the thesis and by some experiments executed on the Tilera TilePro64 processor, which constitute an interesting example of a chip multiprocessor, given its 64 cores and the use of innovative solutions for the interconnection network and the cache coherence mechanisms.

- **Chapter 8** present the conclusions of the thesis and some reflections on the results obtained.

### 1.6 Current Publications by the Author

The following represents the publications that I worked on during my Ph.D. research:


• Daniele Buono, Marco Danelutto, Silvia Lametti and Massimo Torquati. Parallel Patterns for General Purpose Many-Core. Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on, Pages 131-139, ISSN 1066-6192, DOI: 10.1109/PDP.2013.27.
CHAPTER 2

Background

This chapter presents the current state of the art in multicore computing, both from the hardware and the software point of view, with a special attention to the cache coherence problem.

We give first an overview of the architectures that we consider most interesting and promising in the industrial and research landscape. After that, we will try to summarize the state of art of the cache coherence solutions, in terms of automatic and non-automatic mechanisms and some results about the evaluations and the comparison of these solutions.

This chapter concludes with a brief overview on the parallel programming tools both for multiprocessor and CMP-based architectures.

2.1 CMP Architectures

In this section, we present a brief state of the art of CMPs from the hardware perspective. While processors with a few (4 to 12) cores are common today, this number is projected to grow. Because of this rapid evolution and of the consequent open issues, we present a small set of architectures in order to understand what are common choices in the various features of CMPs and that will characterize future architectures.

We consider the following four examples, all represented in figure 2.1, as representative for our study:

- AMD Opteron 6100 [34]
- IBM PowerEN [50]
2. BACKGROUND

(a) The AMD Opteron 6100

(b) The IBM PowerEN

(c) The Intel SCC

(d) The Tilera TilePro64

Figure 2.1: Examples of current CMP-based architectures


2.1. CMP ARCHITECTURES

- Intel Xeon scc \(^{13}\)
- Tilera TilePro64 \(^{19}\)

and we have been focusing on the following characteristics, which are summarized in table 2.1.
<table>
<thead>
<tr>
<th>Characteristics</th>
<th>AMD Opteron 6100</th>
<th>IBM PowerEN</th>
<th>Intel SCC</th>
<th>Tilera TilePro</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of processors</strong></td>
<td>2 six-cores processors</td>
<td>16 cores</td>
<td>24 dual-cores tiles</td>
<td>100 cores</td>
</tr>
<tr>
<td><strong>Processor architecture</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out-of-order execution</td>
<td>in-order execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3-way superscalar</td>
<td>4-way superscalar</td>
<td>2-way superscalar</td>
<td>3-way superscalar</td>
</tr>
<tr>
<td></td>
<td>6 execution unit (floating point and SIMD)</td>
<td>2 execution unit</td>
<td>floating point execution unit</td>
<td>3 execution unit</td>
</tr>
<tr>
<td><strong>Instruction Level Parallelism</strong></td>
<td>no hardware multithreading</td>
<td>4-way SMT</td>
<td>no hardware multithreading</td>
<td>VLIW instruction set</td>
</tr>
<tr>
<td><strong>Memory and Cache hierarchy</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory bandwidth and organization</td>
<td>UMA with a single memory interface directly connected to the L3</td>
<td>UMA with 2 memory interfaces connected by a bus, possible hierarchical composition (NUMA of SMPs)</td>
<td>NUMA with 4 memory interfaces</td>
<td>NUMA with 4 memory interfaces</td>
</tr>
<tr>
<td>Caches</td>
<td>separated L2 for data and instructions</td>
<td>4 L2 caches each shared by 4 cores</td>
<td>unified L2 for each core</td>
<td>unified L2 for each core with directory</td>
</tr>
<tr>
<td>Cache Coherence</td>
<td>snoopy-based inside the chip (among L2 and L3) and directory-based among different chips</td>
<td>snoopy-based</td>
<td>no hardware cache coherence</td>
<td>directory-based</td>
</tr>
<tr>
<td>interconnection network</td>
<td>crossover among L2 caches and L3, partial crossover for multiprocessor configurations</td>
<td>crossover among groups of 4 cores and the L2, bus connection between all L2 and the memory, crossover for multiprocessor configurations</td>
<td>4 by 6 two-dimensional mesh</td>
<td>10 by 10 two-dimensional mesh</td>
</tr>
<tr>
<td>Atomic operations and synchronizations</td>
<td>“hardware” passive wait for threads</td>
<td>message-passing</td>
<td>possibility of inter-core communication provided by the mesh</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: CMPs characterization
Number of processor  In CMPs this is represented by a constant value, which
depends on the chip size and on how all components are organized and balanced
inside the chip. The lower complexity of Intel SCC and Tilera TilePro64 cores
allow a larger number of cores inside a single chip. We expect two parallel lines
of development in the future: complex cores with a relatively low parallelism and
simpler cores with a higher parallelism.

Processor architecture  The core complexity is relative to the domain of each
processor: general purpose servers, like the AMD Opteron, have to maintain good
performance on sequential code thus leading producers to maintain high complexity
as it was in high performance uniprocessors. When the target is different, simple
cores obviously allow more cores per chip and lower power consumption.
An important aspect is how Instruction Level Parallelism (ILP) is extracted. A
clean and elegant solution has been offered by the so-called Very Long Instruction
Word (VLIW) model, which makes intensive use of compile-time optimizations. In
VLIW terminology, a long instruction is the implementation of a stream element.
The main feature of a VLIW long instruction is that it is composed of n independent
instructions, of which at most one may be a branch instruction provided that the
target instruction is the first of a distinct long instruction. Another adopted solution
is Simultaneous MultiThreading (SMT) that permits multiple independent threads
of execution. Hardware multithreading, especially in the form of SMT, is having
more and more success and the Opteron is a representative example also because of
the combination of SMT with out-of-order execution.

Memory and Cache hierarchy  A very important aspect in CMPs is its memory
hierarchy. A first level cache per core is the norm, in fact it is simply regarded as
part of the core. The main challenge ahead is scaling to chip with thousands of
cores. Thus there is a need to minimize the number of misses out of a core.
Furthermore, another open issue is how to keep good memory bandwidth and latency.
Because of the pin-count problem, in CMPs it is impossible have each core with
its own interface to memory, like in multiprocessors. In all the examples memory
interfaces are considerably less than the number of processors, and it will remain
the trend for future architectures. Regarding the memory organization, we usually
have SMP architectures, but with complex interconnections and multiple interfaces
the architectures usually become NUMA.

Interconnection Network  The interconnection network is another important
characteristic because it defines how cores exchange data among each other and
the cost of each communication. Simple interconnections like crossbar and bus are
typically used in CMPs with small number of cores and in a hierarchical configu-
ration of the network. Crossbar is an all-to-all connection that keeps the latency
constant and minimizes conflicts, but it can be applied to a limited number of nodes
because of the number of links (quadratic in the number of nodes). Bus and rings (uni-dimensional mesh) are characterized by a latency proportional to the number of nodes, so are not adopted in highly parallel CMPs. The current complexity of the cores is probably the reason for which fat tree and butterfly interconnections are still missing. The meshes used by Tilera TilePro64 and Intel SCC are simpler and take less space. However we expect that, as the number of cores will increase, low latency interconnections like mesh, fat tree and butterfly will be also implemented on-chip [97].

Atomic operations and synchronizations Current CMPs usually support atomic memory operations for core synchronization. An inter-core synchronization could be much faster and allow fine grain parallelism. Many works in the literature study this problem [105]. PowerEN use optimized synchronization mechanisms that stop execution a thread until a specified memory location is written by other threads. Tilera TilePro64 instead, directly use the mesh to exchange data, without accessing memory. However, it appears that message passing can give a significant latency reduction in inter-core communication and both Tilera TilePro64 (with instructions between registers) and Intel SCC provide this solution.

2.2 Cache Coherence

The cache coherence problem arises from the possibility that more than one cache of the system may maintain a copy of the same memory location. If different processors transfer into their cache the same cache line, it is necessary to ensure that copies remain consistent with each other and against the copy in the memory hierarchy. Cache coherence schemes include protocols and policies that prevent the existence of copies of writable data in more than one cache at the same time.

2.2.1 Automatic Cache Coherence

Off-the-shelf systems usually implement the cache coherence techniques entirely at the hardware level. Two main techniques, called automatic cache coherence techniques, are used:

- **invalidation**, in which when a line is modified on a private cache, all the other caches remove (if present) the old value;

- **update**, in which each modification of a cache line is communicated (broadcasted) to all other caches.

Cache Coherence Protocols

In all the systems which use the automatic techniques, a proper protocol must exist in order to perform the required actions atomically.
2.2. CACHE COHERENCE

In a generic cache coherence protocol each line in a cache has a state associated with it, along with the tag and data, which indicates the disposition of the line. The cache policy is defined by the cache block state transition diagram, which is a finite state machine specifying how the state of a line changes. While only cache lines that are actually in cache have state information, logically, all blocks that are not resident in the cache can be viewed as being in either a special “not present” state or in the “invalid” state.

The MSI protocol The MSI protocol is a basic invalidation-based protocol for write-back caches. The protocol uses the three states (i.e. Modified, Shared and Invalid) required for any write-back cache in order to distinguish valid cache lines that are unmodified from those that are modified (dirty). Before a shared or invalid line can be written and placed in the modified state, all the other potential copies must be invalidated.

The MESI protocol The MESI protocol (known also as Illinois protocol due to its development at the University of Illinois at Urbana-Champaign [87]) is a widely used cache coherence protocol. It is the most common protocol which supports write-back cache. The MESI protocol adds an Exclusive state to reduce the traffic caused by writes of cache lines that only exist in one cache. This new state indicates an intermediate level of binding between shared and modified:

- unlike the shared state, the cache can perform a write and move to the modified state without further requests;
- it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block from another cache.

The MO(E)SI protocol The MOSI protocol is another extension of the basic MSI cache coherence protocol. It adds the Owned state. A cache line in the owned state holds the most recent, correct copy of the data. The MOESI protocol, introduced in [103], encompasses all of the possible states commonly used in other protocols.

The Dragon protocol The Dragon protocol is a basic update-based protocol for write-back caches. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system [74]. The Dragon protocol consists of four states (i.e. Exclusive-clean, Shared-clean, Shared-modified and Modified) which are comparable to MESI’s states.
Multilevel Cache Hierarchies  As seen in Section 2.1, many systems use private first (L1) and second level (L2) caches. Multilevel cache hierarchies would seem to complicate coherence. Notably, we distinguished inclusive vs victim 2-level cache:

- with inclusive cache: cache lines in L1 are a proper subset of cache lines currently allocated in L2;
- while, in victim cache: it is possible that a cache line in L1 is not currently allocated in L2.

L1 and L2 are independent processing units, thus independent cache controllers exist for each hierarchy level. However, the distinction inclusive vs victim is relevant mainly for the actual form of cache coherence control: in victim cache, coherence control is actually decentralized between L1 and L2, while in inclusive cache, coherence control can actually be implemented in L2 only.

In the inclusive case, automatic solutions also need to preserve the inclusion property, which requires the following:

- if a memory block is in the L1 cache, then it must also be present in the L2 cache; in other words, the contents of the L2 cache must be a subset of the contents of the L2 cache;
- if the cache line is in an owned state (e.g., modified in MESI, owned in MOSI, shared-modified in Dragon) in the L1 cache, then it must also be marked modified in the L2 cache.

Keep the inclusion property allows for advantages in performance, even in systems where a level of cache hierarchy is shared (as shown in [51]), as it avoids, as also said before, making unnecessary communications.

In [18] are presented some techniques used to maintain the inclusion property.

**Snoopy vs Directory**

Automatic cache coherence techniques allow programmers to develop programs without taking into account the cache coherence problem. In fact no explicit coherence operations must be inserted in the program.

Two main classes of architectural solutions have been developed for automatic caching:

- *Snoopy-based*, in which the cache coherence protocols exploits a real centralization point (e.g., a single bus) and the associated snooping and broadcast operations;
- *Directory-based*, which implements cache coherence protocols using shared data (e.g., in main memory); this solution is adopted in highly parallel systems, with powerful interconnection networks.
In Snoopy-based systems each of devices connected to the interconnection network can observe every network transaction, e.g., every read or write request. When a processor issues a request to its cache, the cache controller examines the state of the cache and takes suitable action, which may include generating network transactions to access memory or other caches. Coherence is maintained by having all cache controllers “snoop” on the network and monitor the transactions from other nodes. An interesting issue that arises with the use of automatic techniques based on snooping is the cache-to-cache sharing: when more than one cache has a valid copy of the line, it is necessary to have a selection algorithm to determine which of these should provide the data. The MESIF protocol was developed by Intel [46] to solve this problem. To do this the protocol use a new state *Forward*, which indicates that the cache should act as a designated responder for any requests for the corresponding cache line.

In highly parallel systems interconnection structures are used to allow greater scalability than what can be achieved with linear latency networks. This choice is also reflected in the decision to integrate automatic cache coherence mechanisms that scale better than the solutions based on Snoopy bus or similar interconnection networks.

Scalable cache coherence is typically based on the concept of a directory. Since the state of a cache line in the caches can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place, called directory. This way, memory operations can be sent only to the set of nodes that are actually interested. In the following, we will use the term *node* to represent a processing element (processor or core), its private cache subsystem and cache controller(s). Notably, in a director-based solution we can apply the following definitions to a shared cache line:

- the *home* node: is the node in whose main memory the cache line is allocated, or more generally, is the node in charge of controlling a given partition of cache lines;
- the *local* (or requestor) node: is the node that issues a request for the cache line;
- the *owner* node: is the node that holds the valid copy of the line in its local cache, and must supply the data when needed.

Depending on where the directory is maintained, we can distinguish two main schemes:

- *memory-based* schemes, that store the directory information about all cached copies at the home node of the cache line;
2. BACKGROUND

- cache-based schemes, where the information about cached copies is not all contained at the home but is distributed among the copies themselves; the home simply contains a pointer to one cached copy of the block; each cached copy then contains a pointer to the node that has the next cached copy of the block, in a distributed linked list organization.

The directory-based approach minimizes traffic in the interconnection networks, however keeping and accessing the directory have a cost in term of space (the directory must reside in a fast memory) and increased memory access latencies.

Some works in the literature have discussed alternative automatic solutions: in [44] the authors propose a MSI implementation within the network, while in [91] the authors propose a new directory scheme and a cache coherence scheme based on it for a mesh interconnection. However, the cost in terms of space is simply moved from the memory or the caches to the network switches, without a significant improvement from the performance point of view.

2.2.2 Software or non-Automatic Cache Coherence

Software managed coherence represents an alternative design point that places the burden of maintaining coherence on compilers, libraries, and runtime systems.

Software and hybrid cache coherence Software-directed cache coherence on shared memory multiprocessors has been proposed as an alternative to automatic cache coherence. Such schemes are based on self invalidations and forced writebacks of the private caches at synchronization points (some required a write-through cache instead of forced writebacks). They relied on explicitly marked synchronization points (often simply the boundaries of parallel loops). Additionally, to filter out unnecessary invalidations, these schemes relied on the programmer [29] or the compiler [29, 30] to identify what data is involved in the communication and, thus, must be written back and invalidated. Unfortunately, these techniques require sophisticated analysis, and are applicable to very regular array-based or loop-based computations where static analysis can identify the communicated data or when the programmer can clearly specify sections and their data.

Other works [28] have proposed additional hardware support to assist with the actual invalidation, e.g., using per-word tags updated by software to identify cache lines that do not have to be reloaded after a synchronization point. However, they require complex program analysis to statically manage the update of the tags, obtaining good results only for programs with simple control flow and regular data accesses.

Hybrid self invalidation Self-invalidation is a technique used to invalidate cache lines locally and it has been proposed as a mechanism to reduce the amount of coher-
2.3. EXISTING EVALUATIONS OF CACHE COHERENCE SOLUTIONS

Compiler-directed schemes still rely on automatic coherence mechanism to ensure correctness, hence without obtaining a real improvement on performance with respect to “pure” automatic solutions.

**Software distributed shared memory** Several works have been focused on software and hybrid distributed shared memory systems. These approaches are mainly targeted to clusters of workstations and not only dealt with cache coherence but also provide a shared memory environment.

**Software-only directory protocol** This approach has evolved along two main directions: either a separate protocol processor is used to execute the software handlers that emulate cache coherence protocol or the handlers are executed on the compute processor. The first direction is represented by the Stanford FLASH and the Wisconsin Typhoon. Several design efforts aim at executing the software handlers on the compute processor, e.g., the MIT Alewife and the Cooperative Shared Memory. Both approaches have a bad effect on performance in highly parallel system mainly because of the small possibility of optimizations for the specific algorithm.

### 2.3 Existing Evaluations of Cache Coherence Solutions

The most common parallel performance models for multiprocessors are built for parallel algorithm designers, who are not interested in particular architectures, but look for algorithms that perform well in general. The models are therefore based on an asymptotic prediction of the performance, exactly as the complexity order is analyzed in sequential algorithms. The idea of allowing a simple and machine-independent study of the algorithm is indeed in contrast with our idea of a detailed and machine-dependent prediction, able to compare different cache coherence solutions.

Several works in the literature have tried to evaluate the overhead introduced on read and write operations by automatic cache coherence solutions. In general the problem was addressed by adopting several simplifications on the workload model, and analytically deriving the coherence overhead by using tools such as Markov chains, Generalized Timed Petri Nets or queuing networks. However, modeling the behavior of cache coherence analytically is still an open research, because of the strong implications of the program (in particular its data access pattern) on the coherence traffic.

First results on comparison of automatic and non-automatic techniques have shown that the software schemes are comparable to directory-based protocols for a wide
class of computations [4]. In this work, authors classify shared objects into classes: for example, mostly-read objects represent data written very infrequently and may be read more than once by multiple processors before a write by some processor, while frequently read-written objects represent data written frequently and also read by multiple processors between writes. Figure 2.2 shows the ratio between processor efficiency of a software scheme and that of an automatic scheme, where $f_{MR}$ represent the fraction of references to mostly-read objects. There are separate curves to represent the effect of conservative analysis of the software scheme. Automatic cache coherence significantly outperforms non-automatic solutions for the mostly-read class of data. However, if memory access conflicts can be detected accurately at compile time ($\geq 0.9$), the non-automatic schemes is competitive with the automatic scheme for most cases. The performance of classical software solutions is limited by the need to use compile-time information to predict run-time behaviour, forcing these approaches to be conservative. On the other hand, for well structured programs (many scientific programs fall under this class) non-automatic schemes are comparable and in some cases better (within 10%) than automatic schemes.

2.4 Parallel Programming on CMPs

CMPs inherit their ideas from Shared Memory Multiprocessors and for this reason and because many of these processors are also given in multiprocessor configurations, they can be programmed using all tools developed in the past for SMP architectures. However, as seen in section 2.1, CMPs have specific features, that have to be exploited.

As a consequence, a lot of new parallel programming environments emerged. How-
ever, in our opinion, many of these are very similar to previous tools and they still do not exploit all the possibilities offered by these new architectures. These approaches are independent of the underlying architecture, but all the advantages like performance portability of structured parallel programming are missing: they have not sufficient expressive power to support high-level development of complex applications and "performance portability".

The way adopted in order to have an efficient solution for a specific CMP is, typically, to downgrade to a lower level of implementation writing code strongly related to that specific platform. However, the possibilities to have performance predictions are not achievable, because a detailed cost model of the architecture is still unavailable.

In this section we briefly analyze the most important tools available for shared memory multiprocessing and some specific for CMPs. We will first present programming languages and then libraries.

### 2.4.1 Programming Languages

Parallel programming languages are usually extensions of sequential languages like C and Java, only some of them are actually new programming languages. With respect to shared memory and message passing libraries there is a sort of facility in the definition of the parallel program. However, the programmer has to define all the execution flows and the cooperation among them. For these reasons we consider these solutions a "low level" approach to parallel programming.

**OpenMP** is probably the best known for shared memory parallel programming. It provides an extension for different sequential languages: C, C++, Java and Fortran. With OpenMP the main program is run sequentially and in specified points the code is executed in parallel. The several languages supported offer to programmers good portability among different architectures. However, to each supported language corresponds a different compiler that implements the same program in different ways on the same architecture, making it impossible to predict the performances. This is an example of languages that do not exploit any specific feature of CMPs.

**Cilk** is a task parallel extension of C, which is recently commercialized as Cilk Plus by Intel (as C and C++ extension) designed for multithreaded parallel computing. As in OpenMP, there are some keywords used by the programmer to define parallel parts of the code and synchronization points. We have the same disadvantages of OpenMP, principally no exploitation of CMP specific characteristics.

**Berkley Unified Parallel C** has an important characteristic that distinguishes it from previous languages: it uses a Partitioned Global Address Spaces, so
it can be ported also on distributed memory systems. An advantage over previous solutions is related to the run-time support, which is being optimized for CMPs. It also provides some collective operations, but it is still a low-level shared memory language from the point of view of performance portability.

Erlang [111] was mainly targeted at distributed systems because it is a message-passing parallel programming language. However, recently a shared memory implementation has been developed. Also this is a low-level solution, in which the programmer has to define all the parallel application.

Go Programming Language [33] is a recently introduced object-oriented language that exhibits a C-like syntax and greatly focuses on concurrency. Parallelism is automatically achieved by using “goroutines”, functions specifically marked to be executed concurrently. Goroutines can exchange data by using asynchronous channels; thus, go highly resembles a low-level message-passing parallel language.

2.4.2 Libraries

There are many different libraries for parallel programming of CMPs and, in order to have a simple classification, we have chosen representative examples that summarize typical choices adopted. The various libraries described below are listed from low level to high level parallel programming.

Posix threads [26], usually referred to as Pthreads, is one of the lowest level libraries for parallel programming of shared memory systems. It gives to the programmer access to OS threads and basic mechanisms for synchronization. There are no distributed memory versions of the library. The definition of the parallel application is entirely delegated to the programmer and also there are no optimizations for CMPs.

MPI [102], the Message Passing Interface, allows to define a set of processes with a local environment and gives to the programmer primitives for send and receive and for some collective communications. It is also available a shared memory implementation and specific implementation for high performance interconnection networks. MPI offers good portability among different platforms, however we consider it a low level library which could be used as a sort of run-time support for an high-level parallel programming environment.

Intel Libraries [92, 83, 73]: Intel Threading Building Block (TBB) and Intel Array Building Block (ArBB) are library provided with Intel multiprocessors, while Intel RCCE is the most recently Intel library provided for the Intel SCC. TBB is a stream-parallel library, while ArBB is specialized for data-parallel programming.
However, both have no optimizations for CMPs. RCCE tries to fill this lack, offering a message-passing programming model with collective communications. Because there is no cache coherence among cores in SCC, this library is an interesting example which shows how message passing can be an effective way to avoid having to provide cache coherency in CMPs.

FastFlow \[15\] is a CMP library created by our research group. It provides a lock-free and wait-free communication channel that can be used directly by the programmer. Moreover, the library implements on top of this level a generic master-worker skeleton, which can be used to define stream-parallel and data-flow applications. We consider it an interesting approach to CMP parallel programming. However, no cost models are used, and performance tuning is still a concern of the programmer.

Skandium \[72\] is a skeleton-based high-level parallel library specifically targeted to CMPs. It offers several task and data-parallel skeletons. However, the library is based on Java threads, which make no distinction between multiprocessor and multicore architectures. Furthermore, performance can partially be limited by the use of the Java language.

2.5 Summary

In this chapter we introduced the reader to the world of chip multiprocessors. We presented the current state-of-the-art in CMPs and the most feasible future trend, in which the amount of processors per chip will increase to a point that programming these chips will become even more difficult.

We introduced the reader to the cache coherence problem, describing off-the-shelf automatic solutions with respect to the software or non-automatic approach. These solutions have been proposed by the research world to alleviate performance degradation problems of automatic cache coherence in specific data access patterns. However, comparing and evaluating the effect of cache coherence solutions is still an open research.

Given the level of detail of certain works, it was not really feasible to introduce and describe all of them here. We therefore decided, for the sake of readability, to keep this chapter at an introductory level.

Finally, we presented the most important programming tools used for generic multiprocessors and CMP-based multiprocessors. Notably, we highlighted the most important problem in software development, which is the absence of complete environments especially targeted at multicores, able to fully exploit these architectures without manual intervention of the programmer.
CHAPTER 3

Our Methodology: Programming and Cost Models

In this chapter we summarize the basic features characterizing the methodology proposed by our research group and used in this thesis. Our idea starts from the definition of a parallel application defined as a directed graph whose nodes are co-operating (parallel) modules. By “solving the computation graph” we are able to understand which modules represent bottlenecks for the application performance. The next step consists in providing a functionally equivalent parallel computation for each bottleneck, without modifying the semantics and the logical interfaces with respect to the sequential version. In order to evaluate and to compare some alternative versions of the parallel transformation, we use proper performance metrics according to a cost model. This performance model is parametrically dependent on the application characteristics and on the target parallel architecture(s).

Over the last decade a significant research effort has been invested in studying and developing new programming models and frameworks for parallel computations. A big challenge has been the definition of approaches which render parallel programming easy to use, improving the reuse of existing components to create different and more complex systems and providing performance portability without requiring intensive interventions of the programmer to tune the performance of each application. Portable parallel applications should be able to be used on different computing platforms without modifying the program source code, and the porting phase should also be able to exploit in the best way possible the physical aspects of the underlying architecture.

As it is well established by the scientific community [41, 108, 101], a high-level approach is the only solution to performance portability of parallel applications. The structured parallel programming is probably the most interesting class of high-level parallel models. This approach allows the programmer to define the parallel program having in mind only an abstract high-level view of the application, while all
the most critical implementation choices (e.g., the parallelism degree, task granularity, process/data mapping on corresponding processing elements) are left to the programming tools and run-time support.

In the first part of this chapter, we introduce the general concepts of a structured programming model. Then, we describe our methodology, that finds its roots on the ASSIST programming language [109] developed by our research group some years ago, comparing it with respect to other classic structured programming models. Notably, we introduce the workflow of our parallel compiler, highlighting the open research points that will be partially addressed in this thesis.

3.1 Structured Parallel Programming

Structured parallel programming is probably the most powerful class of high-level parallel models. It started with the concept of algorithmic skeletons defined by Cole [32] and has been successfully applied in a range of parallel environments, starting from clusters [39] and shared memory machines [72], to grid [13], cloud and pervasive environments [21]. Two of the most important points of structured parallel programming are the ability to automatically create different parallel implementations starting from the high-level description, and the parametric nature of the produced code, that is able to run with different parallelism degrees. These points are the basic building blocks to ensure performance portability on the various architectures. Structured parallel programming also allows composability: a parallel code can be mixed with others, such that an application can be described as a collection of parallel kernels, instead of a single, large parallel code. Composability also allows reuse: the “kernel” code can be reused inside different programs with no modifications.

Our research group history in structured parallel programming is quite long, starting with the $P^3 L$ skeleton language in 1992 [17], and culminating with ASSIST in recent years. These projects proposed many interesting developments for parallel programming: parallel code restructuring [9], to better exploit the composition of parallel kernels; efficient fault tolerance [20], and dynamic reconfigurations up to self-adaptive programs [77], i.e. programs that are able to exploit performance portability dynamically, at run-time, to better fit dynamic environments such as grids or clouds. Finally, we also extended the concept of High Performance Computing to Grid computing [13] and lately to Pervasive Grids. We never, however, really focused our efforts in multi-core and shared memory architectures in general. Our experiments with FastFlow [12] demonstrated the need, and the possibility, of multicore-specific optimizations.

The structured parallel programming methodology is based on the concept of parallelism forms, also called parallelism paradigms or parallel patterns. Parallel paradigms are schemes of parallel computations that recur in the realization of many real-life algorithms and applications. They exhibit the following features:
• they are characterized by constraints in the parallel computation structure;
• they have a precise semantics;
• their behavior can be predicted through a suitable performance model;
• they can be composed to form complex graph computations.

We can characterize two broad categories of parallel paradigms: *stream parallelism* and *data parallelism*.

### 3.1.1 Stream Parallelism

Parallelism forms belonging to this class are able to improve the throughput of a computation in the case in which a large sequence (possibly of unlimited length) of input elements is defined (i.e., stream-based computation). For example, the execution of a specific computation is applied to a stream of images or video frames represented as matrices. The existence of a large sequence of input elements is a necessary precondition in order to apply these parallelization techniques, on the contrary no performance enhancements can be obtained if we consider a single or a limited set of input elements. Parallelism schemes that follow this assumption are the *task-farm* and *pipeline*.

**Pipeline** In a pipeline computation the sequential code is divided in multiple pieces executed concurrently. The application of the pipeline paradigm requires some knowledge of the form of the sequential computation, that is the sequential computation must be expressed (or rewritten) as the composition of $n$ functions:

$$F(x) = F_n(F_{n-1}(...F_2(F_1(x))...))$$

In this case, a pipeline parallelization consists in a set of (at most) $n$ entities $\{S_1, ..., S_n\}$, called *stages*, each executing one (or more) of the $n$ functions. Each stage $S_i$ will receive each input element and will compute its function $F_i$ on it. The output of each entity is sent to the next one, respecting the function ordering (i.e., the output of $F_i$ is sent to $F_{i+1}$), so that the output of the last stage (i.e., $F_n$) correspond to $F(x)$, as depicted in Figure 3.1.

![Figure 3.1: Pipeline parallelism form](image-url)
30 3. OUR METHODOLOGY: PROGRAMMING AND COST MODELS

**Task-Farm** Task-farm is a stream-parallel scheme based on the replication of a pure function among a set of identical *workers*, without knowing the internal structure of the function itself. Figure 3.2 shows the farm internal structure as a computation graph. An *emitter* module, according to a certain scheduling strategy, distributes each input stream value to a worker. The general objective is to *balance* the workers’ loads, in order to exploit best their processing capabilities. A possible scheduling strategy is the round-robin one, i.e. circular distribution. However, this strategy is not able to assure load balancing if the calculation time of a worker has a significantly high variance, especially in the case in which it depends on the input data values. For this reason an *on-demand* approach is much more effective. Basically, its implementation is based on the availability of workers to accept a new input task. The *collector* module is essentially an output interface which is responsible for transmitting onto the output streams the results which are received nondeterministically from the workers, possibly (but not necessarily) applying an ordering strategy. From the performance viewpoint a task-farm scheme has the main advantage of reducing the mean service time of the computation.

![Figure 3.2: Farm parallelism form](image)

**3.1.2 Data Parallelism**

Data-parallel computation is characterized by *partitioning* (and/or replication) of data structures and function *replication*, so that distinct, functionally identical modules (workers) are able to apply the same operations to distinct data partitions in parallel. In this scheme, an *input* module provides the distribution of each input element among the set of workers according to proper collective communications: *scatter* for sending distinct partition of the input to distinct workers or *multicast*
3.1. STRUCTURED PARALLEL PROGRAMMING

for sending the same input. Collection of the worker results is achieved by the output module exploiting a gather operation: the collector receives worker results and builds a unique data structure, such as a vector or a matrix of elements.

The data-parallel paradigm is able to reduce the computation latency for a single input element but, in the case of a large sequence of input tasks, it can also improve the throughput of the computation by reducing the mean service time.

![Data-parallel with stencil parallelism form](image)

Figure 3.3: Data-parallel with stencil parallelism form

In a data-parallel scheme each worker applies a sequential elaboration on its own data. In order to apply this function, a worker may require to access data contained in other worker partitions, according to the particular data dependencies imposed by the computation semantics.

In this case we speak about **stencil-based** computations (Figure 3.3, where a stencil is a data dependence pattern implemented by information exchanges between different workers. A Stencil can be:

- **Static fixed**, if the dependencies are defined at compile-time and remain the same throughout the computation;
- **Static variable**, if the dependencies are defined at compile-time but changes during the computation;
- **Dynamic**, if the dependencies are defined at run-time, depending on data structure values.

A very special, but sometimes possible, data-parallel scheme is the so-called **map**, in which workers are fully independent, that is each of them operates on its own local data only, without any communication during the execution, as shown in Figure 3.4.

Another interesting scheme is the **reduce** pattern which is applicable every time we have a computation of the form:

\[ y = x_1 \oplus x_2 \oplus ... \oplus x_k \]
where the result is a single value obtained by applying a function ⊕ to all the elements of the input data structure. To ensure correctness ⊕ also has to satisfy the associative property.

This can be easily parallelizable by partitioning the data structure in \( nw \) workers (parametric but limited by \( k \)), each performing a “local” reduce on their partition, followed by a “global” reduce of the results of each worker. The global reduce can be executed in several ways (even in parallel) by one (or more) of the workers. Notable examples that naturally fit this paradigm are many vector- or matrix-based operations such as the maximum and minimum of a vector, the dot product and many others.

Finally, many algorithms can be defined as a composition of two steps, in which at first a function is applied to all the elements, and then the results are merged by using some reduction function. We can straightforwardly represent this class of
3.2 Parallelization Methodology and Cost Model

The algorithmic skeletons defined by Cole [32] represent the first approach to structured parallel programming. He proposed a quite small set of skeletons (Fixed Degree Divide & Conquer, Iterative Combination, Cluster and Task Queue) obtained both by the isolation of particular algorithmic techniques, and by an analysis of patterns that could perform well on the initial target machine (a Transputer). From his idea, however, many researchers focused on finding general yet effective patterns that could be promoted to skeleton. Among the others, P3L provided pipeline, task farm, map and reduce, plus geometric, loop and tree as data-parallel with stencils [17]; SKELib [40] offered only stream-based skeletons (farm and pipe), while Lithium [8] supported pipe, map, farm and reduce. Once stabilized, the set of used skeleton basically remained the same over the years: Skandium [72], one of the newest skeleton frameworks, implements seq, pipe, farm, for, while, map, d&c, fork, not introducing new patterns with respect to the first works.

All these systems employ the very same concepts introduced by Cole: the user just writes a skeletal specification, such that a program is basically a composition of skeletons. The majority of environments define three kinds of skeletons: data parallel, task parallel and sequential skeletons. Sequential skeletons encapsulate functions written in a sequential language and are not considered for parallel execution. The others provide typical task and data parallel patterns.

The initial specification provided by the programmer may then be subjected to a cost-driven transformation process with the aim of improving the performance of the parallel program. Such transformation is done by means of semantic-preserving rewriting rules. A rich set of rewriting rules and cost models for various skeletons have been developed in the past [14, 49, 100].

Despite several advantages of skeletons, a strong evolution of structured parallel programming beyond such models is needed. In addition to the capability of expressing some typical parallel schemes, we need a larger degree of flexibility in expressing parallel and distributed program structures: we cannot afford to produce a skeleton for any data-parallel pattern, nor force the programmer to write applications choosing in a small set of well studied patterns. Although very interesting, pattern composable patterns is still limited becoming a limitation when describing large, complex applications. Finally we recognize that parallel patterns cannot efficiently capture every parallel application: dynamic stencils, for example, cannot be modeled by a skeleton; we need to allow some kind of cooperation with different parallel environments so that skeleton-based patterns can cooperate with pre-existing, or manually optimized, parallel code.
ASSIST  An interesting and effective approach to overcome the limitations of skeleton environments has been introduced by our research group with ASSIST [109] (A Software development System based upon Integrated Skeleton Technology). In ASSIST an application is described by a generic graph of modules connected by streams. This alone allows some basic stream-parallel paradigm such as pipelining, but at the same time permits very complex behaviors and loops among the modules that compose the application. Parallelism is also available inside the nodes, because each module represents a parallel pattern.

ASSIST employs a novel approach to data-parallel by describing the parallel application (and its stencil) at the minimum partitioning level. This approach, called “Virtual Processor” (VP) generalizes the class of data-parallel algorithms and allows the programmer to describe with a single formalism a generic data-parallel algorithms with a static stencil. Lastly, a module is not forced to be implemented as a parallel pattern: the programmer may provide its specific, hand-made implementation of a parallel module. This effectively solves the cases in which a parallel paradigm cannot be applied.

The main idea of the VP approach is to describe the application by using a set of VPs, i.e. virtual entities that, like processors, own a partition of the data structure, execute the calculation on it and exchange data with others. Notably, the programmer defines the stencil at the minimum partitioning level by using this abstraction, while the parallel environment is in charge of analyzing it to determine if it represents one of the basic, well studied paradigms, or a new, “unknown” stencil. In any case, a proper worker partitioning must be established, so that the Virtual Processors becomes Real Processors, perhaps with a different stencil, and perform the computation. The input data are partitioned, so that each VP “owns” a single element, and is in charge of computing it by following the owner compute rule.

This way of describing data-parallel algorithms is indeed very powerful, because it explicitly defines the stencil at the element level. From this, an intelligent compiler can apply optimizations like stencil transformation [78], and optimize the stencil with respect to the execution environment.

ASSISTANT  The long-term project of our research group is ASSISTANT, the extension and adaptation of ASSIST for the current world of parallel computing, composed of multicores, pervasive grids and clouds. Many of the principles introduced in ASSIST are inherited and extended, in order to provide a significant leap forward in the world of CMP-oriented parallel programming.

Respecting the basic ASSIST principles, a parallel program will be described as a generic graph of stream-connected parallel modules. Each module will be constituted by one of the previously mentioned parallel patterns, or by a VP-based description in the case of a data-parallel. Programming models based on libraries are considered unsuitable for achieving the desired level of programmability and
performance portability: our environment will need an intelligent source-to-source parallel compiler, able to analyze the module-based description to determine the possible parallel implementations, evaluate them for the target machine and, finally, produce the source code of a low-level parallel program.

Our experience in parallel programming also indicated that there are many cases in which performance portability is not completely achievable at compile-time: the cost model may be not detailed enough to accurately fit the combination application implementation-target architecture, or some model parameters may be unpredictable (because of both the architecture and the algorithm) so that mere compiler-based performance portability becomes ineffective. To handle all these important cases, it is also mandatory to support adaptivity, by means of efficient run-time reconfigurations, in addition to static optimizations.

In addition, to better allow performance portability and adaptivity, we believe it is necessary to allow the programmer to explicitly define different patterns for each module. In this way, if multiple parallel patterns, with different performance characteristics, are known by the programmer, we further increase the possibilities of our compiler. This approach, which has been introduced in our works with pervasive grids, remains consistent with the programming model. The compiler will then use its cost model to select (and optimize) the best among the whole set of implementations.

Figure 3.6 represents the specific “compilation workflow” that we have in mind for ASSISTANT, where, for the reasons describe before, it is evident the importance of the cost model, which affects basically every step of the workflow.

Notably, starting from computation graph, the cost model is used to recognize which
modules of the application graph represents a bottleneck. In order to eliminate, or at least to reduce the effects of, the bottlenecks, we try to parallelize each bottleneck module according to some parallelism paradigms. The result is a computation graph, functionally equivalent to the initial one, in which some nodes are transformed through the internal structured parallelization defined by the selected parallelism form.

In the second step the compiler determines, for each parallelized module, the best parallel paradigm and its implementation, considering both the user-provided and the automatically derived transformations.

At the end of this step the bottleneck detection algorithm is run again, considering the expected implementation of each module and their specific cost model. If new bottlenecks are found, the steps for the bottleneck detection and their parallelization are executed iteratively to further refine the parallel implementation.

The compiler then generates the low-level parallel source code, to be compiled using a generic compiler. The resulting application, however, is also enriched with monitoring tools and other possible parallel implementations, so that, by continuously monitoring and applying the cost model, the program self-adapts to better match the running environment and guarantee the best possible performance.

3.2.1 Performance Modeling with Queueing Networks

As already introduced, in our programming environment, a parallel program is expressed as a graph, whose nodes are co-operating (parallel) modules. Consider the example of graph computation shown in Figure 3.7. It is a directed acyclic graph (DAG) in which nodes correspond to modules and arcs to interactions, possible by means of streams or single values. Moreover, each module can be described by a parallel pattern, and therefore able to exploit stream-parallel or data-parallel parallelism, depending on the chosen implementation. Performance metrics are associated to nodes (notably, internal calculation time and/or ideal service time) and
to arcs (e.g., asynchrony degree, communication latency, in some cases a probability of utilization).

The methodology proposed in [77] is aimed to completely model the performance at any level, analyzing both the internal behavior of a single module, and the performance of the entire computation graph, by providing a performance modeling approach expressed in terms of fundamental results in the area of Queueing Theory and Queueing Networks. In this way we will be able to formalize important issues related to:

- how to evaluate the performance of a graph computation starting from the knowledge of the performance of each module;
- how to evaluate the effective performance of a module based on the ideal performance behavior of all the modules of the computation graph;
- how to detect bottlenecks in a computation graph, that is modules that seriously limit the performance of the entire application.

In this section we will just introduce the concept, needed to intuitively understand the ideas and how the model works; the interested reader can refer to [77] for more specific details.

The basic idea consists in modeling the performance of a module M (either sequential or internally parallel) by abstracting its behavior as a queueing system, as shown in Figure 3.8. This scheme is a logical one, not necessarily corresponding to the real structure of the computation. From the performance evaluation viewpoint, the logical scheme reduces the analysis complexity and makes it possible to obtain an approximate evaluation, which is quite acceptable provided that the mathematical and stochastic assumptions are validated.

The queueing systems can be analytically defined knowing the following characteristics:

- **Service discipline**: if not otherwise stated, the FIFO discipline is assumed.
- **Queue size**, which is the number of elements available for storing the clients requests.
• The probability distribution of the random variable service time $t_s$, which represents the ideal time needed to serve a customer, that is the time passed between the beginning of the executions on two consecutive stream elements. We denote with $T_S$ the mean value and $\sigma_S$ the variance of this random variable.

• The probability distribution of the random variable inter-arrival time $t_A$, which indicates the time interval between two consecutive arrivals of requests, with mean value $T_A$ and variance $\sigma_A$.

• The probability distribution of a random variable inter-departure time $t_P$, which represents the time between two successive result departures from the module, with mean value $T_P$ and variance $\sigma_P$.

The queue utilization factor, or equivalently the server utilization factor, is defined as

$$\rho = \frac{T_S}{T_A}$$

It is a very meaningful parameter for performance evaluation, expressing a global, average measure of the congestion degree, or traffic intensity, of the requests to the server. When $\rho > 1$, the server represents a bottleneck with respect to the client(s) requests.

Each computation module can be abstracted as a queueing system and the computation graph can be described as a network of queues, where the departures of some nodes establish the arrivals of others. Figure 3.9 shows the example module graph used before and its queueing network representation.

From the network topology viewpoint queueing networks can be categorized into two broad classes namely open queueing networks and closed queueing networks. In an open queueing network a possibly infinite number of requests are generated by source nodes, go through several nodes or even revisit a particular node more than once and finally leave the system. On the other hand, in a closed queueing network requests neither arrive at nor depart from the system, but a fixed number
of requests continuously circulate through the nodes of the network. In our case, the graph of modules depict an open queueing network, given the presence of infinite streams. For the sake of simplicity, our approach will be limited to *acyclic computation graphs*, where each task follows a certain path, passing through each module at most once. With this simplification, we are able to analyze the performance of this kind of graph in a completely independent way with respect to the internal behavior of each computation module, which may implement any parallelism paradigm. The only parameter required is the average value of the ideal service time of each module. Anyway, the case of *client-server computations with request-reply behavior* will be studied in Chapter 6 to model multiprocessor systems.

**Acyclic graph computations**

The evaluation methodology, derived from common queueing theory, consists in two interrelated phases: *transient* and *steady-state* analysis. As said before, when $\rho > 1$, the server represents a bottleneck with respect to the clients requests. There is a transient period during which more requests of the same client can be delivered to the queue where they are buffered, so the client behavior is relatively independent from the server one. However, in the steady-state behavior the mean number of queued elements tends to grow indefinitely. Thus, if $\rho > 1$, on the average the server is not able to satisfy the client requests.

In real systems and computations, because the queue size is of finite length, when $\rho > 1$ in the steady-state behavior on the average the client is temporarily blocked each time it tries to send a new request, thus in the steady-state behavior the client request rate adapts to the server service rate. For this reason the situation $\rho > 1$ is a transient one. However, for our purposes, it is meaningful, because it is the condition we have to check in order to discover the possible existence of a bottleneck starting from the definition of the computation modules.

When $\rho < 1$, the server is not a bottleneck, and the distinction between the transient phase and the steady-state phase has no substantial impact on the average performance measures.

While, when the server is a bottleneck a non-null transient phase exists before reaching the steady-state behavior. Once the behavior is stable, the mean interarrival time becomes equal to the mean service time of the server, thus the mean service time of any client is increased with respect to the initial values, i.e. with respect to the ideal service time.

For performance evaluation of acyclic graph computations, the condition $\rho = 1$ denotes a “limit situation” in which, on the average, the clients are not delayed, although considerable fluctuations around the mean values exist. From a mathematical point of view, in an acyclic graph computation, the condition $\rho = 1 - \delta$, with $\delta > 0$ arbitrarily small, is sufficient for a steady-state behavior without bottlenecks.
In other words, in acyclic graph computations, $\rho$ values less than one correspond to the bottleneck elimination and, if close to one, to the optimal server utilization. A further parallelization of the server, thus a further $T_S$ reduction, is not beneficial for the client and implies lower server efficiency. In the design of parallel computations expressed by acyclic graphs, where possible, we try to eliminate all the bottlenecks by imposing utilization factor values less than one and very close to one, so achieving the best server efficiency.

### 3.2.2 Performance evaluations of modules and graph computations

Once a bottleneck is found, the module will be parallelized according to the “compilation workflow” presented in this section. In order to do that, we need to evaluate the ideal and effective bandwidths of acyclic graph computations, and, by using this evaluations, we can determine the optimal degree of parallelism.

The following results are used in the evaluation of performances for sub-classes of graph computations. The first theorem evaluate the interarrival time of the requests generated from a client to a set of servers.

**Theorem 3.2.1** (Interarrival time during the transient phase). Assume a graph composed of a client node $C$ and a set of server nodes $S_1, \ldots, S_n$. Let $p_i$ be the probability that a request from $C$ is directed to $S_i$, where $\sum_{i=1}^{n} p_i = 1$. During the transient phase, the interarrival time $T_{A_i}$ to each server $S_i$ is given by:

$$T_{A_i} = \frac{T_C}{p_i}$$

where, $T_C$ is the interdeparture time of $C$ towards any server.

The next theorem evaluate the interarrival time of the requests generated from a set of clients to a single server.

**Theorem 3.2.2** (Total interarrival time during the transient phase). If a server node $S$ has $n$ multiple clients each one with an initial inter-departure time $T_{P_i}$ to $S$, during the transient phase the total inter-arrival time to $S$ is given by:

$$T_A = \frac{1}{\sum_{i=1}^{n} \frac{1}{T_{P_i}}}$$

These theorems are valid in the transient behavior of the computation. Thus, they are valid also in the steady-state behavior only if there are no bottlenecks, as discussed in the previous section. Otherwise, all interarrival and interdeparture times must be re-evaluated in order to have a correct evaluation of the steady-state behavior performances. The interested reader can consult [77] for a complete...
Let now consider a generic acyclic graph computation \( \Sigma \), consisting of a module, or subsystem, \( \Sigma_1 \) having one or more input streams with interarrival time \( T_A \), as depicted in Figure 3.10.

**Ideal and effective service time of \( \Sigma_1 \)** The ideal service time of \( \Sigma_1 \) \( (T_{\Sigma-id}) \) is evaluated by considering it as an “isolated” system. That is, the ideal service time does not depend on the interarrival rate \( 1/T_A \). The inverse of the ideal service time measures the *offered bandwidth* of \( \Sigma_1 \), while the interarrival rate represents the *requested bandwidth* of \( \Sigma \) to \( \Sigma_1 \).

If \( \Sigma_1 \) is a single module, it is characterized by an average internal calculation time \( T_{calc} \), which gives a first idea of the ideal service time. The ideal service time is also evaluated as a function of the latency of communications \( L_{com} \) performed by \( \Sigma_1 \) towards the external world. For example, we can have

\[
T_{\Sigma-id} = T_S = \max(T_{calc}, L_{com})
\]

if the calculation and communications phases can be overlapped, or

\[
T_{\Sigma-id} = T_S = T_{calc} + L_{com}
\]

If \( \Sigma_1 \) is a n-parallelization (n is the parallelism degree) of a sequential subsystem with ideal service time \( T \), the ideal service time of \( \Sigma_1 \) is given by

\[
T_{\Sigma-id} = T_S = \frac{T}{n}
\]

Because \( \Sigma_1 \) belongs to a system \( \Sigma \), its effective service time is given by its interdeparture time, which can be evaluated according to the definition of \( \rho \) as follows:

\[
T_P = \begin{cases} 
    T_A & \rho < 1 \\
    T_S & \rho \geq 1 
\end{cases}
\]

or, equivalently,

\[
T_P = \max(T_A, T_S)
\]
The efficiency of Σ₁, is then evaluated considering the complex system as follows

\[ \varepsilon_{\Sigma_1} = \frac{T_{\Sigma-id}}{T_\Sigma} = \frac{T_S}{\max(T_A, T_S)} \]

Therefore, we have that the relative efficiency of a module/subsystem, belonging to a more complex system, is equal to its utilization factor if it is not a bottleneck, otherwise it is equal to one:

\[ \varepsilon_{\Sigma_1} = \begin{cases} \rho_{\Sigma_1} & \rho_{\Sigma_1} < 1 \\ 1 & \rho_{\Sigma_1} \geq 1 \end{cases} \]

Thus, the evaluation of the whole system Σ can be derived as follows:

\[ T_\Sigma = T_{\Sigma_1} = T_P \quad T_{\Sigma-id} = T_A \]

Therefore, the relative efficiency of the entire system is:

\[ \varepsilon = \frac{T_{\Sigma-id}}{T_\Sigma} = \frac{T_A}{T_P} \]

This means that the system is able to achieve the ideal bandwidth, and the maximum efficiency, if it does not contain bottlenecks, that is \( T_P = T_A \), otherwise \( T_P > T_A \).

These results are applicable to any acyclic system where a module/subsystem \( \Sigma_{\text{in}} \) generates the stream with interdeparture time \( T_A \).

**Optimal parallelism degree** Suppose that \( \Sigma_1 \) represents a bottleneck in an acyclic graph computation. The optimal parallelism degree for its parallelization can be evaluated as

\[ n_{opt} = \left\lceil \frac{T_S}{T_A} \right\rceil \]

Actually, this value represents the potentially optimal parallelism degree, which means that the bottleneck can be eliminated, provided that a parallelism paradigm exists able to actually exploit this parallelism degree.

If the optimal parallelization is feasible, then the effective service time of \( \Sigma_1 \) is:

\[ T_{S_{opt}} = T_A \]

and the relative efficiency:

\[ \varepsilon_1 = \rho_{\Sigma_1} = \frac{T_S}{n_{opt}} = \frac{T_S}{T_A} \]

\[ \varepsilon_1 = \rho_{\Sigma_1} = \frac{T_S}{n_{opt}} = \frac{T_S}{T_A} \]
3.2. PARALLELIZATION METHODOLOGY AND COST MODEL

Otherwise, the parallelized $\Sigma_1$ is still a bottleneck. If $n_0 < n_{opt}$ is the best parallelism degree achievable, the effective service time is

$$T_{S}^{n_0} = \frac{T_S}{n_0} > T_A$$

Regarding the motivations for which the optimal parallelization cannot be feasible, a trivial one is due to the insufficient number of available processing nodes in the target architecture.

Anyway, $n_{opt}$ depends on $T_{calc}$ and $L_{com}$. These values are initially evaluated on a single processing node of the target architecture. Using $n$ processing nodes, some parameters affecting $T_{calc}$ and $L_{com}$ might assume different values with respect to the “sequential” case of $n = 1$.

In general, a parallel program uses additional functionalities and data structures with respect to the sequential module, e.g., for synchronizations and cooperations between the modules that implements the parallelism paradigm.

Moreover, communication latencies and memory access latencies depend on the number of active processing nodes and their processing times, as we will deeply analyze in Chapter 6.

Therefore, in principle the parallelism degree should be re-evaluated in terms of the “parallel-version” values of $T_{calc}$ and $L_{com}$.

3.2.3 Parallelism forms and cost models

As introduced in Section 3.1, each parallelism form is characterized by a specific semantics and its behavior, in terms of data partitioning or replication and function replication, is well-defined.

Moreover, each parallel paradigm is characterized by a specific cost model.

**Pipeline** Consider the pipeline computation

$$F(x) = F_n(F_{n-1}(...F_2(F_1(x))...))$$

As in the previous Section, let $T_{calc}$ the average calculation time of the whole function $F(x)$:

$$T_{calc} = \sum_{i=1}^{n} T_{F_i}$$

In the most general case, stages might be unbalanced in terms of internal calculation time and/or of communication latency. Therefore, one stage is the (heaviest) bottleneck for the whole computation. The cost model is derived by applying the general theory of the previous section.

Let $b$ be the index of the bottleneck stage, and let $T_b = T_{max}$ denote its ideal and effective service time. If $T_{i-id}$ denotes the ideal service time of stage $i$, with
44 3. OUR METHODOLOGY: PROGRAMMING AND COST MODELS

![Diagram of task farm pipeline](image)

Figure 3.11: Stream-oriented pipeline modeling of task farm

\[ i = 1, \ldots, n, \] in the steady-state behavior all stages have effective service time and efficiency equals to:

\[ T_i = T_{\text{max}} \quad i = 1, \ldots, n \]

and

\[ \varepsilon_i = \frac{T_{\text{1-id}}}{T_{\text{max}}} = \begin{cases} \rho_i & i \neq b \\ 1 & i = b \end{cases} \]

If the stream is generated by the first stage, for the whole computation we have:

\[ T^n_{\Sigma-id} = T_{1-id} \quad T^n_{\Sigma} = T_{\text{max}} \quad \varepsilon^n_{\Sigma} = \frac{T_{1-id}}{T_{\text{max}}} \]

While, if the stream is generated externally with interarrival time \( T_A \):

\[ T^n_{\Sigma-id} = T_{\text{max}} \quad T^n_{\Sigma} = \max(T_{\text{max}}, T_A) \]

\[ \varepsilon^n_{\Sigma} = \begin{cases} 1 & T_{\text{max}} \geq T_A \\ \rho_{\Sigma} = \frac{T_{\text{max}}}{T_A} & T_{\text{max}} < T_A \end{cases} \]

Assuming that all stages have the same \( L_{\text{com}} \), the latency of the pipeline computation is

\[ L^n = \sum_{i=1}^{n} (L_i + L_{\text{com}}) \]

which is greater than the sequential module latency.

**Task Farm** In the task farm paradigm a pipeline effect exists among emitter, set of workers, and collector, as summarized in Figure 3.11.
3.2. PARALLELIZATION METHODOLOGY AND COST MODEL

If $T_{\text{calc}}$ is the calculation time of function $F$ and $T_{\text{com-W}}$ is the (non-overlapped) latency of communications performed by a worker, the service time of each worker is

$$T_W = T_{\text{calc}} + T_{\text{com-W}}$$

Let $T_E$ and $T_C$ be the emitter and collector service times, and $T_A$ be the interarrival time to farm $\Sigma$.

The optimal number of workers is given by the general theory:

$$n_{opt} = \left\lceil \frac{T_{\text{calc}}}{T_A} \right\rceil$$

This is always true if the emitter is not a bottleneck, that is

$$T_A \geq T_E$$

which is verified in the large majority of cases, e.g., when $T_E = L_{\text{com}}$. Farm is not able to exploit the optimal parallelism degree when:

$$T_A < L_{\text{com}}$$

Therefore, the emitter interdeparture time is

$$T_{P_E} = \max(T_A, T_E)$$

Since the workers are load balanced, the probability that an input stream element is sent to any worker is constant and equal to $1/n$. Therefore for 3.2.1, we have:

$$T_{A_i} = n \cdot \max(T_A, T_E) \quad i = 1, \ldots, n$$

and

$$T_{P_i} = \max(T_W, n \cdot \max(T_A, T_E)) \quad i = 1, \ldots, n$$

For 3.2.2, the collector interarrival rate is given by:

$$\frac{1}{T_{\Sigma}} = \sum_{i=1}^{n} \frac{1}{T_{P_i}} = \min\left(\frac{n}{T_W}, \frac{n}{n \cdot \max(T_A, T_E)}\right) = \min\left(\frac{n}{T_W}, \frac{1}{\max(T_A, T_E)}\right)$$

which represents also the effective farm bandwidth, if the collector is not bottleneck. Thus, the farm service time is:

$$T_{\Sigma} = \max\left(\frac{T_W}{n}, \max(T_A, T_E)\right)$$

The best number of workers is the $n$ value that maximizes the bandwidth, which in the most frequent case in which emitter is not a bottleneck is:

$$n_{opt} = \left\lceil \frac{T_W}{T_A} \right\rceil$$

As in the pipeline, the latency is greater than the sequential latency:

$$L_{\text{farm}} \sim T_{\text{calc}} + L_E + L_{\text{com-W}} + L_C$$
Data-parallel  More complex with respect to the task farm case is the definition of performance models for data-parallel schemes. As we have seen in the example provided in Section 3.1 for data-parallel computations many forms exist and, for each form, some variants are possible, e.g. with or without replicated data, with or without data communications between workers, and so on. In this section, we provide a general description of a performance modeling for data-parallel programs, which needs to be instantiated to real cases.

We consider data-parallel programs in which a composite input state (vectors and/or matrices) is partitioned/replicated among a set of workers which apply a function $F$ on each element of its assigned partition for a certain number of iterations. The function evaluation is a sequential computation that can feature statically known data dependencies: for instance the evaluation of $F$ on the i-th element of an array can depend on the values of the nearest neighbors $i-1$ and $i+1$. Such dependencies can vary between different iterations of the a data-parallel program (variable stencil) or they can be the same for all iterations (fixed stencil). In our model at each iteration $i$, all data dependencies are related to the element values computed at the end of the previous iteration $i-1$. In some cases, at the end of the computation, a dedicated process performs the gathering of the local results of each worker, filling an output data-structure.

In the above description emerge three different phases of a data-parallel program: (i) the distribution of the input data-structures; (ii) the execution phase composed of a set of iterations performed by each worker in parallel; (iii) the collection of worker results. The performance model of data-parallel programs is defined in terms of the service time and the computation latency of these three phases.

We can evaluate the latency of a data-parallel computation as

$$L_{dp} = L_{IN} + \sum_{i=1}^{n_{it}} T_{\text{iter}}(i) + L_{OUT}$$

where $L_{IN}$ is the computation latency for completing the distribution of the input data-structures of a task executed by the INPUT module, and $L_{OUT}$ is the latency for completing the results collection executed by the OUTPUT module. The middle term of the equation ($T_{\text{iter}}(i)$) indicates the computation time of each worker, where $n_{it}$ represents the number of iterations executed by each worker.

For each worker, the computation time per iteration consists of a calculation phase, in which the sequential computation is applied to all the elements of its partition (of dimension $g$), and in a communication phase, in which a portion of the local data is transmitted to other workers according to the data-dependencies imposed by the computation semantics. Therefore we have:

$$T_{\text{iter}}(i) = gT_{F} + T_{\text{comm}}(i)$$

where, $T_{F}$ indicates the calculation time per element and $T_{\text{comm}}$ indicates the communication time required for exchanging data with other workers of the computation.
If a data-parallel scheme operates in a stream-based computation, its ideal service time is given, as in the farm case, by the following expression:

\[ T_{dp} = \max \left( T_{IN}, \sum_{i=1}^{n_{st}} T_{iter}(i), T_{OUT} \right) \]

where, for each stage, the corresponding modules service times are used.

### 3.2.4 Evaluating the model parameters

In this section we reached a set of equations to model the service time and the latency of the various parallelism forms which are architecture independent. The impact of the underlying architecture is captured by a set of parameters that need to be estimated, using information derived from the specific target architecture and from the algorithm.

In general, to evaluate these parameters we can use any of the three main techniques of performance evaluation: measurement, simulation and analytical modeling. All these techniques play equally important roles in performance studies, because each one has its own advantages and disadvantages, that basically consist in a proper mix of precision and cost. We cannot really say that one technique is always better than another, as it usually depends on what we are evaluating. When a single technique cannot be effectively applied, a mix of the three is used to evaluate the performance of the system.

**Modeling communications latencies** The communications latencies \( L_{com} \) is a key parameter in our cost models. It measures the latency for completing communication primitives used by the modules of the applications to perform interactions (i.e., synchronizations and communications). For example, in the message-passing implementation model it measure the latency of communication primitives \textit{send} and \textit{receive}.

\( L_{com} \) captures a large number of characteristics of the underlying architecture, like

- shared vs distributed memory,
- interconnection structures,
- memory hierarchies, caching strategies (e.g., cache coherence protocols interactions),
- features of CPUs, coprocessors, and other processing units,
- process scheduling,

and so on.

In order to take into account all fundamental issues in the cost model definition, we
utilize a typical concept in computer science: \textit{abstract architecture}.

An abstract architecture is a simplified view of the concrete target architecture able to describe the essential performance properties and abstract from all the others that are useless. It aims to throw away details belonging to different concrete architectures and emphasizes all the most important and general ones. An abstract architecture for shared memory architectures could be the one in Figure 3.12 wherein there exist many processing nodes (PEs) as processes connected to the main memory through the interconnection structure.

A cost model is associated to the abstract architecture. This cost model has to sum up all the features of the concrete architecture, the interprocess communication run-time support and the impact of the parallel application. Further, we strongly advocated that a cost model should be easy to use and conceptually simple to understand.

For example, let us consider the typical module behavior consisting of the phases “\textit{receive} – \textit{compute} – \textit{send}”. For each input stream element, $T_{\text{calc}}$ is the compute phase service time, while $T_{\text{com}}$ evaluates the delay introduced by receive and send communication primitives. In the simplest design approach, the three phases are executed in a strictly sequential order. Thus, $T_{\text{com}}$ is just the latency of the communication phase ($L_{\text{com}}$), which corresponds to the sum of the send and receive latencies:

$$T_{\text{com}} = L_{\text{com}} = T_{\text{send}} + T_{\text{receive}}$$

According to the interprocess communication run-time schemes, for example, in message-passing run-time support, we are able to define the the latency of send and receive primitives as:

$$T_{\text{send}}(M) = T_{\text{setup-send}} + M \cdot T_{\text{trasm-send}}$$

$$T_{\text{receive}}(M) = T_{\text{setup-receive}} + M \cdot T_{\text{trasm-receive}}$$

where $M$ is the message length, $T_{\text{setup}}$ is the latency of all run-time support actions except the message copy (sender-receiver synchronization, buffer of “target
variables” management, low-level scheduling), and $T_{\text{transm}}$ is the latency for copying one word of the message.

**Evaluating the sequential time** The evaluation of $T_{\text{calc}}$ is the only part that requires the knowledge of the application algorithm. Because of isolation, we only need to estimate the execution time of the sequential source code in the target architecture. This is surely an easier problem with respect to evaluating the performance of a parallel application. Several works aimed to solve this problem exists in literature [58, 61], so that we can safely consider this a solved, or at least solvable, problem that do not require further studies in this thesis. Anyway, for the sake of simplicity, in our thesis we use a very simple methodology based on the actual execution of the sequential program on our target machine. Notably, we use the following execution time model:

$$T_{\text{calc}} = CPU_{\text{execution}} \cdot \text{clock cycles} + CPU_{\text{execution}} \cdot \text{clock cycles} \cdot Memory_{\text{access}} \cdot \text{clock cycles}$$

That essentially separate the CPU time and the memory hierarchy latencies. Despite its simplicity, this is actually used in many performance evaluation works and is believed to model the behavior of a program with sufficient precision.

**Base and under-load memory access latencies** It is important to notice that for the evaluation of both communication latencies and sequential time we need to use two different values for the memory latency:

- the under-load memory access latency, which represents the access latency in a parallel program, with multiple PEs issuing memory requests concurrently, that can be predicted by means of specific architecture-based models, as we will studied in Chapter 6.

- the base memory latency, that is the memory latency without any congestion effect, depends only on the specific architecture; notably, in Chapter 4 we will provide a complete definition of base memory latency starting from the definition of an abstract architecture able to model different cache coherence solutions.

### 3.3 Summary

In this chapter we summarized the basic features characterizing the methodology proposed by our research group and used in this thesis. Starting from the definition of a parallel application defined as a graph of co-operating (parallel) modules,
We “solve the computation graph” in order to understand which modules represent bottlenecks for the application performance. The next step consists in providing a functionally equivalent parallel computation for each bottleneck, without modifying the semantics and the logical interfaces with respect to the sequential version. In order to evaluate and to compare some alternative versions of the parallel transformation, we use proper performance metrics according to a cost model. This performance model is parametrically dependent on the application characteristics and on the target parallel architecture(s) and derived starting from the abstract description of the architecture.
Part II

Applying Our Methodology to the Cache Coherence Problem
Modelling Cache Coherent Architectures

Off-the-shelf architectures offer different solutions to deal with the cache coherence problem. As we saw in Section 2.2, we can distinguish between invalidation-based and update-based coherency protocols and also between snoopy-based and directory-based implementations. Moreover, each platform defines a specific set of actions performed in order to maintain the consistency of data, namely the coherency protocol.

However, we can analyze the most important aspects of a cache coherent system by reasoning on an abstract model for these architectures.

The model will help us to understand when and how the coherency protocol is applied and give us a first idea on the differences in terms of performance of the different implementations used in current architectures.

4.1 Shared Memory Organization in CMP-based Architectures

In a generic multiprocessor architecture, we recognize the memory organization according to the relative “distance” between the Processing Elements (PEs) and the Shared Memory (M), which characterize the shared memory access latency.

Traditionally, we distinguish two main organizations:

- **Uniform Memory Access (UMA)**, often called **Symmetric MultiProcessor (SMP)**, where PEs are “equidistant” from the shared memory (from any memory modules), which means the base memory access latency is equal for any PE;

- **Non Uniform Memory Access (NUMA)**, where for each PE we distinguish between “local” and “remote” memory, which means the base memory access
latency depends on the specific PE and on the specific memory module that interact.

Figure 4.1 shows typical schemes of the two organizations. Figure 4.1a represents a SMP with \( n \) PEs and \( m \) main memory modules. In the NUMA scheme, shown in Figure 4.1b, each \( PE_i \) has a local memory \( M_i \), and the shared memory is the union of all the local memories. Each PE can address its own local memory and any other memory module. The remote accesses utilize the interconnection network, while the local accesses exploit the dedicated links. This leads to a base latency for local memory accesses (\( PE_i - M_i \)) which is lower than the remote memory access latency (\( PE_i - M_j \), with \( i \neq j \)). For this reason, NUMA organization is typically exploited allocating all the private information of process(es) mapped on \( PE_i \) in \( M_i \), in order to limit remote accesses to a subset of shared information.

With the advent of CMP-based architectures, this SMP vs NUMA classification needs to be further specialized in order to better model and exploit both single-CMP and multiple-CMP systems. In the following, the term core is a synonym for Processing Element (PE), which is the basic unit of parallelism at the architectural level.

### 4.1.1 Single-CMP

In a single-CMP architecture, as well as the main memory organization, also the cache subsystem characterizes the internal CMP organization. In fact, when a cache level is shared, the internal organization is clearly a SMP. For example, the Intel SandyBridge and the AMD Opteron processors \[95, 34\] have a third level cache shared between the various PEs, with the same base cache access latency for any PE-L3 pair.
NUMA-SMP architecture  Regarding the main memory, in current CMPs there is no internal main memory subsystem. A notable exception was the IBM Cell [65, 63]. As shown in Figure 4.2a, the main memory is typically off-chip, realized by distinct subsystems and it is interconnected through a memory controller or External Memory Interface (MINF). The increasing memory bandwidth requirements of CMPs drove to the inclusion of the memory controller inside the processor chip, and later even to the increase of the number of controllers per chip: the IBM Power7 7 and the Tilera TilePro64 both contain four MINFs on-chip. Depending on the connectivity of PEs to MINF and to the process code and data allocation, for single-MINF CMP we are again in a SMP or in a NUMA organization. For multiple-MINFs CMP, it is possible that each MINF can be, at least logically, associated to a partition of PEs. As shown in Figure 4.2b, each core is physically interconnected to all MINFs, but it is nearer to one of them. In this case, if all private information of each core partition plus some shared information are allocated on the associated shared memory partition, the architecture can be logically used as a NUMA. Otherwise, when no specific allocation strategy is planned, the NUMA characterization is related to the specific interconnection network for which the distance might depend on the communicating PE.
4. MODELLING CACHE COHERENT ARCHITECTURES

4.1.2 Multiple-CMP

The SMP vs NUMA characterization is meaningful for multiple-CMP architectures. Figure 4.3 shows how the SMP/NUMA distinction can be applied to the CMPs, according to the external interconnection and to the memory allocation strategy. EXT is the interface subsystem between the CMP MINFs and the external interconnect. The NUMA architecture is actually a NUMA-SMP architecture, since a CMP with its local external memory is a SMP or a NUMA-SMP itself, as discussed in the single-CMP section. For example, the multiprocessor configuration of both Intel SandyBridge and AMD Opteron processor fall into this characterization due to their external interconnections.

4.2 An Abstract Model for Cache Coherent Architectures

In this section we define an abstract model for a generic cache coherent CMP-based system. Figure 4.4 shows the abstract model for a generic system with a minimal memory hierarchy, composed of a shared main memory (M) and a private cache (C) for each processing element (PE).

The generic $P_E_i$ contains information about the state (e.g., exclusive or shared) of all cache lines currently allocated in $C_i$. For each memory operation, the cache management actions depend on the local state of the referred line. However, the local knowledge, managed by the Local cache Control (LC), is not sufficient for achieving cache coherence. In order to ensure that, a global knowledge of the current system-wide situation of cache allocation and coherency states is needed. Logically the global state is centralized in order to be always updated: in the abstract model this global knowledge is contained in, and managed by, a centralized Global Con-
4.2. AN ABSTRACT MODEL FOR CC-ARCHITECTURES

Shared Main Memory

Global Control of Cache Coherence
GC

PE1
LC
P1
Local Cache Control

PEi
LC
Ci
Local Cache Control

PEj
LC
Cj
Local Cache Control

PEn
LC
Cn
Local Cache Control

Cache to Cache Communications
C2C

Figure 4.4: Abstract model for a cache coherent architecture with N processing elements

trol module (GC). The actual implementation of GC will be centralized, or decentralized (statically partitioned), or distributed (dynamically partitioned/replicated), depending on the specific system architecture. The model includes also an abstract Cache-to-Cache interconnection facility, through which cache lines can be exchanged between PEs according to the protocol actions. In the actual architecture, this facility can be implemented by means of one or more interconnection network and the shared main memory M.

For example, in Snoopy-based low parallelism multiprocessors, the abstract GC centralized module is implemented just in a centralized manner. The snoopy bus arbitration logic and the (several) snooping messages strictly correspond to the existence of a centralized unit. The cache lines transfers (Cache-to-Cache communications) are implemented through the same bus. While, in Directory-based medium-high parallelism architectures, the abstract GC centralized module is decentralized through the global state partitioning. In memory-based schemes each node contains the global state directory entries corresponding to all the blocks in its local memory (home node). In cache-based schemes, the information about cached copies is distributed among the copies themselves, and the home node simply contains a pointer to one cached copy of the block; each cached copy then contains a pointer to the node that has the next cached copy of the block, in a distributed linked list organization. All these solutions has been improved to solve the cache coherence problem also in highly parallel CMP-based multiprocessor, for this reason in the following we extend our model to deal with these new solutions and systems.
4. MODELLING CACHE COHERENT ARCHITECTURES

4.2.1 A Hierarchy-based Classification

In order to deal with the complex memory hierarchies employed in CMP-based architectures and the possible use of different interconnection networks, we can extend our abstract model in the following way. First, it is reasonable to assume that each PE has one or more levels of private cache (PrC). Without any loss of generality, we still keep the Local Control (LC) associated to each PE, independently of the use of inclusive or victim caches. In both cases, LC will correspond to the set of units (e.g., cache controller(s)) used in that PE to implement the cache coherence solution adopted.

Of highest impact, from the point of view of implementation and performance of cache coherence solutions, is the presence or not of one or more modules and/or levels of shared cache (ShC). Therefore, we can distinguish between CMP with at least a shared level cache and architectures where the first shared level in the memory hierarchy directly corresponds to the main memory. The model describes a system with S modules of shared cache, where shared data are distributed among the various modules. The PE-ShC Interconnection is used by each processing element to access each module of the shared level cache. For example, the Intel Sandy Bridge processor [95] has a third level cache, inside the chip, with a number of modules equal to the number of cores, that is S = N. In this system, the three logical interconnection structures (i.e., memory to ShC (M-ShC), PE-ShC and C2C) actually correspond to the same (ring) interconnection network. In the same way, the M-PE Interconnection that connects the PEs with the main memory, can also be used for the cache-to-cache communications.

A further classification concerns the implementation of the Global Control (GC). As we said before, the actual implementation of GC will be centralized, or decentralized (statically partitioned), or distributed (dynamically partitioned/replicated). Anyway, in all these cases, we can logically associate the GC to a specific level of the memory hierarchy. Figure 4.5 shows the GC implementation when a shared level cache is present (on the left side) or not (on the right side). We can distinguish the case in which the GC is implemented at:

- the main memory level (Figures AM1a and AM2a),
- the shared level cache (Figures AM1b),
- the private cache level (Figures AM1c and AM2b)

Multiple-CMP model A typical solution used in multiple-CMP architectures is to provide a cache coherence protocol hierarchy. CMP internal caches are kept coherent by a certain protocol, called the inner protocol. Coherence across CMPs is maintained by another, and possibly different, protocol, called the outer protocol.
4.2. AN ABSTRACT MODEL FOR CC-ARCHITECTURES

Each level of the cache coherence protocol hierarchy can also adopt a different architectural solution to implement the respective protocol.

To represent these architectures with our abstract model we use a two-level hierarchy for the GC. Each of these levels maintains the specific, inner or outer, protocol global information, respectively $GC_{out}$ and $GC_{in}$. $GC_{out}$ is typically associated to the main memory level (as in CC-NUMA systems). For example, the AMD Opteron processor uses a snoopy-based solution as inner protocol implemented on the L2 caches connected to the shared L3 with a crossbar. The outer protocol for NUMA configurations is directory-based. L3 cache is directly connected to the MINF and maintains the directory of the cache line present in the entire CMP. In this case, the $GC_{out}$ is maintained at the main memory level while the $GC_{in}$ is associated to each CMP at the shared level cache.

The Intel SandyBridge processor uses a similar solution, but there are two alternative solutions as outer protocol in NUMA configurations: a sort of snoopy-based solution with multicast communications, or a directory-based solution can be used on the crossbar interconnection. The abstract model is the same, what changes are the communications performed to maintain the consistency of the data, which are analyzed in the following sections.
4.3 Base Memory Access Latencies

The abstract model introduced in the previous section, allows us to reason about the basic actions of a cache coherency protocol in order to evaluate the base memory and cache access latencies in CMP-based architectures. We distinguish between reading (Load instructions) and writing (Store instructions) operations on cache lines. In both cases, depending on the local and the global state of the referred cache line, we analyze the actions required by a generic coherency protocol.

Moreover, we refer to the different abstract models, shown in Figure 4.5 in the following way:

[AM1a] system with a shared level cache and GC associated to the main memory;
[AM1b] system with a shared level cache and GC associated to the shared level cache;
[AM1c] system with a shared level cache and GC associated to the private level cache;
[AM2a] system without a shared level cache and GC associated to the main memory;
[AM2b] system without a shared level cache and GC associated to the private level cache.

For simplicity, we start with single-CMP architectures (when not specified GC represents $GC_{in}$) and we consider a system with $S = 1$ in the case [AM1b].

4.3.1 Reading Operations

We now consider what happens when $PE_i$ execute a load instruction:

1. $P_i$ requires the data to $PrC_i$
2. $PrC_i$ checks the local state (LC)

$\langle a \rangle$ [LOCAL_READ] the local state is sufficient to satisfy the request (e.g., the data is present in the private cache in shared state)

3. $PrC_i$ replies to $P_i$ sending the data required

$\langle b \rangle$ the global state (managed by GC) has to be examined

3. $PE_i$ sends a request read_req to

[AM1a] $M$ through the interconnection network(s) (i.e., $PE - ShC$ and $M - ShC$) and read_req is forwarded to GC and $M$

[AM1b] $ShC$ through the $PE - ShC$ interconnection network and read_req is forwarded to GC and $ShC$
4.3. BASE MEMORY ACCESS LATENCIES

[AM1c],[AM2b] PrC\textsubscript{h} (note that can be \(i = h\) if \(i\) is the home node for the requested cache line) through the \(C2C\) interconnection network and read\_req is forwarded to \(GC\textsubscript{h}\) and \(PrC\textsubscript{h}\).

[AM2a] \(M\) through the \(M\text{-}PE\) interconnection network and read\_req is forwarded to \(GC\) and \(M\).

4. GC checks the global state

\(\langle a \rangle\): the global state is \textit{sufficient} to satisfy the request (i.e., the data is updated in the corresponding hierarchy level)

5. the data are sent back to \(PE\textsubscript{i}\) through the interconnection network(s)

\(\langle a \rangle\) the data are sent (typically) in parallel also to \(ShC\), to maintain the shared level updated

6. \(PE\textsubscript{i}\) updates the local state and the \(PrC\textsubscript{i}\), which forwards the data to \(P\textsubscript{i}\)

\(\langle b \rangle\): a \textit{third entity} has to be involved (e.g., \(PrC\textsubscript{j}\))

5. GC forwards read\_req through the interconnection network(s)

\(\langle a \rangle\) the third entity could be the \(ShC\) or the \(PrC\textsubscript{j}\) and, depending on the actual interconnection network(s) implementation, the communications required can be done in parallel

6. the global state is compared to the actual state of the cache line

\(\langle a \rangle\): the global state is \textit{updated}

7. the third entity notifies GC through the interconnection network(s)

8. the data are sent to \(PE\textsubscript{i}\) through the interconnection network(s) by

\[AM1a],[AM2a]\ M
\[AM1b]\ ShC
\[AM1c],[AM2b]\ PrC\textsubscript{h}

9. \(PE\textsubscript{i}\) updates the local state and the \(PrC\textsubscript{i}\), which forwards the data to \(P\textsubscript{i}\)

\(\langle b \rangle\): the global state is not updated or, anyway, the \textit{third part} involved is \textit{responsible} for sending the data

7. the data are sent to \(PE\textsubscript{i}\) directly by

\[AM1a]\ ShC through the \(PE\text{-}ShC\) interconnection network or by \(PrC\textsubscript{j}\) through the \(C2C\) interconnection network

\[AM2a]\ PrC\textsubscript{j} through the \(C2C\) interconnection network
\[ [\text{AM1b, AM1c, AM2b}] \] \( PrC_i \) through the C2C interconnection network or \( M \) through the interconnection network(s)

and, in parallel, it updates its local state and notifies GC through the interconnection network(s)

8. \( PE_i \) updates the local state and the \( PrC_i \), which forwards the data to \( P_i \)

With this analysis we can estimate the base access latency for a cache line reading operation in the various cases. Note that we did not use a specific coherency protocol, in fact the above and the next considerations remain valid both for invalidation-based and update-based protocol.

Table 4.1 simply define the costs of accessing data in the memory or cache level corresponding to the GC \( (T_{GC}) \) and the costs of the global state lookup \( (T_{lookup-GC}) \).

In Table 4.2, each line corresponds to one of the possible situations described above.

Table 4.1: GC and Network Latencies

<table>
<thead>
<tr>
<th>Operation</th>
<th>AM1a</th>
<th>AM2a</th>
<th>AM1b</th>
<th>AM1c</th>
<th>AM2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC Data Access ( T_{GC} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC State Lookup ( T_{lookup-GC} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Respectively, they estimate the latencies of condition 2 \( \langle a \rangle \), 2 \( \langle b \rangle \) 4 \( \langle a \rangle \), 2 \( \langle b \rangle \) 4 \( \langle b \rangle \) 6 \( \langle a \rangle \)

and 2 \( \langle b \rangle \) 4 \( \langle b \rangle \) 6 \( \langle b \rangle \). In the first column \( L_{read}(LCstate, GCstate, Tstate) \) represents the read latency where the local state returned by LC is \( LCstate \), the global state in GC is \( GCstate \) and the eventually third part involved state is \( Tstate \). With \( LCstate, GCstate \) and \( Tstate \) \( \in \{ M, S, E, I \} \cup - \). For simplicity, we use the MESI\[87\] states to represent the coherency states ((M)odified, (E)xclusive, (S)hared and (I)nvalid or not present). While, with \( - \) (not specified) we represent one of the possible state, which is not relevant for the latency.

Regarding the various latencies, we use \( T_M, T_{ShC}, T_{PrC} \) to represent the reading access time of \( M, ShC \) and a generic \( PrC \) respectively. With \( T_{LC} \) we mean the time needed to check the local state. Finally, we assume a system with an interconnection network that implements all the logical interconnection networks used in the abstract
4.3. BASE MEMORY ACCESS LATENCIES

models. The network latency is represented by $L_{\text{net}}(\text{n\_words})$, where $\text{n\_words}$ is the number of words in the messages ($L_{\text{net}}$ for request or notify messages of one word). For example, $L_{\text{net}}(\sigma)$ represents the communication latency of a cache line of $\sigma$ words.

Table 4.2: Reading Operations Latencies

<table>
<thead>
<tr>
<th>$L_{\text{read}}(LC_{\text{state}}, GC_{\text{state}}, T_{\text{state}})$</th>
<th>Cache Block Read Base Latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{\text{read}}(M/E/S, -, -)$</td>
<td>$T_{PrC}$</td>
</tr>
<tr>
<td>$L_{\text{read}}(I, M/E/S, -)$</td>
<td>$T_{LC} + L_{net} + L_{GC} + L_{net}(\sigma) + T_{PrC}$</td>
</tr>
<tr>
<td>$L_{\text{read}}(I, -, E)$</td>
<td>$T_{LC} + L_{net} + T_{\text{lookup-GC}}$ $+ 2L_{net} + T_{LC}$ $+ T_{GC} + L_{net}(\sigma) + T_{PrC}$</td>
</tr>
<tr>
<td>$L_{\text{read}}(I, -, M)$</td>
<td>$T_{LC} + L_{net} + T_{\text{lookup-GC}}$ $+ L_{net} + T_{M}/T_{ShC}/T_{PrC}$ $+ L_{net}(\sigma) + T_{PrC}$</td>
</tr>
</tbody>
</table>

As we said before, we consider a generic coherency protocol in order to be able to model as many as possible systems. Obviously, actual implementations represent subsets of the abstract models, coherency protocols and latencies discussed above. For example, Intel Sandy Bridge \cite{95} is represented by the abstract model [AM1b] and all examined situations may occur. Differently, Tilera TilePro64 corresponds to the [AM2b] model and, due to the writing policy adopted, condition 6 $\langle a \rangle$ never occurs. Moreover, the third entity that could be involved (condition 6 $\langle b \rangle$) is only the main memory, which sends the data to $PE_i$ through $PE_h$ adding a delay of $T_{PrC}$ to the latency.

4.3.2 Writing Operations

Estimating the latency of writing operations is also important for two main reasons:

- with a write-allocate policy, when data is not present in the $PE$, in order to
(and before) perform the actual write, similar actions to those required in the reading operations have to be carried out;

- even when, in most cases, some operations may be considered with negligible latency (e.g., because executed in parallel and overlapped with other), to ensure the correctness of a parallel program these actions may have to be serialized.

For simplicity, we now assume an invalidation-based coherency protocol. Similar consideration can be made about update-based protocols, with the obvious adjustments (e.g., write request of shared copies involves update requests to the sharers wrt invalidation requests). Therefore, consider what happens when $PE_i$ executes a store instruction:

1. $P_i$ asks for write the data to $PrC_i$
2. $PrC_i$ checks the local state (LC)
   
   \( \langle a \rangle \) the local state is sufficient to satisfy the request (e.g., the corresponding cache line is present in the private cache in exclusive or modified state)

3. $PrC_i$ replies to $P_i$ sending the outcome
   
   \( \langle b \rangle \) the global state (managed by GC) has to be examined

3. $PE_i$ sends a $rfo_{req}$ request when the data is not present in $PrC_i$ or a $write_{req}$ request in the other case to
   
   [AM1a] $M$ through the interconnection network(s) (i.e., $PE - ShC$ and $M - ShC$) and the request is forwarded to $GC$ and $M$

3. [AM1b] $ShC$ through the $PE - ShC$ interconnection network and the request is forwarded to $GC$ and $ShC$

[AM1c],[AM2b] $PrC_h$ (note that can be $i = h$ if $i$ is the home node for the requested cache line) through the $C2C$ interconnection network and the request is forwarded to $GC_h$ and $PrC_h$

[AM2a] $M$ through the $M - PE$ interconnection network and the request is forwarded to $GC$ and $M$

4. GC checks the global state
   
   \( \langle a \rangle \) the global state is sufficient to satisfy the $rfo_{req}$ request (i.e., the data is updated in the corresponding hierarchy level and it is not present in any of the other higher levels of the hierarchy)

5. the data are sent back to $PE_i$ through the interconnection network(s)
   
   ([AM1a] the data are sent (typically) in parallel also to $ShC$, to maintain the shared level updated)
6. $PE_i$ updates the local state and the $PrC_i$, which forwards the data to $P_i$

$b$ a third entity has to be involved to satisfy the $rfo_{req}$ request (e.g., $PrC_j$)

5. GC forwards $rfo_{req}$ through the interconnection network(s) in order to invalidate that copy

([AM1a] the third entity could be the $ShC$ or the $PrC_j$ and, depends on the actual interconnection network(s) implementation, the communications required can be done in parallel)

6. the global state is compared to the actual state of the cache line

$a$ the global state is updated

7. the third entity invalidates its copy and notifies GC through the interconnection network(s)

8. the data are sent to $PE_i$ through the interconnection network(s) by

[AM1a],[AM2a] $M$

[AM1b] $ShC$

[AM1c],[AM2b] $PrC_h$

9. $PE_i$ updates the local state and the $PrC_i$, which performs the actual write and sends the outcome to $P_i$

$b$ the global state is not updated or, anyway, the third part involved is responsible for sending the data

7. the data are sent to $PE_i$ directly by

[AM1a] $ShC$ through the $PE-ShC$ interconnection network or by $PrC_j$ through the $C2C$ interconnection network

[AM2a] $PrC_j$ through the C2C interconnection network

[AM1b],[AM1c],[AM2b] $PrC_j$ through the C2C interconnection network or $M$ through the interconnection network(s)

and, in parallel, it invalidates its copy and notifies GC through the interconnection network(s)

8. $PE_i$ updates the local state and the $PrC_i$, which performs the actual write and sends the outcome to $P_i$

$c$ some other entities have to be involved to satisfy a $rfo_{req}$ or a $write_{req}$ request (e.g., $PrC_j$ and $PrC_k$)

5. GC forwards the request through the interconnection network(s) in order to invalidate those copies
6a. PrC\textsubscript{j} and PrC\textsubscript{k} invalidate their copies and notify GC through the interconnection network(s)

(6b.) for rfo\_req the data are sent back to PE\textsubscript{i} through the interconnection network(s)

7. PE\textsubscript{i} updates the local state and the PrC\textsubscript{i}, which performs the actual write and sends the outcome to P\textsubscript{i}

With this analysis we can estimate the base access latency for writing operation in the various cases. As in the reading operation case, in the following table each line corresponds to one of the possible situations described above. Respectively, they estimate the latencies of condition 2\langle a\rangle, 2\langle b\rangle 4\langle a\rangle, 2\langle b\rangle 4\langle b\rangle 6\langle a\rangle, 2\langle b\rangle 4\langle b\rangle 6\langle b\rangle and 2\langle b\rangle 4\langle c\rangle.

Regarding the various latencies, we also use \( L_{inv}(n_{sh}) \) to represent the latency of \( n_{sh} \) invalidations. For example, the invalidation of a private level cache can be \( L_{inv}(1) \approx 2 \ast L_{net}(1) + T_{PrC} \). When \( n_{sh} > 1 \), invalidations latencies can overlap with each other. Anyway, in some cases, we use square brackets to mean that the inside term can be partially or totally overlapped.

In this case, we focused on invalidation-based coherency protocol, and the Intel Sandy Bridge still remains a good example of the abstract model [AM2b]. Differently, Tilera TilePro64 implements a hybrid approach in which PrC\textsubscript{i} always updates the PrC\textsubscript{h} on write operations. Whereas, PrC\textsubscript{h} still remains responsible for the invalidations. Referring to the above analysis and table, as in the reading case, condition 6\langle a\rangle never occurs, and the third entity that could be involved (condition 6\langle b\rangle) is only the main memory, which sends the data to PE\textsubscript{i} through PE\textsubscript{h} adding a delay of \( T_{PrC} \) to the latency. Finally, condition 2\langle a\rangle changes in the following way:

2. PrC\textsubscript{i} checks the local state (LC)

\( \langle a\rangle \) the local state is sufficient to satisfy the request (e.g., the corresponding cache line is present only in the private cache and in the home PE)

3. PE\textsubscript{i} sends an update\_req request to PE\textsubscript{h} through the C2C interconnection network

4. PE\textsubscript{h} updates PrC\textsubscript{h} and sends an acknowledgment to PE\textsubscript{i} through the C2C interconnection network

5. PrC\textsubscript{i} replies to P\textsubscript{i} sending the outcome

Therefore, the corresponding latency change in the following way:

\[
L_{write}(E, -, -) = L_{net} + T_{PrC} + L_{net} + T_{PrC}
\]
### 4.3. Base Memory Access Latencies

#### Table 4.3: Writing Operations Latencies

<table>
<thead>
<tr>
<th>$L_{\text{write}}(LC\text{state}, GC\text{state}, T\text{state})$</th>
<th>Write-Allocate Base Latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{\text{write}}(M/E, -, -)$</td>
<td>$T_{PrC}$</td>
</tr>
<tr>
<td>$L_{\text{write}}(I, M/E, -)$</td>
<td>$T_{LC} + L_{net} + T_{GC} + L_{net}(\sigma) + T_{PrC}$</td>
</tr>
</tbody>
</table>
| $L_{\text{write}}(I, -, E)$                                   | $T_{LC} + L_{net} + T_{\text{lookup}-GC}$  
+ $2L_{net} + T_{LC}$  
+ $T_{GC} + L_{net}(\sigma) + T_{PrC}$ |
| $L_{\text{write}}(I, -, M)$                                   | $T_{LC} + L_{net} + T_{\text{lookup}-GC}$  
+ $L_{net} + T_{M/T_{ShC}}/T_{PrC}$  
+ $L_{net}(\sigma) + T_{PrC}$ |
| $L_{\text{write}}(I, S(n_{sh}), -)$                           | $T_{LC} + L_{net} + T_{\text{lookup}-GC}$  
+ $L_{net}(\sigma) + [L_{inv}(n_{sh})]$  
+ $T_{PrC}$ |
| $L_{\text{write}}(S, S(n_{sh}), -)$                           | $T_{LC} + L_{net} + T_{\text{lookup}-GC}$  
+ $L_{inv}(n_{sh}) + [L_{net}]$  
+ $T_{PrC}$ |
4.3.3 Reading and Writing Operations in Multiple-CMP Architectures

In multiple-CMP architectures, we need to consider the case in which the outer protocol has to be involved in the reading and writing operations.

For **reading latency** we have an additional case:

2. *PrCi* checks the local state (LC)
   
   \(\{b\}\) the global state (managed by GC which acts as *GCin*) *has to be examined*

4. GC checks the global state
   
   \(\{c\}\) the global state has no information, the outer protocol has to be involved

5. GC forwards **read_req** through the interconnection network(s)
   
   to *GCout_h*, which can be
   
   - \(M_i\) if \(i\) is also the home node for the cache line
   
   (no communications are needed in [AM1a])
   
   - \(M_h\) with \(h \neq i\)

6. *GCout_h* checks the global state of the outer protocol
   
   \(\{a\}\) the global state of the outer protocol is *sufficient* to satisfy the request (i.e., the data is updated in the corresponding hierarchy level)

7. the data are sent back to *PE_i* through the interconnection network(s)

8. *PE_i* updates the local state and the *PrCi*, which forwards the data to \(P_i\)

\(\{b\}\) a third entity (e.g., *PrC_j* of CMP_h) has to be involved to

7. *GCout_h* forwards **read_req** through the interconnection network(s)

8. the data are sent back to *PE_i* through the interconnection network(s)

9. *PE_i* updates the local state and the *PrCi*, which forwards the data to \(P_i\)

Therefore, the corresponding reading latencies for conditions 2 \(\{b\}\) 4 \(\{c\}\) 6 \(\{a\}\) and 2 \(\{b\}\) 4 \(\{c\}\) 6 \(\{b\}\) are respectively:

\[
L_{\text{read-rem}}(I, M/E/S, -) = T_{\text{GC}} + L_{\text{net}} + T_{\text{lookup-GC}} + \]
\[
P(i \neq h)(T_{\text{ext-net}}) + T_{\text{GCout_h}} + \]
\[
P(i \neq h)(T_{\text{ext-net}}(\sigma)) + L_{\text{net}}(\sigma) + T_{\text{PrC}}
\]
4.3. BASE MEMORY ACCESS LATENCIES

\[ L_{\text{read-rem}}(I, -, M/E) = T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + \]
\[ P(i \neq h)(T_{\text{ext-net}}) + T_{\text{lookup-GCout}} + \]
\[ P(h \neq j)(T_{\text{ext-net}}) + T_{M}/T_{\text{ShC}}/T_{PrC} + \]
\[ P(i \neq h \text{ OR } h \neq j)(T_{\text{ext-net}}(\sigma)) + \]
\[ L_{\text{net}}(\sigma) + T_{PrC} \]

where \( GC_{\text{state or}} L_{\text{read-rem}}(LC_{\text{state}}, GC_{\text{state}}, T_{\text{state}}) \) now represents the union of the global state information maintained by the \( GC_{\text{in}} \) and \( GC_{\text{out}} \). Moreover, \( P(i \neq h) \) is the probability that home node is not the requestor node, and \( P(h \neq j) \) is the probability that home node is not the owner node.

For writing latency with the write-allocate policy, the same operations are required in the case 2 \((b)\) and the corresponding writing latencies change as before (functions \([4.2]\) and \([4.3]\)).

Moreover, in order to invalidate all the possible copies, condition 2 \((b)\) 4 \((c)\) changes in the following way:

\((c)\) some other entities have to be involved to satisfy a \text{rfo req} or a \text{write req} request (e.g., \( PrC_j \) of \( CMP_i \) and \( PrC_k \) of \( CMP_j \))

5. GC forwards the request to \( GC_{\text{out}}_h \) through the interconnection network(s) and in parallel forwards the request also to the possible local copies (e.g., \( PrC_j \))

6. \( GC_{\text{out}}_h \) checks the global state of the outer protocol and eventually forwards the request in order to \text{invalidate} the copies

(the request is forwarded through the interconnection network(s) to the specific(s) \( CMP(s) \) if \( j \neq k \))

7a. \( PrC_j \) and \( PrC_k \) invalidate their copies and notify GC and \( GC_{\text{out}}_h \) through the interconnection network(s)

\((7b.)\) for \text{rfo req} the data are sent back to \( PE_i \) through the interconnection network(s)

7. \( PE_i \) updates the local state and the \( PrC_i \), which performs the actual write and sends the outcome to \( P_i \)

Therefore, the corresponding writing latency changes in the following way, in the case of \text{rfo req}:
\begin{align*}
L_{\text{write-rem}}(I, S, -) &= T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + \\
&\quad P(i \neq h)(T_{\text{ext-\text{net}}} + T_{\text{lookup-\text{GC}out}_h}) + \\
&\quad P(h \neq j)(T_{\text{ext-\text{net}}} + T_{M}/T_{\text{ShC}}/T_{PrC}) + \\
&\quad P(i \neq h \ OR \ h \neq j)(T_{\text{ext-\text{net}}}(\sigma)) + \\
&\quad L_{\text{net}}(\sigma) + [L_{\text{inv}}(n,_{\text{sh}})] + T_{PrC} \\
\end{align*}

while, in the case of write_req the writing latency is:

\begin{align*}
L_{\text{write-rem}}(S, S, -) &= T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + \\
&\quad P(i \neq h)(T_{\text{ext-\text{net}}} + T_{\text{lookup-\text{GC}out}_h}) + \\
&\quad P(h \neq j)(T_{\text{ext-\text{net}}} + L_{\text{inv}}(n,_{\text{sh}})) + \\
&\quad P(i \neq h \ OR \ h \neq j)(T_{\text{ext-\text{net}}}) + \\
&\quad [L_{\text{net}}] + T_{PrC}
\end{align*}

\section*{4.4 Benchmarks for Reading and Writing Latencies}

An interesting study has been reported in \cite{52} where, by means of specific benchmarks, the authors measured the latency of read operations in two current multicore architectures (an Intel Nehalem and an AMD Shangai processor). This work shows that load/store latency and bandwidth are affected by the cache coherence state of the line, as discussed at abstract level in the previous section.

We decided to adapt the benchmark to the Intel SandyBridge and to the Tilera TilePro64 architectures, in order to have a quantitative evaluation for the access latencies derived in the previous part of this Chapter. Of course, benchmark results do not strictly match to base access latencies due to the use of multiple PEs in the systems, as we will discuss in Chapter \ref{chap:6}.

Tables 4.4 and 4.5 shows the benchmark results for memory read latencies respectively for the Intel SandyBridge and Tilera TilePro64 processors.

For the first architecture, we can easily notice that, in general, the access of a cache line in the shared state costs as much as reading a private data (i.e., from the private level caches); modified and exclusive states, on the other hand, increase the access latency. It should also be noted that in these measurements we always pay the cost of snooping that, although limited, could still be avoided in an incoherent approach.

For Tilera TilePro64 platform, of course we expect an increased overhead, as we deal with complex interconnection networks and protocols. In this case we have a
4.4. BENCHMARKS FOR READING AND WRITING LATENCIES

<table>
<thead>
<tr>
<th>Source</th>
<th>State</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>M/E/S</td>
<td>6</td>
<td>12</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>Same Chip</td>
<td>Modified</td>
<td>108</td>
<td>102</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other Chip</td>
<td>Modified</td>
<td>304-312</td>
<td></td>
<td></td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td>195-205</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>195</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Memory read latencies for core 0, depending on the cache line state, for Intel SandyBridge processors. Times in clock cycles.

limited amount of possibilities: if the automatic cache coherence is enabled, because of the write-through mechanism between the PrC and the home node, a cache line will always be either shared or invalid. The real difference is therefore only in having or not the cache coherence enabled. We can notice that the memory access time is increased from 120 up to 204 clock cycles. The latency depends on the distance between the home node and the requestor node, but we are talking of an overhead that goes from 33% up to 70% for each memory load.

Of course, even disabling cache coherence has a cost: with automatic cache coherence, if another cache has the required value, we pay from 40 to 70 clock cycles, depending on the distance; with an incoherent mode we have no way of knowing if another cache has the copy, and must always go to the memory paying 120 clock cycles. In fact, the ability of cache-to-cache transfers is always presented as one of the many benefits of automatic cache coherence mechanisms; however, it should be noted that cache-to-cache transfers mostly depend on the program and the cache sizes: for example, if the working sets of the processor do not fit in cache, the probability of exploiting cache-to-cache transfers is very limited. It should also be noted that, on this architecture, enabling the automatic cache coherence in fact lower the amount of second level cache (GC) space available for each core, thus possibly further increasing performance losses.

<table>
<thead>
<tr>
<th>Source</th>
<th>L1</th>
<th>L2</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>2</td>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>Incoherent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coherent HOME</td>
<td>40-70</td>
<td>160-204</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Memory read latencies for core 0, on the Tilera TilePro64 architecture, with or without cache coherence. Times in clock cycles.
In [91], a similar benchmark has been executed on the Intel xeon phi architecture [62].

We finally adapt the benchmark for writing operations in order to measure the *synchronous* implementation of the actions required. As we will see in the next Chapter, in order to solve memory ordering problems in architectures with weak memory consistency models, the run-time support (thus the cost model) need to consider the use and the performance impact of the synchronous semantics of writing operations. Tables 4.6 and 4.7 shows benchmark results for the memory write latencies respectively for the Intel SandyBridge and Tilera TilePro64 processors.

<table>
<thead>
<tr>
<th>Source</th>
<th>State</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Modified</td>
<td>9</td>
<td>15</td>
<td>39</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td></td>
<td></td>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td></td>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same Chip</td>
<td>Modified</td>
<td>102</td>
<td>97</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td></td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>83-95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other Chip</td>
<td>Modified</td>
<td>228</td>
<td>225</td>
<td>190</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td>213-215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>233-285</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Memory write latencies for core 0, depending on the cache line state, for Intel SandyBridge processors. Times in clock cycles.

Notably, for Intel SandyBridge architecture, analogous considerations of that made for read latencies can be applied to the write cases. Very important is the impact of the invalidations required by the cache coherence protocols which is a function of the number of copies shared among the PEs. Fortunately, we will see in the next chapter, that with Structured Parallel Programming and a proper process/thread mapping strategies we are able to minimize this number. Similar results are obtained for the Tilera TilePro64 platform, with the distinction in the case of using write-allocation policy or not.

### 4.5 Summary

In this chapter we study the impact of different automatic cache coherence solutions by reasoning on an *abstract model* which is able to capture all the essential information of CMP-based architectures and the specific cache coherence protocols.
4.5. SUMMARY

Table 4.7: Memory write latencies for core 0, depending on the cache line state, for Tilera TilePro64 processors. Times in clock cycles

<table>
<thead>
<tr>
<th>Source</th>
<th>L1</th>
<th>L2</th>
<th>M</th>
<th>wRead</th>
<th>w/oRead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Exclusive</td>
<td>5-28</td>
<td>174-219</td>
<td>57-86</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>12-292</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOME</td>
<td>Exclusive</td>
<td>45-73</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>52-332</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local = HOME</td>
<td>Exclusive</td>
<td>2-7</td>
<td>226-274</td>
<td>121-147</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>7-285</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notably, we provide a simple and effective way to evaluate the read and write memory operation latencies according to the specific cache coherence solution adopted by different abstract architecture models, by studying the impact on both single-CMP and multiple-CMP systems.

We finally used benchmark results for the evaluation of read and synchronous write latencies, in order to have a first validation of our cost model.

Of course, as discussed in the Introduction part of the thesis, this base latencies constitute the first step in the evaluation of under-load memory latencies which depends on the execution of the parallel program on the target architecture. This evaluation is studied in the following two chapters.
Automatic cache coherence solutions allow programmers to develop programs without taking into account the coherency problem. In fact no explicit coherency operations need to be inserted in the program by the application programmer. We are interested in evaluating the cost of these solutions and understand how to improve parallel programs’ performances with different approaches to the cache coherence problem. We know, from the previous chapter, that for each abstract model we have specific read/write latencies that are strictly correlated to the coherency state of data. In this chapter, we use these base latencies as a first approximation of the under-load memory latencies, which will be derived according to the methodology presented in the next chapter. Even if they represent only the base latencies, in this chapter we reason about how performances of parallelism forms defined by a structured parallel programming environment, are affected by them. Notably, in the first part of this Chapter (Section 5.1) we analyze different implementations of farm and data-parallel paradigms in order to evaluate the cost of the various operations executed on the run-time support data structures.

In Section 5.2 in the same way, we use the latencies derived from the previous Chapter, to evaluate the cost of the cooperation and synchronization mechanisms used in parallel paradigms implementations.

Finally, in Section 5.3 we analyze the impact of cache coherence (always in terms of the memory and cache base access latencies) on parallel programs’ performances which use lock-free data structures for communications between modules.
5.1 Recognizing Cache Coherence Patterns in Parallel Paradigms

We know, from the previous chapter, that for each abstract model of cache coherent architectures we have specific read/write latencies that are strictly correlated to the coherency state of data. Therefore, we can study well-known parallel pattern implementations and identify specific cache coherence patterns. In fact, knowing the structure of the parallel application and the mechanisms used in their implementations (e.g., shared memory or message passing model), we are able to

- identify data structures that must be kept coherent
- and use the latencies derived from the abstract models and apply them to the various parallel pattern implementations.

We consider now farm and data-parallel computations. For the time being, we focus on the computation data postponing the study on the synchronization data. In particular, for each parallel pattern we compare three different implementation models, depending on the way in which interactions between modules are expressed. The first one is a shared-variable environment in which processes cooperate by sharing data structures. Shared objects are used, at the same time, for computation data and for synchronization mechanisms (e.g., locks and condition variables). The second one also uses shared memory mechanisms, but in this case, synchronizations are implemented by passing pointers from producers’ and consumers’ modules. The messages exchanged by the various entities are actually references to the data, thus is not a local environment model like in the message-passing approach. In this last case, we map modules into processes and interactions between modules in communication channels used to send and receive the data. Therefore, shared data belong only to the run-time support data structures.

In the following, we used the term $PrC_p$ as a synonym for the private level cache of the PE where the process module $p$ is allocated.

5.1.1 Farm

Farm computations can be implemented according to several strategies, that can be mapped to each of the implementation models discussed above. In Figure 5.1 we show the most used strategies:

Master-Slave (M-S)  The stage 1 and 3 of the farm are both mapped on the same master module is responsible for: (1) collecting input elements; (2) selecting a worker for scheduling; (3) delivering the input element to the selected worker; (4) selecting a worker to collect a result; (5) collecting a result from the selected worker; (6) delivering the collected result to the output stream.
5.1. Recognizing CC Patterns in Parallel Paradigms

Emitter-Worker-Collector (E-W-C) Each stage of the farm is mapped to a different module: an emitter (E) on which we map the stage 1; a set of workers (W) on which we map the stage 2; and a collector (C) on which we map the last stage 3.

Consider now the following farm computation, in which the function $F$ is applied to the i-th element of the input stream (in_task) and produces the i-th element of the output stream (out_res):

```java
while (true) {
    out_res = F(in_task, ...);
    in_task = getNextTask(...);
}
```

To be as generic as possible, consider that in_task and out_res could be of small dimensions, e.g. a small set of integer numbers, or bigger, like matrices of real numbers. In addition, the function $F$ may need other data to compute the out_res. Therefore, starting from this scenario we can study its possible implementations.

Shared-variable implementation of Farm computations A shared-variable environment is often linked to a M-S strategy in which streams are implemented as shared queues, on which the Master (M) and Slaves $\{S_0, ..., S_{N-1}\}$ execute get and put operations atomically.

For each in_task, M performs the put operation and, more specific, the following actions:

1. it refers the next available element of the task queue $tq_i$ used to distribute the input element elements to the workers
2. it modifies $tq_i$ adding the address of the in_task

Figure 5.1: Farm implementation strategies
These actions actually correspond to specific cache coherency state and transitions:

1. the cache line that includes $tq_i$ is held in $PrC_M$

2. the coherency state of that cache line changes (if necessary), for example, to $Modified$; hence, $PrC_M$ holds the most recent copy of the data

**Observation 1.** *After performing the action 2, M no longer needs to use the data just modified. It will modify again $tq_i$ only after performing other $tq_{size} - 1$ put operations.*

When $S_i$ performs the *get* operation on the task queue, it actually reads the modified $tq_i$ and then applies $F$ to $in\_task$. Depending on the dimension of $tq_i$, which in this case is small because it is the address of $in\_task$, that read operation could refer to a cache line which is still in $PrC_M$. As we saw in Chapter [4], we know that this operation leads to the greater latency cost for the $S_i$.

**Cost Model 5.1.1 (Shared-variable Farm).** *In the worst case, the cost of a get operation executed by a $S_i$ process for accessing to the next $in\_task$ is*

$$T_{get} = T_{read}(I, -, M) + T_{synch}$$

*where $T_{synch}$ is the cost related to the synchronization mechanisms used in the get operation.*

Moreover, some read operations relative to $in\_task$, during the $F$ computation, may causes the same cost. In fact, $M$ could produce itself the input stream or receive, in turn, the data from another module and eventually modify them. This can lead to having part of or the entire $in\_task$ data structure still in $PrC_M$. Even in the case in which $M$ use $in\_task$ in a read-only way, $PrC_M$ may have to be involved in the read operations ($T_{read}(I, -, E)$ required by $F$. Therefore, the following observation is valid.

**Observation 2.** *Read operations related to $in\_task$ may involve the use of coherency protocols even if these are not actually necessary.*

If $M$ is also responsible for collecting the results from $S_i$, then all the considerations made on the *put* and *get* operations can be dually applied to the $out\_res$ and the corresponding results shared queue. Actually, here we know that the output is produced by each $S_i$ and when $M$ uses the results, the corresponding read operations cause the worst case cost, reading the modified data from $PrC_i$ ($T_{read}(I, -, M)$).
Pointer-passing implementation of Farm computations  In a pointer-passing solution an explicit Emitter process includes the input stream queue and transmits the pointer (reference) of a queue element to a Worker through a short message, so that the Worker can address the input data structure without copying it and without additional synchronizations. The dual solution is applied to the Worker-Collector cooperation.

For each in_task, E puts the message (msg) that contains the input data reference, while the selected Wi obtains it with the get operation. In the same way, for each out_res Wi put the message that contains the output data reference and C obtains it with the get operations. Therefore, all the considerations made above on the tqi can be applied to msg (and dually to the output data and results queue) with the same consequences for the coherency actions and costs.

Message passing implementation of Farm computations  In message passing solutions, modules (E/M, Si/Wi and C) are mapped in processes which work in a local environment, and their interactions are implemented as send and receive operations on communication channels. Thus, in_task and out_res are passed as messages in communication channels.

Consider an E-W-C strategy and an optimized implementation of message-passing primitives (e.g., a “zero-copy” communication which is able to reduce the number of message copies to one [107]). For each in_task, E sends the message, which in this case is exactly the in_task data structure, to Wi through the task_ch communication channel. With the send primitive, E performs the following actions:

1. it uses the next available target variable vtgi, that is the next available position in the buffer of the corresponding communication channel (i.e., the channel used by E to distribute tasks to Wi)

2. it copies in_task into vtgi

Therefore, as in the previous cases for tqi, these actions actually correspond to specific cache coherency state and transitions:

1. the cache line(s) that includes vtgi is(are) held in PrCE

2. the coherency state of that(those) cache line(s) is, for example, Modified; hence, PrCE holds the most recent copy of vtgi

and the Observation [1] is still valid and it can be applied to vtgi. When Wi uses the in_task obtained with the receive primitive, it actually read the modified vtgi and the following observation is valid.
Observation 3. In message passing solutions, $W_i$ works only on $vtg_i$, applying $F$ directly to it. Hence, coherency operations have to be applied only on $vtg_i$ which is the only data shared between $E$ and $W_i$.

Therefore, the read latency is paid during the $F$ computation and, even in this case, the read operation could refer to cache line(s) which is(are) still in $PrC_E$.

Cost Model 5.1.2 (Message-passing Farm). In the worst case, the cost for $W_i$ process for accessing to the next in_task is

$$T_{vtg} = T_{\text{read}}(I, -, M) \times N_{\text{lines}}$$

where $N_{\text{lines}}$ is the number of cache lines used for $vtg$.

(Note that in this case the synchronization cost ($T_{\text{synch}}$) is paid during the receive operation.)

All the considerations made on the E-$W_i$ interaction can be dually applied to the out_res and the corresponding $W_i$-C communication.

5.1.2 Data-Parallel: Map

The previous analysis can be extended to data-parallel computations on streams and/or on single data values. As in the farm computations data-parallel can be implemented according to several strategies. We consider an IN-W-OUT implementation (IN and OUT are simply more generic names for the E and C modules), which is able to describe most of the functionalities required in this type of computation. In data-parallel computations, IN provides the distribution of each input element among the set of workers according to proper collective communications (e.g. scatter or multicast). While, OUT is responsible for the collection of the worker results, with a gather or reduce operations.

In map computations, workers are fully independent, that is each of them applies a sequential elaboration on its own local data only.

Consider now the two following map computations, in which the function $F$ is applied to all the elements of the input $A$ and produces, respectively, a new (large) data structure output $B$ or a more synthetic and smaller result $x$.

The distinction between the two types of computations is useful only to distinguish the size of the data exchanged between $W_i$ and OUT.

In both cases the considerations made for farm computations remain valid. In fact, from the point of view of the cache coherence we have the same “producer-consumer” behaviour between IN-$W_i$ and $W_i$-OUT.
5.1. RECOGNIZING CC PATTERNS IN PARALLEL PARADIGMS

```c
int A[N], B[N], x;
for (i = 0..N-1) {
    B[i] = F(A[i]);
}
for (i = 0..N-1) {
    x = F(A[i]);
}
```

Shared-variable and pointer-passing implementations of Map computations

When IN performs a scatter distribution, each \( W_i \) reads the (modified) input task (i.e., the initial address of the appropriate partition of the data structure), when it performs a get on the shared queue.

While, when IN performs a multicast distribution, it merely puts the same address to all the shared queues.

Regarding the coherency operations related to the \( W_i \)-OUT interaction, as above, the smaller the output the greater the possibility to have both the address and the data itself still in \( PrC_{W_i} \). Therefore, the cost model 5.1.1 is also valid for map computations.

Message-passing implementation of Map Computations

When IN performs a scatter distribution, each \( W_i \) reads the (modified) target variable which contains its partition of the input task, during the \( F \) computation.

While, when IN performs a multicast distribution, it sends the same message to each worker.

As for Farm computations, the same considerations on the target variable can be done also for the \( W_i \)-C interaction, and the cost model 5.1.2 is also valid for map computations.

5.1.3 Data-Parallel with Stencil

In order to apply its function, a worker may need to access data contained in other worker partitions, according to the particular data dependencies imposed by the computation semantics. In this case we speak about stencil-based computations, where a stencil is a data dependence pattern implemented by information exchanges between different workers.

We consider the case in which the function application is iterated for a given (possibly unknown) number of steps. A worker can communicate with other workers to obtain their previous results for its local computation. For example, we can functionally describe the k-th application of a given function \( F \) on the i-th position
of an array $A$ in the following way:

$$A^k[i][j] = F(A^{k-1}[i][j], A^{k-1}[i-1][j], A^{k-1}[i][j-1], A^{k-1}[i+1][j], A^{k-1}[i][j+1])$$

In this example, to compute the $k$-th value of $A[i][j]$ we need the values of the previous iteration ($k-1$) of the same element and its “neighbors”. If these latter values are assigned to another worker

- $W_i$ and $W_{i+1}$ share their border values in a shared-variable implementation;
- a communication is required between $W_i$ and $W_{i+1}$ in message-passing and pointer-passing implementations.

**Shared-variable implementation of Stencil-based computations** As represented in pseudo-code 5.2 at each step of the computation, $W_i$ uses two data-structures: one ($A_{in}$) containing the results computed at previous step and the other ($A_{out}$) used to store the results of the current step. In order to maintain the data consistent across the various steps, a synchronization among workers is required. It may be implemented by locking mechanisms or by global barriers.

```c
int A_{in}[N][N], A_{out}[N][N];
...
while (!done) {
  for (i=myMin..myMax) {
    for (j=0..N-1) {
      A_{out}[i][j] = F(A_{in}[i][j], A_{in}[i-1][j], A_{in}[i][j-1], A_{in}[i+1][j], A_{in}[i][j+1]);
    }
  }
  done = computeEndCondition(...);
  if (!done)
    swap(A_{in}, A_{out});
  BARRIER();
}
...
```

Figure 5.2: Pseudo-code for stencil computation in shared-variable implementation

Consider now a stencil computation from the point of view of cache coherence, remaining as generic as possible with respect to the kind of stencil computation. At step $k$, $W_i$ uses $A_{in}$ elements to computes $A_{out}$ elements. However, $A_{in}$ elements have been actually modified at the step $k-1$, possibly by another worker. This means that $W_i$ acts as a consumer that reads the required data by accessing the producer Worker $W_j$ partition. Therefore, at each step $W_i$ performs a certain number (depends on the stencil structure) of read operations related to *modified* data (and
invalidated during the previous step by $W_j$). Depending on the stencil and the computation organization (e.g., priority to local value calculation), that data may still be kept in $PrC_j$.

**Cost Model 5.1.3** (Shared-variable Stencil). In the worst case, the cost of read operations for the stencil-data by a $W_i$ process is

$$T_{stencil} = T_{\text{read}}(I, -, M) \ast N_{\text{stencil-lines}}$$

where $N_{\text{stencil-lines}}$ is the number of cache lines used for the stencil-data.

**Message-passing and pointer-passing implementations of Stencil-based computations** Stencil-based computations are typically structured in two phases in the message-passing approach. As shown in pseudo-code 5.3, each worker first performs the stencil required communications and, after that, it applies the function to its partition. Here, communications also act as synchronizations across the various computation steps.

```
1 int $A_{in}[N][N], A_{out}[N][N]$;
2 ...
3 while (!done) {
4     if (myId != 0) then send($A_{in}[1][*], N, myId-1$);
5     if (myId != nw-1) then send($A_{in}[g][*], N, myId-1$);
6     if (myId != 0) then receive($A_{in}[0][*], N, myId-1$);
7     if (myId != nw-1) then receive($A_{in}[g+1][*], N, myId-1$);
8 
9     for (i = 1..g-1) {
10         for (j = 0..N-1) {
11             $A_{out}[i][j] = F(A_{in}[i][j], A_{in}[i-1][j], A_{in}[i][j-1], A_{in}[i+1][j], A_{in}[i][j+1])$;
12         }
13     }
14     done = computeEndCondition(...);
15     if (!done) then
16         swap($A_{in}, A_{out}$);
17 }
18 ...
```

Figure 5.3: Pseudo-code for stencil computation in message-passing implementation

From the point of view of cache coherence, message-passing still delimits coherency operations for stencil-data to the relative target variables. Therefore, we can evaluate the cost of the stencil communications as follows.
Cost Model 5.1.4 (Message-Passing Stencil). In the worst case, the cost of read operations for the stencil-data by a $W_i$ process is

$$T_{\text{stencil}} = T_{\text{vtg}} \ast N_{\text{stencil-vtg}}$$

where $N_{\text{stencil-vtg}}$ is the number of receive communications required by the stencil.

If communication among workers is implemented by pointer-passing, than the solution is similar to the classical message-passing. Anyway, as we saw for E-$W_i$ and $W_i$-C interactions, cache coherency operations may be required both for stencil-data references and stencil-data itself (with a possible additional cost of $T_{\text{read}}(I, -, M/E)$).
5.2 Synchronization Issues in Shared Memory Systems

As we saw in the previous section, parallel paradigm implementations need proper mechanisms for processor/thread cooperation and synchronization. Synchronization operations typically rely on some atomic read-modify-write hardware primitives, in which the value of memory location is read, modified and written back atomically without intervening access to the location by others.

The focus of this section is on how synchronization operations are implemented in cache coherent systems, paying attention to memory ordering issues. In particular, we describe the implementation of mutual exclusion through lock-unlock pairs and global event synchronization through barriers. We also consider the consequences of the use of lock-free approaches, based on the classic work of Lamport [69]. These solutions provide concurrent algorithm on specific data structures which solve mutual exclusion without locking mechanisms.

5.2.1 Mutual Exclusion

Two threads in the same address space, or two processes in shared memory architectures, accessing common resources must synchronize their behavior in order to avoid wrong or unpredictable behavior. The period of exclusive access is referred to as a critical section, which is enforced through mutual exclusion implying that only one entity (thread or process) at a time is able to execute this critical section of code.

Mutual exclusion is ensured by enclosing the indivisible sequence of actions between two synchronization operations, called lock and unlock, which are implemented using a wide range of algorithms.

CAS-based locking

A typical solution to the lock problem is to have a single shared variable v and acquiring the lock is done by using an atomic read-modify-write instruction such as test_and_set or compare_and_swap. Using test_and_set on v, the value in the memory location M which stores v is read into a specified register of the PE that performs the instruction, and the constant 1 is stored into the location M atomically if the value read is 0. As shown in the pseudo-assembly code shown in Listing 5.1, the lock implementation keeps trying to acquire the lock using test&set instructions, until it returns zero indicating that the lock was free when tested.

The unlock construct simply sets the location associated with lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. More sophisticated variants of such atomic instructions exist, e.g. swap-based or fetch&op instructions.

From the point of view of cache coherence, every attempt to check whether the lock
is free to be acquired, whether successful or not, generates a write operation (to write the value 1) to the lock variable cache line. Since this line is currently in another cache, e.g. \( PrC_j \) because \( P_j \) wrote it doing the test-and-set, an invalidation request is sent through the interconnection network by each write to invalidate the previous owner of the block. Moreover, when \( P_i \), which is executing the critical section, performs the unlock operation, it refers to a modified location in another private cache, e.g. \( PrC_j \). Therefore, the write operation causes the worst case cost.

We can estimate the cost of lock and unlock operations in terms of the read and write operation latencies from Chapter 4.

\[
T_{\text{max}}^{\text{LOCK}} = T_{\text{write}}(I, -, M) \\
T_{\text{max}}^{\text{UNLOCK}} = T_{\text{write}}(I, -, M)
\]

Both lock and unlock operations can take advantages of the reuse of the same location respectively doing consecutive lock/unlock operations and when no other processes have executed a lock operation during the critical section execution. We represent with \( p_{FREE} \) the probability of finding the lock free during the lock operation. Therefore, unlock operation and consecutive lock operations have the following cost.

\[
T_{\text{min}}^{\text{LOCK}} = p_{FREE} * T_{\text{write}}(M, -, -) + (1 - p_{FREE}) * T_{\text{write}}(I, -, M) \\
T_{\text{min}}^{\text{UNLOCK}} = T_{\text{write}}(M, -, -)
\]

Two simple variants of this algorithm are typically used for two main reasons: (1) to reduce the frequency with which test-and-set instructions are issued, (2) to reduce the invalidations and misses during the busy-wait.

In the first case, the basic idea is to insert a delay after an unsuccessful attempt to acquire the lock. This solution is called lock with backoff. In the second case, a test-and-test&set lock solution is adopted. Busy-wait is done by repeatedly reading with a standard load, not a test-and-set, the value of the lock variable until it

---

Listing 5.1: Lock and unlock implementation with the test-and-set instruction

```
lock: t&s register, location     //copy location to reg. and
   \( \text{bnz} \) register, lock  //if 0 set location to 1
      \( \text{bnz} \) register, lock  //compare old value with 0
      \( \text{ret} \) \hspace{1em} //if not 0, try again

unlock: st location, #0         //write 0 to location
     \( \text{ret} \)
```

---
5.2. SYNCHRONIZATIONS IN SHARED MEMORY SYSTEMS

turns from 1 (locked) to 0 (unlocked). On a cache coherent system, the reads can be performed in the private cache by all processors, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes (e.g., P\textsubscript{j} and P\textsubscript{k}) are invalidated, and the next read of the variable will generate a miss. When P\textsubscript{j} then finds that the lock has been made available, it will only then generate a \texttt{test_and_set} instruction to actually try to acquire the lock.

With the test-and-test\&set solution, the cost for lock and unlock operations, considering the reuse of the lock variable location, become as follows.

\[
T_{\text{max}}^{\text{LOCK}} = T_{\text{read}}(I, -, M) + p_{\text{FREE}} \times T_{\text{write}}(S, -, -) \\
T_{\text{min}}^{\text{LOCK}} = p_{\text{FREE}} \times (T_{\text{read}}(M, -, -) + T_{\text{write}}(M, -, -)) + (1 - p_{\text{FREE}}) \times T_{\text{read}}(I, -, M)
\]

\[
T_{\text{max}}^{\text{UNLOCK}} = T_{\text{write}}(I, -, M) \\
T_{\text{min}}^{\text{UNLOCK}} = T_{\text{write}}(M, -, -)
\]

Improved instruction sets

Several instruction sets provide a pair of instructions called \texttt{load-locked} or \texttt{load-linked} (LL) and \texttt{store-conditional} (SC) to implement atomic operations, instead of an atomic read-modify-write instructions like test\&set. These instructions allow to avoid the invalidations generated by failed attempts to complete the read-modify-write phase. The LL instruction loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register. The last instruction of the sequence is the SC instruction, which writes the register back to the synchronization variable location if and only if no other processor has written to that location since this processor completed its LL. Therefore, if the SC succeeds, it means that the LL-SC pair has read, perhaps modified, and written back the variable atomically. If the SC detects that a write has occurred to the variable, it fails and does not write the value back or generate any invalidations. This means that the atomic operation on the variable has failed and must be retried starting from the LL. Using LL-SC to implement atomic operations, lock and unlock primitives can be written as shown in Listing 5.2.

PowerPC architectures use this solution to support atomic operations [88]. In particular, this mechanism is implemented by the cache coherency protocol and uses reservation information attached to each cache line. When issuing a \texttt{lwarx} (LL instruction in PowerPC’s instruction set), the reservation for the current thread is simultaneously set with the load operation. In order to have the right to perform a store operation to a cacheline the issuing cache must contain the data exclusively, that is no other cache contains the data. If the cache line is not exclusively held, a request (e.g., \texttt{rfo}\_req) is sent. Finally atomic operations are realized by looping.
Parallelism and Cache Coherence

lock: ll reg1, location //load-linked the location
    //to reg1
    bnz reg1, lock //if location was locked
        //(not 0), try again
    sc location, reg2 //store reg2 conditionally
        //into location
    beqz lock //if SC failed, start again
    ret

unlock: st location, #0 //write 0 to location
    ret

Listing 5.2: Lock and unlock implementation with LL/SC instructions

until the stwx (SC instruction in PowerPC’s instruction set) succeeds, i.e. invalidations are not required.
Relying on this implementation, we can evaluate the cost of lock and unlock operation based on LL/SC instructions as follows.

\[
T_{\text{max}}^{\text{LOCK}} = T_{\text{write}}(I, -, M) + p_{\text{FREE}} * T_{\text{write}}(M, -, -) \\
T_{\text{min}}^{\text{LOCK}} = p_{\text{FREE}} * 2 T_{\text{write}}(M, -, -) + (1 - p_{\text{FREE}}) * T_{\text{write}}(I, -, M)
\]

\[
T_{\text{max}}^{\text{UNLOCK}} = T_{\text{write}}(I, -, E) \\
T_{\text{min}}^{\text{UNLOCK}} = T_{\text{write}}(M, -, -)
\]

Atomic operations on x86 processors are implemented using a lock prefix: the lock instruction can be prefixed to a number of operations and has the effect to lock the system bus (sometimes only the local cache in recent architectures) to ensure exclusive access to the shared resource. In recent versions of these architectures, the MONITOR and MWAIT instructions have been introduced [60]. The MONITOR instruction supervises a certain memory location for the occurrence of special events, and the MWAIT waits for that event or for a generic interrupt. One possible use is the monitoring of store events: a lock operation can be implemented by using these two instructions. The drawback of the MONITOR/MWAIT instructions is that they must be used in kernel space, with the consequent use of costly system calls to perform these operations.

Reducing invalidations

Some other advanced lock algorithms have been proposed both for providing fairness and to reduce traffic caused by invalidations [37, 76]. In fact, LL-SC is not fair and
when \( P_i \) succeeds in its SC instruction, all other cached copies are invalidated and other processors all incur read operations of modified data in \( PrC_i \).

The ticked-lock algorithm tries to minimize traffic due to the SC instruction. In this solution, a process wanting to acquire the lock takes a number and atomically increments it at the same time, by an atomic fetch and add instruction. Then it busy-waits on a global now-serving number, until this number equals the number it obtained. The corresponding unlock operation simply increments the now-serving number. Ticked-lock implementations still have the problem that all processes spin on the same now-serving variable. When that variable is released by \( P_i \), as above for SC instruction, all other cached copies are invalidated and other processors all incur read operations of modified data in \( PrC_i \).

An alternative is provided by the Array-based Lock algorithms, which eliminates this extra read traffic by having every process spin on a distinct location. In this case, the traffic is reduced because only the processor \( P_j \) that was spinning on a specific location has its cache line invalidated by the unlock executed by \( P_i \). \( P_j \) still incurs a read operation of modified data in \( PrC_i \).

### 5.2.2 Global (barrier) event synchronizations

Global synchronizations are typically required in the run-time support for parallel applications especially in some initialization phases. In some cases, a global synchronization between the entities of a parallel program has to be performed in order to ensure the correctness of the application. For example, in a shared-variable implementation of a data-parallel with stencil application, a global coordination among the workers is required across the various steps of the computation. For this reason, an evaluation of the cost of this kind of operation becomes necessary.

Algorithms for barrier synchronizations are typically implemented using shared variables (e.g., counters and flags) and manipulating their values in mutual exclusion. A simple barrier among \( p \) processes or threads is the so-called centralized barrier, in which a single counter and a single flag is used. The shared counter maintains the number of processes that have arrived at the barrier, and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, the process checks to see if the counter equals \( p \), i.e. if it is the last process to have arrived. If not, it busy waits on the flag associated with the barrier, otherwise it writes the flag to release the waiting processes.

The pseudo-code in Listing 5.3 shows how this algorithm can be implemented. In particular, it presents the sense reversal version of the algorithm [37], in which a private variable (i.e., local sense) is used to prevent a process from entering a new instance of a barrier before all processes have exited the previous instance of the same barrier.

The lock/unlock protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation. Relying on this im-
\begin{lstlisting}[language=C]
struct barrier_type {
    int counter;
    struct lock_type lock;
    int global_sense = 0;
} bar;

BARRIER (bar, p) {
    local_sense = !local_sense;
    lock(bar.lock);
    mycount = bar.counter++; /* mycount is a private variable*/
    unlock(bar_name.lock);
    if (mycount == p) { /* last to arrive */
        bar.counter = 0; /* reset counter for next barrier */
        bar.global_sense = local_sense; /* release waiting processes */
    } else {
        while (bar.global_sense == local_sense) {} /* busy wait for release */
    }
}
\end{lstlisting}

Listing 5.3: Global barrier algorithm with mutual exclusion

Implementing, we can evaluate the latency of a barrier operation executed by \( p \) processes or threads as follows.

\[
T_{\text{BARRIER}}(p) = (p - 1)T_{\text{write}}(I, -, M) + T_{\text{write}}(S, -, -)[p - 1]
\]

The first \( p - 1 \) executions of the barrier cause several transfers of the cacheline containing the \texttt{barrier_type} data structure due to the atomic increment of the counter. Finally, the last execution causes the invalidations of the other \( p - 1 \) copies of the cache line holding the \texttt{global_sense} flag.

Several alternative implementations have been proposed for the barrier algorithm, e.g. tree-based barrier [76]. In particular, in order to minimize the contention on the centralized flag, these solutions use different flags and organize the various processes in subgroups. The consequences in terms of the latency of the barrier operations consist in a decreasing number of invalidations on the same flag during the release phase.

5.2.3 Memory Ordering and Memory Consistency Models

With \textit{memory ordering} we mean the order in which memory operations (read and write) are performed. Memory ordering might be changed with respect to the or-
5.2. SYNCHRONIZATIONS IN SHARED MEMORY SYSTEMS

der specified in the program (program order) for some reason. The compiler might change memory ordering as a result of static optimizations, or out-of-order processor might change memory ordering as a result of dynamic optimizations. In this case, mechanisms such as compiler barrier \(^1\) can be used to prevent the reorder of read and write operations in order to ensure correctness of algorithms. Moreover, because of the nondeterminism behavior of shared memory systems, problems arise with the order in which memory operations, which are executed by a PE on shared locations, become visible to the other PEs. Notably, in this case we need to consider the Memory Consistency Model adopted by the target architecture \([85, 99]\). The interested reader can refer to \([75]\) for a detailed characterization of the various models. For our purposes we distinguish between two main classes of main memory consistency strategies:

- **Total Store Ordering** (TSO) in which
  
  - Loads are ordered with respect to earlier Loads
  
  - Stores are ordered with respect to earlier Load and Stores

  Thus, Load can bypass earlier Stores but cannot bypass earlier Loads. Stores cannot bypass earlier Loads or Stores, ensuring that store operations are visible to the system in program order. Examples or this model include x86 TSO \([85]\) and SPARC TSO.

- **Weak Store Ordering** (WSO) which does not guarantee such orderings (i.e., two distinct writes at different memory locations may be executed not in program order). Examples or this model include Tilera \([19]\), Power \([99]\) and ARM \([31]\) processors.

### Locking mechanisms and memory ordering

To complete the evaluation of mutual exclusion mechanisms consider the execution of lock/unlock operations in systems with a WSO memory consistency model. Notably, write operations executed by \(P_i\) inside the critical sections could be visible to a generic \(P_j\) after the unlock’s write operation. Thus, \(P_j\) might use non-consistent value of the critical section data. For this reason, to ensure the correct memory ordering, a memory fence or write memory barrier instruction has to be inserted in the unlock operation before the store instruction.

Following a structured parallelism approach, where a clear level-structuring is defined, all memory ordering problems are confined inside the run-time support of process primitives (e.g., in the send-receive run-time support). In this way, no other modifications of the ordering of memory operation done at compile-time on sequential codes

\(^1\)modifications of the ordering of memory operation done at compile-time on sequential codes
instances of this problem can affect the parallel program design at the process level.

From the cost model point of view, also atomic read-modify-write instructions, implicitly incorporate (behave as) a memory fence both on systems with TSO and WSO. An interesting evaluation of the cost of cas operation on TSO processor is discussed in [7]. Moreover, an explicit memory fence instruction is also required in WSO systems after each atomic read-modify-write implemented through a LL/SC pair.

Therefore, this means that we need to consider synchronous writing semantics in the locking latency evaluation. That is, none of the latencies related to the operations required by the coherency protocol (e.g., invalidations latencies) can be overlapped with others and considered negligible. Therefore, the result is a latency that grows with the number of invalidations/updates necessary, which can be considerable especially in the case of global event synchronizations.

5.3 Lock-free Data Structure for Communication Mechanisms

In the run-time support of stream-based parallel computations, it is essential to adopt a communication mechanism, used between the various modules, with the smallest overhead possible. Typically, shared queues or similar data-structures are the basic building block both in the shared-variable, message-passing and pointer-passing implementation model. For these reasons, concurrent lock-free queues have been widely studied in the literature [68, 81, 48, 15, 79, 104].

A shared data structure is lock-free if its operations do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in a finite amount of time, even if other processes halt, the data structure is non-blocking. If the data structure operations can guarantee that every process will complete its operation in a finite amount of time, then the data structure is wait-free. Therefore, wait-free protocols are a subclass of lock-free protocols characterized by stronger properties: roughly speaking lock-free algorithms are based on retries, while wait-free algorithms guarantee termination in a finite number of steps.
Consider a simple lock-based queue and the corresponding \texttt{enqueue} and \texttt{dequeue} operations, as shown in the Listing 5.5.

Locking queues can have an algorithmic source of overhead. In fact, lock and unlock operations strongly couple the producer and consumer. Even when the consumer is reading an earlier enqueued element, the producer cannot enqueue an element into a different buffer position.

Lamport proved that, under sequential consistency \cite{69}, the locks could be removed in the single-producer/single-consumer case, resulting in a concurrent wait-free queue. Thus, Lamport’s queue requires no explicit synchronization between the producer and consumer decoupling the two at the algorithmic level.

However, there still exists an implicit synchronization between the producer and consumer as the control data (i.e., head and tail) are still shared. In fact, head and tail are used to implicitly indicate full and empty queue conditions.

From the point of view of cache coherence, this means that the enqueue and dequeue operations read the constantly modified head and tail indexes (in addition to the buffer data).

Relying on the Lamport queue implementation the latency of enqueue and dequeue operations...
operation can be evaluated as follows.

\[ T_{\text{max}}^{\text{enqueue}} = T_{\text{max}}^{\text{dequeue}} = T_{\text{sync}} + T_{\text{data}} = T_{\text{read}}(I, GC, -) + T_{\text{read}}(I, -) + T_{\text{write}}(M, -) + T_{\text{data}} \]

In particular, \( T_{\text{sync}} \) represents the latency related to the control data operations, which in a lock-based implementation also includes the locking operations latency. While, \( T_{\text{data}} \) is strictly related to the type of data exchanged, as we saw in section 5.1.

A first optimization consists in having both head and tail indexes in the same cache line, resulting in the following evaluation.

\[ T_{\text{max}}^{\text{enqueue}} = T_{\text{max}}^{\text{dequeue}} = T_{\text{sync}} + T_{\text{data}} = T_{\text{read}}(I, -) + T_{\text{read}}(S, -) + T_{\text{write}}(M, -) + T_{\text{data}} \]

Note that we evaluate the worst case latencies. Both enqueue and dequeue operations can take advantage of the reuse of the control data respectively doing consecutive enqueue and dequeue operations. In this best case scenario, the latency related to the control data operations can be evaluated as follows (the cache line state in the second term depends on the number of cache lines used for head and tail indexes as discussed above).

\[ T_{\text{sync}} = T_{\text{read}}(M, -) + T_{\text{read}}(S/M, -) + T_{\text{write}}(M, -) \]

If the sequential consistency requirement is released, Lamport’s algorithm fails. For example, when write to write relaxation is allowed (i.e., two distinct writes at different memory locations may be executed not in program order), the consumer may incur a read of stale data. In fact, the update of the tail index, modified by the producer, can be seen by the consumer before the producer writes in the tail position of the buffer.

A few simple modifications to Lamport’s algorithm have been proposed for pointer-passing implementation models to allow correct execution even under weakly ordered memory consistency models [48, 15]. This solution is shown in Listing 5.6. In particular, an empty position in the buffer is represented using a know value, called bottom (\( \bot \)), that cannot be used as buffer element. In this way, the consistency problem of Lamport’s algorithm cannot occur provided that the generic store buffer[tail]=data is seen in its entirety by a processor, or not at all, i.e., a single memory store operation is executed atomically.

From the point of view of cache coherence, using the \( \bot \) value also solves the sharing problem between producer and consumer about the head and tail indexes. In fact, the head and tail indexes are always in the local cache of the consumer and the
5.3. LOCK-FREE DATA STRUCTURE FOR COMM. MECHANISMS

Listing 5.6: ⊥-based lock-free queue implementation

```c
enqueue(data) {
    if (buffer[tail] != ⊥) { /* the queue is full */
        return false;
    }
    buffer[tail] = data; /* copy the data into the queue */
    tail = NEXT(tail); /* change the insertion index */
    return true;
}

dehqueue(data) {
    data = buffer[head]; /* extract the next data */
    if (data == ⊥) { /* the queue is empty */
        return false;
    }
    buffer[head] = ⊥
    head = NEXT(head); /* change the extraction index */
    return true;
}
```

producer respectively, without incurring read operations of modified control data. Therefore, the latency of enqueue and dequeue operations can be evaluated as follows.

\[
T_{\text{enqueue}}^{\text{max}} = T_{\text{dequeue}}^{\text{max}} = T_{\text{sync}} + T_{\text{data}}
\]

\[
= T_{\text{read}}(I, GC, -) + T_{\text{write}}(E, -, -) + T_{\text{data}}
\]

In the best case scenario, the latency related to the control data operations can be evaluated as follows.

\[
T_{\text{sync}} = T_{\text{read}}(M, -, -) + T_{\text{write}}(M, -, -)
\]

These evaluations rely on implementation techniques used to avoid false-sharing. In fact, this problem can arise because the cache coherence protocol works at cache line granularity. For example, there is actually an implicit sharing of the control data queue if tail and head reside in different cache lines. In order to avoid this situation, a proper amount of padding is required to force the two indexes to reside in different cache lines.

As we said, in these solutions the communication buffer is used to transfer references of data. In WSO systems, the enqueue algorithm may need to be slightly modified to introduce a memory fence instruction before the reference is written
into the queue’s buffer (Listing 5.6 line 5). Without a memory fence the queue’s
buffer write could be visible to the consumer before the referenced data has been
committed in memory, potentially resulting in a read of stale data.

**Multiple Producer and/or Multiple Consumer Queues**

The following step consists in providing one-to-many (SPMC), many-to-one (MPSC),
and many-to-many (MPMC) communication mechanisms. SPMC, MPSC, and
MPMC queues can be realized in several different ways, for example using locks,
or in a lock-free way in order to avoid lock overhead.

Lamport’s queues have been extensively studied and extended [79, 104] especially fo-
cusing on improving the performance of the more general but more difficult MPMC
variants.

However, all these queues could not be directly implemented in a lock-free way with-
out using at least one atomic read-modify-write operation, which is typically used to
guarantee the correct serialization of updates from either many producers or many
consumers. As we saw in Section 5.2.1 these atomic operations implicitly behave
as a memory fence instruction, which can result in considerable synchronization la-
tency due to the required coherency protocol operations (i.e., cache invalidations or
updates).

In addition, an inherent problem in these MP and/or MC communications is the
so-called **ABA problem** [79]. The ABA problem occurs when a location is read twice
by a process (or thread) $P_1$, has the same value for both reads, and the fact that
the “value is the same” means that “nothing has changed” between the two reads.
If, between the two reads, another process (or thread) $P_2$ changes the value, does
other work, then changes the value back, thus the $P_1$ might think that “nothing
has changed” even though $P_2$ did work that violates that assumption. Handling the
ABA problem requires particular mechanisms and/or strategies to correctly ensure
that all parties (producers and/or consumers) agree on the order of transactions:
dual-lock queues in lock-based solutions and deferred reclamation/hazard pointers
or two bottom values in lock-free alternatives [79, 104].

These solutions build MP queues and MC queues as passive entities on which
processes or threads concurrently synchronize to access data. A different approach
[15] is based on the use of an active entity (i.e., process or thread) that acts as
an arbiter for the synchronizations among producers or consumers. Consider a
structured parallel application and the various implementation strategies that we
have discussed in Section 5.1 the emitter and the collector modules (or the master
module in the MS strategy) can actually be the active entity. According to their
role, the emitter and the collector perform enqueue or dequeue operations on one or
more lock-free SPSC queues. While, MPMC queues can be implemented combining
the emitter and collector functionalities with the consequent cost of an additional
memory copy. Therefore, this solution avoids the use of atomic read-modify-write
5.4. Summary

In this Chapter we analyzed the impact of automatic cache coherence solutions to the performance of parallel programs defined by well-know parallelism paradigms. By using, as a first approximation, the base memory latencies derived in the previous chapter, we reason about how performances of parallelism forms defined by a structured parallel programming environment, are affected by them. Notably, knowing the semantic, as well as the structure of the implementation of each parallelism form, we are able to derive important properties like a “producer-consumer” cache coherence pattern which is recurrent in all of them.

These properties, easily drive to definition of a cost model for the various paradigms both from the point of view of the computation data and the synchronization mechanisms.

Notably, we analyzed the state-of-the-art of synchronization techniques and com-
munication mechanism from the point of view of the cache coherence impact, which are typically used in the implementation of parallel programs. This Chapter allowed us to start from an evaluation of cache coherence mechanisms in terms of read and write latencies up to understand how the interactions between the modules of parallel paradigms are in relation (in terms of performances) with the coherency state of, and operations executed on, both computation and synchronization data structures.

The next step to complete the evaluation of the impact of cache coherence on parallel programs’ performance require to use the results obtained until now (i.e., costs of the operations executed by the various modules of the parallel program and how modules obtain the data through the cache coherence protocol) with a cost model which takes into account of the under-load behavior of a parallel program.
In this chapter, we finally discuss how to evaluate the under-load memory latencies in order to complete the evaluation of the performances of parallel application executed on CMP-based systems.

In the first part (Section 6.1), we use the results of Queueing Theory for the client-server model. The work presented in [22], was probably the first to introduce the main idea of modeling the Processor-Memory subsystem of a shared memory architecture, considering the memories as servants and the processors as clients. Notably, we apply the extension of the work proposed in [107].

The results obtained are used to reason about the effect of under-load latencies to parallel program performances.

In Section 6.2, we discuss the impact of specific choices (e.g., parallel process mapping and cache coherence optimizations) in the run-time support of parallelism forms.

In the second part of this chapter (Section 6.3), we combine the base memory and cache latencies defined in Chapter 4 with an interesting performance evaluation tool: Performance Evaluation Process Algebra (PEPA). This process algebra represents an alternative approach to study different aspect of the methodology used in this thesis. Notably, we are able to describe a complete abstract architecture (i.e., with different levels of servers for each level in the memory hierarchy) and derive a complete cost model for the various abstract models used to model automatic and non-automatic cache coherence solutions.
6.1 Cost Model for Under Load Memory Latencies

To evaluate a multiprocessor system we can use a queuing model with $m$ distinct and independent servers, each of which is a memory macro-module $\text{M}_i$ and $p$ clients, where $p$ is the average number of PEs sharing the same memory macro-module. Logically, the interconnection network path from each of the $p$ nodes to the shared macro-module, including the used switch nodes, belongs to the server. For a complete treatment about the evaluation of network latencies, the interested reader can consult \cite{89,107}.

An example of this scheme is shown in Figure 6.1, where $n$ is the number of PEs in the system and $p \leq n$. $T_i$ represents the average response time of the module $i$, while $T_a$ is the total interarrival time to the queue. Notable cases of this interaction pattern are some client-server parallel applications as well as processor-memory systems.

The main goal in a client-server system with request-reply behaviour is to estimate the average response time $R_Q$ of $S$.

In client-server models with request-reply behaviour, in order to reduce the response time both the service time (utilization factor) and the latency of the server are critical. Thus, we need techniques able to improve both the bandwidth and the latency of the shared memory macro-modules and of the network. A modular

\footnote{We call a memory macro-module a memory subsystem consisting of a memory interface and some memory modules}
memory organization is adopted, often relying on cache line interleaving to increase the bandwidth. This solution applies to any shared memory support, for example to shared caches too. In fact, some CMP shared caches have a modular interleaved structure [95].

Let \( p \) be the average number of processing nodes sharing the same memory/cache module. This number depends on how data structures are shared between processors or threads. For example, consider a synchronization variable which is shared between \( k \) processors or threads, then \( p \) could be equal to \( k \). It is very important that \( p \) is as low as possible in order to minimize the congestion overhead at the server.

In a SMP architecture, in which statistically the memory accesses are uniformly distributed over the \( m \) macro-modules, \( p \) can be estimated as the mean of the binomial distribution, i.e. \( p = n/m \).

In NUMA architectures, the uniform distribution does not hold, and the \( p \) value depends on specific characteristics of the parallel program and its mapping onto PEs. \( p \) value is greater than one when different processing nodes access the same data, different data belonging to the same cache line, or to different cache lines allocated on the same memory macro-module. In the next section we will discuss optimality issues in NUMA architecture mapping.

The under-load memory access latency is given by the server response time. Let \( R_{Q0} \) be the base latency, which is the latency evaluated in absence of contention (e.g., as evaluated in Chapter 4). In NUMA architectures it is meaningful to distinguish between remote memory access latency \( (R_{Q-rem}) \) and local memory access latency \( (R_{Q-loc}) \). Assuming that, in case of conflict between a remote request and a local request, this last is served with higher priority, we can write:

\[
R_{Q-loc} = R_{Q0}(1 + \rho)
\]

where \( \rho \) is the server utilization factor, which is a result of the queuing model resolution and expresses a global, average measure of the congestion degree of the requests to the server. The analysis can be used also for cache-to-cache transfers: the server is modeled by a remote \( PrC \) and by the interconnect path, either inside the same CMP or between distinct CMPs.
6.1.1 Model Resolution

We use the following system of equations as resolution technique of the client-server system.

\[
\begin{align*}
T_{cl} &= T_p + R_Q \\
R_Q &= W_Q(\rho) + R_{Q0} \\
\rho &= \frac{T_s}{T_{A}} \\
T_A &= \frac{T_{cl}}{p} \\
\rho &< 1
\end{align*}
\]

Assume that all clients are identical (i.e., \(T_1 = \ldots = T_n\)). The first equation is used to describe the behavior of each client that generates the next request only when the result of the previous one has been received. The behaviour of a client is cyclic: think periods (the client ideal service time \(T_p\)) alternates to wait ones (depending on the average response time of S, \(R_Q\)), leading to a certain client average interdeparture time \(T_{cl}\).

Once we know \(T_{cl}\), we can determine the server average interarrival time \(T_A\); by resorting on Theorem 3.2.2 introduced in Chapter 3, we have that \(T_A = T_{cl}/p\).

The utilization factor of the system is given by \(\rho = T_s/T_A\).

Finally, the under-load memory access latency \(R_Q\) is simply given by the average waiting time \(W_Q\) plus a constant known in advance, which is the base latency \(R_{Q0}\). The expression of \(W_Q\) depends on the type of Q. Notably, we use a M/D/1 queue \[66\], where the symbol M represents an exponential interarrival time distribution, while the symbol D represents a deterministic service time distribution, in a system with a single server (1). The service discipline is FIFO and it is assumed that the queue size is infinite. The deterministic distribution of service time is a good approximation for a memory subsystem.

For this queue, we get the following fundamental result

\[W_Q = T_S \frac{\rho}{2(1 - \rho)}\]

Therefore, solving the system with respect to \(R_Q\) leads to a second degree equation in \(\rho\). The two solutions \(\rho_1\) and \(\rho_2\) are always such that \(\rho_1 < 1\) and \(\rho_2\), thus the solution of the model must be subjected to the constrain \(\rho < 1\).

6.1.2 Complexity vs Approximation of the model

The cost model for a shared memory multiprocessor implies a complex evaluation because of the rather large number of variables, architectural variants and, perhaps most important, many different situations related to the parallel application and its mapping (as we will discuss in Section 6.2).

Our goal is to define a method which is characterized by an acceptable complexity.
and, at the same time, is able to capture the essential elements and the qualitative behavior. This implies an approximate approach based on some assumptions. For these reasons the most meaningful assumptions are:

- All conflicts are concentrated on the memory macro-modules or caches only, i.e. conflicts on the network switch nodes and links have a negligible impact. Clearly, this assumption is a simplification, anyway, the CMP internal interconnect has a very low latency which contributes to minimize the effect of network contention;

- let $T_P$ be the mean time between two consecutive accesses by the same PE to a certain memory macro-module (during this time, the processor executes instructions operating on registers or private caches). We assume that $T_P$ is the mean value of an exponentially distributed random variable. Actually this distribution depends on the parallel application characteristics, and might be different from the exponential one. However, for our purposes, the interarrival time distribution is approximated as an exponential one because of the independent behavior of the various PEs. In other words, the combination of $p$ requests can be approximately characterized by a random behavior. Thus, for the $W_Q$ evaluation a good approximation is represented by the M/D/1 queue.

- Clients are not identical. Though the majority of modules of a structured parallel paradigm are identical (workers), “service” modules (e.g., emitter and collector) are present; moreover, a parallel computation might consist of the graph composition. Thus, the identical client approximation is a worst-case approximation (e.g., service modules typically cause less memory contention).

- The server service time $T_S$ and base latency have to be estimated for the specific architecture. In this context, another modeling problem arises. Queuing Theory analytical results are valid for single sequential servers (or, when multiple servers are considered, they are merely independent sequential servers). In other words, no theoretical results exist for parallel servers. The consequence is that, for an analytical approach, we are forced to exploit formulas for $W_Q$ (the mean waiting time in queue) which in Queuing Theory have been derived for servers having equal service time and latency. Of course, in no way we can accept this assumption for a parallel server subsystem with a pipeline request-reply behavior: this is the reason for which the classical relation $R_Q = W_Q + T_S$ is extended to $R_Q = W_Q + L_S$, where $L_S$ represents the server latency (i.e., $R_{Q_0}$ in the system equations presented in the previous Section).

With these assumptions, the approximation error is about 20% when the server utilization factor $\rho$ is high (close to one), while is much lower for servers with (low-)medium $\rho$. That is, actually the method is sufficiently approximated in all cases of our main interest. Anyway, the approximation on $R_Q$ is always by-excess, thus
it is reliable for the utilization in the cost model of parallel program design and implementation [107].

6.1.3 Memory access latency

In Figures 6.2 and 6.3, the ratio between memory access latency $R_Q$ and base memory access latency $R_{Q_0}$ for reading a cache line is shown as a function of the most relevant parameters $p$ and $T_P$, for some typical combinations of $T_S$ and $R_{Q_0}$. The effect of $p$ shows the importance of the so-called “low-$p$ mapping” of parallel programs: with low-$p$ mappings the under-load latency is very close to the base latency. As is expected, the effect of $T_P$ is substantial for fine-grained computations (the utilization factor increases), while for coarse-grained computations the impact on memory conflicts tends to become negligible, so the under-load latency $R_Q$ tends to the base one for large $T_P$ values. The Figures 6.2 represent typical cases with external main memory. Depending on the network latency, we see that $p < 4$ is acceptable for medium-grained (fine-grained) computations; most important: $p \leq 2$ is acceptable even for very fine-grained computations. This confirms the concept that low contention can be achieved with good process mappings, even in the presence of relatively fine-grained computations.

6.1.4 On-chip cache-to-cache transfers

For on-chip cache-to-cache transfers the base latency is lower than the memory access case: thus, degradation is negligible for any computation grain and $p < 10$, while contention is appreciable only for higher $p$ values and very fine-grained
computations. This result shown in Figure 6.3 justifies the enormous potentials of the CMP technology and the trend towards very highly parallel CMPs.

If the value of $R_Q/R_{Q_0}$ is substantially greater than one, the parallel program bandwidth is lowered. To evaluate the contention effect on bandwidth, we must re-evaluate the cost model.

If the value of $R_Q/R_{Q_0} > 1$, all the server latencies are multiplied by $R_Q/R_{Q_0}$ with respect to the base latency evaluation.

In other words, the actual optimal parallelism degree is evaluated in function of $R_Q$

$$n_{opt}(R_Q) = \left\lceil \frac{T_{id}(R_Q)}{T_A(R_Q)} \right\rceil$$

### 6.2 On parallel program mapping and under-load evaluation

In this Section we study how to map processes belonging to the run-time support of structured parallel paradigms, in order to optimize the under-load latency of shared data structures. The main goal of process mapping is to keep the p value as low as possible: we speak about low-p mapping strategies. An efficient exploitation of the architecture is allowed by data structures shared by a relatively low number p of PEs, even if PEs are relatively distant. In the following, we will speak generically of channel descriptors to denote the shared data structures used to implements the communications.
NUMA mapping  In a pipeline process structure each channel descriptor is shared by two PEs (onto which the corresponding stages are mapped), thus $p = 2$. One of these two PEs has the channel descriptor in its local memory. Therefore the data structure is in the node onto which either the destination stage or the source stage is mapped. Each memory macro-module is shared only by the local processor and by the processor onto which the neighbor stage is mapped. The achieved $p$ value represents a very interesting result, leading to negligible contention, and so the under-load latency is very close to the base one.

The best mapping for a farm program consists in using symmetric (one-to-one) channels and allocating, for any worker, the input channel data structure (from emitter to worker) and the two output channel descriptors (from worker to emitter used to implement load balancing distribution and from worker to collector) in the local memory of PE onto which the worker itself is mapped. The emitter and collector PEs share all worker local memories, while each worker PE accesses its own local memory only. In this way we have $p = 3$ (each worker local memory accessed by the emitter, collector and worker itself PEs), which again is a very good result for sharply reducing the contention effects. If the communications from workers to emitter and to collector are expressed by asymmetric channels, channel descriptors are forced to be allocated in the emitter local memory and in the collector local memory, then $p = n + 1$, i.e. all the workers nodes share, and are in conflict on, emitter and collector local memories. More in general, the strategy “always allocate the channel descriptor in the local memory of the destination process node” is far from the optimal one even with symmetric channels (in the example $p = n + 1$).

Low-$p$ mapping strategies, exploiting symmetric channels, can be applied also to the implementation of collective communications in any data parallel paradigm (scatter, gather, multicast), as well as to collective operations (reduce), and to stencil communications. For example, for a reduce operation, allocating $n$ channel descriptors and target variables in reduce node, corresponds to the maximum contention: $p=n$. However, it can be implemented much better: $n$ channel descriptors and target variables are allocated in worker nodes. This corresponds to the minimum contention: $p=2$. Similar considerations are valid for stencil-based computations.

In conclusion, for NUMA architectures we have verified another very important feature of structured parallel paradigms: owing to the detailed knowledge of the process patterns for their run-time support, we are able to identify some efficient implementation strategies in terms of communication forms and related low-$p$ mappings.

SMP and NUMA-SMP Low-$p$ mappings are mainly achieved with a large number of interleaved macro-modules. For this reason, the trend in multiple-CMP archi-
6.2. ON PARALLEL PROGRAM MAPPING AND UNDER-LOAD EVALUATION

...tectures is towards NUMA or, more precisely, NUMA-SMP (as discussed in Chapter 4). However, in CMP-based NUMA-SMP architectures, the local memory organization is critical: a single macro-module is quite inadequate.

Shared cache and cache coherence The low-p mapping strategies, studied for NUMA architectures, must be taken into account in the design of cache coherence strategies.

In fact, the current association cache coherence - SMP architecture is misleading. Independently of the multiprocessor architecture class (and of the possible existence of shared caches in CMPs), the basic semantics of cache coherence is close to the NUMA idea: cache lines are allocated and controlled locally to PEs, cache-to-cache transfers and invalidation (or update) communications occur between PrC. In other words, from the under-load modeling point of view, caches behave as shared local memories of a NUMA machine. Of course, the same directory-based approach is typically NUMA-oriented.

The above consideration is also important from the performance evaluation viewpoint: automatic cache coherence protocols might be a potential source of performance degradation, because they might prevent or hinder low-p mapping strategies. For example, home nodes might become bottlenecks, or external communications (from distinct PEs) for cache coherence actions like C2C transfers or invalidations/updates requests, increase the request traffic in input to PrC servers, in addition to the internal requests of the same PE.

As usual, these problems are too complex if studied independently of the parallel program’s characteristics. However, the set of possible mappings is narrowed by the implementation of structured parallel paradigms and their run-time supports.

Notably, our idea of a solution is in proper NUMA mappings, in particular for process exclusive mapping approach and a specific run-time support that aims to provide a low-p mapping.

For example, in the farm paradigm, a low-p mapping is implemented if the local memory of a worker node contains all the worker channel descriptors: the input channel *task* (Emitter to Worker) and output channels *available* and *result* (respectively, Worker to Emitter and Worker to Collector), as well as the target variables of Worker, Emitter and Collector.

Thus the worker node is the home node of the shared blocks comprising all these channel descriptors and target variables.

However, if the *task* channel run-time support is implemented according to the basic automatic cache coherence solutions, the C2C read requests and invalidation requests from workers’ nodes to Emitter node re-introduce a high-p contention on the Emitter PrC. That is, low-p mappings would require that workers do not perform requests to the Emitter node.

This is possible by a proper combination of basic optimization cache coherence strategy in the implementation of interprocess communication, as we will discuss in the
next Chapter. The solution focus is: communication channel task is implemented according to the *home-flush strategy*, where the send, executed by Emitter, modifies the channel descriptor and the associated target variable through synchronous flush operations, which invalidate the blocks in the Emitter cache locally, thus avoiding both the C2C read requests and the invalidation communications from the worker node in receive/compute phases. Again, the only servers are workers. On the other hand, since the worker is the home node of the interested blocks, the modified channel descriptor and local variable are automatically copied into the worker $PrC$ by the send run-time support. These considerations apply perfectly to data-parallel programs too. This kind of design optimization is possible for multiprocessor systems which provide flexible mechanisms for cache coherence strategies, like *home node selection* and flush operation with local de-allocation.

### 6.3 PEPA: Process Algebra for Quantitative Analysis

Performance Evaluation Process Algebra (PEPA) \[55\] is a high-level description language for Markov processes which belongs to the class of Stochastic Process Algebras (SPA). Among the wide class of SPAs, we choose PEPA because it is simple but at the same time it has sufficient expressiveness for our purposes. The simplicity comes from the structure of the language: PEPA has only a few elements and a formal interpretation of all expressions can be provided by a structured operational semantics.

In this section we just introduce the minimal set of PEPA features strictly necessary to model client-server with request-reply behavior. The interested reader can refer to \[55\] for a detailed description.

First, in Section 6.3.2 we briefly describe how PEPA can be used to perform a performance analysis of graph computations, including graphs with multiple sources, in order to detect bottlenecks in the computation. As introduced in the first part of this thesis, this is a fundamental part of our methodology.

Finally, we focus on the possibility to describe different kind of CMP-based architecture with a PEPA program. Notably, we start from a simple processor-memory architecture up to CMPs with a complex cache hierarchy and cache coherence protocols. This allow us to compare the observations and the analysis done until now about parallel programs’ performance with the results obtained from a high-level description of parallel applications executed on different kind of platforms.

#### 6.3.1 PEPA Language

A PEPA system is described as a composition of components that undertake actions. Components correspond to identifiable parts in the system. For instance, in
6.3. PEPA: PROCESS ALGEBRA FOR QUANTITATIVE ANALYSIS

<table>
<thead>
<tr>
<th>Prefix</th>
<th>$$(\alpha, r).E \xrightarrow{(\alpha, r)} E$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choice</td>
<td>$E \xrightarrow{(\alpha, r)} E' \quad E + F \xrightarrow{(\alpha, r)} E'$</td>
</tr>
<tr>
<td>Cooperation</td>
<td>$E \xrightarrow{(\alpha, r)} E' \quad F \xrightarrow{(\alpha, r)} F'$</td>
</tr>
<tr>
<td>Hiding</td>
<td>$E \xrightarrow{(\alpha, r)} E' \quad E \xrightarrow{(\alpha, r)} E' / L$</td>
</tr>
<tr>
<td>Constant</td>
<td>$A \xrightarrow{(\alpha, r)} E'$</td>
</tr>
</tbody>
</table>

Prefix

$$(\alpha, r).E \xrightarrow{(\alpha, r)} E$$

Choice

$E \xrightarrow{(\alpha, r)} E' \quad E + F \xrightarrow{(\alpha, r)} E'$

Cooperation

$E \xrightarrow{(\alpha, r)} E' \quad F \xrightarrow{(\alpha, r)} F'$

Hiding

$E \xrightarrow{(\alpha, r)} E' \quad E \xrightarrow{(\alpha, r)} E' / L$

Constant

$A \xrightarrow{(\alpha, r)} E'$

Figure 6.4: Structured Operational Semantic of PEPA
our context, clients and servers will be the components of the systems. A component may be atomic or may itself be composed by components. The language is indeed compositional in the sense that new components may be formed through the cooperation of other ones. Each component can perform a finite set of actions. An action has a duration (or delay) which is a random variable with an exponential distribution. Consequently, the rate of the action is given by the parameter of the exponential distribution. For example, the expression

\[ P \triangleq (\alpha, r).Q \]

represents the definition of a new component P which can undertake an action \( \alpha \) at rate \( r \) to evolve into another component Q (defined somewhere else). Since the duration of all actions of the system are exponentially distributed, it is intuitive to say that the stochastic behaviour of the model is governed by an underlying CTMC (continuous-time Markov chain).

The syntax of the PEPA language is formally defined by the following grammar:

\[
S ::= (\alpha, r).S \mid S + S \mid C_S
\]

\[
P ::= P \triangleleft_\bot P \mid P/L \mid C
\]

\( S \) denotes a sequential component and \( P \) denotes a model component which executes in parallel. \( C \) and \( C_S \) stand for constants to denote either a sequential or a model component (the effect of the syntactic separations is to allow to build only components which are cooperations of only sequential components, which has been proved in [55] to be a necessary condition for building ergodic Markov processes, i.e. amenable to steady-state analysis).

The structured operational semantic is shown in Figure 6.4. Below an intuitive description of the most used PEPA operators is provided. The interested reader can refer to [55] for a detailed description.

- **Prefix** \( ((\alpha, r).P) \) This is the basic mechanism to express a sequential behavior in PEPA. As already said, a component performs an action \( \alpha \) at rate \( r \) behaving subsequently as \( P \).

- **Choice** \( (P+Q) \) This operator represents a component that may behave either as \( P \) or as \( Q \). Assume that \( \alpha \) and \( \beta \) are the actions that enable respectively \( P \) and \( Q \), characterized by their own rate. The idea behind the Choice operator is that once an action has been completed, the other is discarded. For instance, if the first action to be completed is \( \beta \) then the component moves to \( Q \), “forgetting” the other branch.

- **Cooperation** \( (P \triangleleft_\bot Q) \) This operator denotes the cooperation between \( P \) and \( Q \) over \( L \). \( L \) is the cooperation set that contains those activities on which the components are forced to synchronized. The rate of this shared activity has
to be altered to reflect the slower component in the cooperation (see how in Figure 5.1). It is important to notice that for actions not in L components proceed independently and concurrently with their enabled activities. Actually cooperation is a multi-way synchronization since more than two components are allowed to jointly perform actions of the same type. When concurrent components do not have to synchronize the cooperation set L is empty; in these cases we will use the abbreviation $P \parallel Q$ to denote $P$ and $Q$ running in parallel. We will use also a simple syntactic shorthand to denote an expression like $(P \parallel P \parallel \ldots \parallel P)$ as $P[N]$, with $N$ the number of times that $P$ is replicated. Finally, we point out that there can be situations in which two components do synchronize, but the rate of the shared activity is determined by only one of the component in the cooperation. In this case the other component is defined as passive. The rate of the activity for the passive component will be denoted with the symbol $\top$.

### 6.3.2 PEPA for graphs

An interesting application of the PEPA language for our purpose, is the analysis of the steady-state behavior of a computation graph. Especially, in the case of graphs with multiple-sources, which are cases not covered by the methodology presented in [77] and discussed in the Introduction part of this thesis.

For example, consider the computation graph in Figure 6.5. We can approximate the result of steady-state analysis using PEPA with the results shown in Table 6.1. The table shows for each node of the graph, the service time (in the second column) and the result of the steady-state analysis in terms of the effective interdeparture time of each module and its utilization factor.

Therefore, PEPA could be an useful and simple tool used to perform the analysis of the graph computation in the bottleneck detection phase described in Section 3.2.
6.3.3 PEPA for under load memory latency

Another interesting application of this tool is that a PEPA program for the classical client-server model with request-reply behaviour can be instantiated to model a processors-memory system just knowing the following parameters: $T_p$, $T_s$, $p$, and the latencies relative to the message request and response sent through the network interconnection, respectively $T_{req}$ and $T_{resp}$. The resulting PEPA program is shown below.

\[
\begin{align*}
\text{Client}_{\text{think}} & \overset{\text{def}}{=} (\text{request}, r_{\text{request}}).\text{Client}_{\text{wait}} \\
\text{Client}_{\text{wait}} & \overset{\text{def}}{=} (\text{reply}, \top).\text{Client}_{\text{think}} \\
\text{Server} & \overset{\text{def}}{=} (\text{request}, \top).\text{Server} + (\text{reply}, r_{\text{reply}}).\text{Server} \\
\text{Client}_{\text{think}}[p]_{\text{reply}, r_{\text{request}}} & \Join \text{Server}
\end{align*}
\]

Each client models a process (on a processing node) that operates forever in a simple loop, completing in sequence the two phases think and wait. As already stated, the length of the think phase is $T_p$. At the end, a request action is executed and the client waits for a reply, i.e. it starts the wait phase. The request action is a shared action between the clients and the server. It models the situation in which a client sends a request and the server receives it. The length of the wait phase of the client is $R_Q$. For this reason, the time needed to complete the reply action (phase wait) is initially unspecified. In fact, it will be imposed in another PEPA expression through the cooperation with another component. Therefore, Client components see reply as a pure synchronization operation.

The server modeling the memory macro-module can either accept a request from one of the $p$ clients (action request) or send them a reply. The time to complete a request action is obviously unspecified because it depends on clients. The action reply is shared to model the fact that a client can go back to the think phase as soon

<table>
<thead>
<tr>
<th>node</th>
<th>service time</th>
<th>interdeparture time</th>
<th>utilization factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>25</td>
<td>127.52</td>
<td>19.60</td>
</tr>
<tr>
<td>M2</td>
<td>30</td>
<td>123.04</td>
<td>24.38</td>
</tr>
<tr>
<td>M3</td>
<td>40</td>
<td>78.63</td>
<td>50.87</td>
</tr>
<tr>
<td>M4</td>
<td>25</td>
<td>307.60</td>
<td>8.13</td>
</tr>
<tr>
<td>M5</td>
<td>150</td>
<td>262.09</td>
<td>57.23</td>
</tr>
<tr>
<td>M6</td>
<td>15</td>
<td>112.33</td>
<td>13.35</td>
</tr>
<tr>
<td>M7</td>
<td>27</td>
<td>120.97</td>
<td>22.32</td>
</tr>
<tr>
<td>M8</td>
<td>120</td>
<td>129.83</td>
<td>92.43</td>
</tr>
</tbody>
</table>

Table 6.1: PEPA steady-state resolution of a multiple-source computation graph
as the server has handled its request. Finally, the last expression instantiate a client-server model with \( p \) clients running in parallel that try to synchronize themselves with the server through the cooperation set containing both the two shared actions request and reply.

It is useful to highlight that even simpler solutions could be formalized: for instance, the synchronization on the action request is not strictly necessary. However, we decided to keep it for two reasons. First, it helps to understand the semantic of the whole system (the “request-reply behaviour”). Second, it will be necessary anyway in further extensions of this basic model.

**Model resolution** Once found, steady-state information is exploited to derive the average response time \( R_{Q_{server}} \) of the server. In particular, these information includes:

- the average population in each state of the underlying CTMC
- the throughput of the actions

In our client-server model we are interested in the average number of clients that reside in state Clientwait (\( p_{\text{wait}} \)) and in the throughput of the action reply (\( \lambda_{\text{reply}} \)). Indeed, by applying Little's Law [66], we can extract the average time that a client stays in the state Clientwait, which actually corresponds to \( R_{Q_{server}} \):

\[
R_{Q_{server}} = \frac{p_{\text{wait}}}{\lambda_{\text{reply}}}
\]

It is extremely important to notice that \( R_{Q_{server}} \) is not the under-load memory access latency, but it is the average time spent by a request at the server. However, to find out \( R_Q \) it is enough to take into account the base latency of the network as in the following equation.

\[
R_Q = T_{req} + R_{Q_{server}} + T_{resp}
\]

**Heterogeneous Clients in PEPA**

In order to model different types of client, the PEPA program can be expressed as follows, where there are \( p_i \) clients of type \( \text{Client}_i \) for \( i = 1, \ldots, C \):
Thanks to the compositional approach of PEPA, we can directly reuse the same server component and the definition of a generic client already defined before. So basically, a generic client has the same behaviour as before and this implies that it is unnecessary to add further operations apart from the already used request and reply. As a consequence of this structured approach, also the cooperation set in the last expression remains the same. Of course, a change occurs in the number of client definitions. In fact, we want to apply the theory seen above in order to recognize C classes of processes. According to the theory, we do not want to have a definition per client but, in order to keep lower the resolution complexity, there must a number of client definitions equal to C. Every definition has own rate of request, that is peculiar for that given class. This rate is the inverse of the $T_P$ characterizing the class and it has been found according to the techniques explained in the previous section, i.e. by profiling in the easiest cases or using the explicit phases technique. The last expression of the program defines the overall system in which clients of C classes run in parallel synchronizing themselves with the server. Obviously, each class of clients specifies the number of clients belonging to that class.

**Model resolution** Following the procedure of the previous section, we have to find $R_{Q_{server}}$ in order to evaluate the under-load memory access latency. Having more wait states, i.e. one for client definition, we can evaluate the average number of clients staying in the state $Client_{wait}$ as

$$R_{Q_{server}} = \sum_{i=1}^{c} \frac{p_{wait_i}}{\lambda_{reply}} = \frac{\sum_{i=1}^{c} p_{wait_i}}{\lambda_{reply}}$$

(6.2)

where $p_{wait_i}$ is the average number of clients belonging to the state $Client - i_{wait}$ in the steady-state condition of the system. Successively, it is sufficient to add the base network latencies for the request and for the reply to obtain the under-load memory access latency as in 6.1.
Hierarchical Shared Memory

In order to model the impact of various memory and cache hierarchy levels, the model can be extended as follow. For instance, suppose that the number of requests satisfied by \( PrC \) is \( c \), while \( m \) is the number of requests satisfied by \( M \). Let \( pc \) be the probability to satisfy a request in \( PrC \) and \( pm \) the probability to satisfy a request in \( M \), we have:

\[
pc = \frac{c}{c + m} \quad pm = \frac{m}{c + m}
\]

The idea is to model processing nodes as clients able to generate requests toward either \( PrC \) or \( M \). In other words, this means to have clients that can choose between two different actions, that are \( request_C \) or \( request_M \).

We know from [56] that we can model this situation with a component engaging in an action (sends a request with mean duration \( 1/T_p \)), which may have two different possible outcomes resulting from the action (in our case sends a request to \( PrC \) or to \( M \)). The client component that performs this single action (sends a request) would be represented by two separate activities (\( request_C \) and \( request_M \)). The activity rates of these activities would be adjusted to capture the probabilities of the different outcomes.

\[
r_C = \frac{pc}{Tp} \quad r_M = \frac{pm}{Tp}
\]

\[
P \overset{\text{def}}{=} (request_c, r_C).P_{wait_C} + (request_m, r_M).P_{wait_M}
\]

\[
P_{wait_C} \overset{\text{def}}{=} (reply_M, \top).P
\]

\[
P_{wait_M} \overset{\text{def}}{=} (reply_M, \top).P
\]

\[
Cache \overset{\text{def}}{=} (request_C, \top). (reply_C, r_{\text{cache}}). Cache + (request_M, \top). (forward, r_{\text{forward}}). Cache
\]

\[
Memory \overset{\text{def}}{=} (forward, \top). Memory + (reply_M, r_{\text{memory}}). Memory
\]

\[
P[n]_\text{request}_C,\text{request}_M,\text{reply}_C \Join Cache[c] \Join Memory[m]
\]

Model resolution The way to evaluate the under-load memory access latency in steady state condition of the system is basically the same as in the previous cases for both versions. Again, we use Littles law [66] to find out the so-called \( R_{Q_{\text{server}}} \).

The difference lies in having more waiting states and more incoming rates to those states. We can easily adjust Formula 6.2 in this way:

\[
R_{Q_{\text{server}}} = \sum_{i=1}^{w} \frac{p_{\text{wait}_i}}{\lambda_{\text{reply}_i}} \quad (6.3)
\]
where \( p_{wait_i} \) is the average number of clients belonging to the state of \( P_{wait_i} \) in steady-state condition of the system and \( w \) is the number of waiting states. Finally, as usual, we have to add the impact of interconnection structures for having the under-load memory access latency. Anyway, in this case it is important to recall that more interconnection structures can be involved in hierarchical shared memory architectures. The best solution is to consider again interconnection structures log-
ically belonging to the server subsystem with the difference that the base network latencies \( T_{req} \) and \( T_{resp} \) are evaluated applying the definition of mean value:

\[
T_{req} = T_{reqPC} \cdot pc + T_{reqCM} \cdot pm \\
T_{resp} = T_{respPC} \cdot pc + T_{respCM} \cdot pm
\]

**Shared level caches and cache coherence**

Finally we can extend the previous models in order to measure the impact of shared level caches and the interactions between the various clients and servers due to the cache coherence protocols.

We developed PEPA models corresponding to some of the abstract models defined in Section 4.2:

1. a single-CMP, with a shared level cache that acts as GC;
2. a single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs;
3. a single-CMP, multiple-MINF, with a shared level cache that acts as GC;

In particular, we show the results of PEPA models used to evaluate the effect of the low-\( p \) mapping strategy. To study this problem, for each model we defined:

- for **standard** mapping (\( p=n \)):
  - a client INPUT that represents the emitter or the input module of a farm or data-parallel paradigms respectively;
  - a set of clients \( W[n] \) which represent the worker modules, which perform requests to \( PrC \), \( ShC \) (when present), \( M \) and to the remote INPUT private cache \( PrC_{INPUT} \);

- for **low-\( p \)** mapping:
  - a client INPUT that represents the emitter or the input module of a farm or data-parallel paradigms respectively, which performs requests to the various remote \( W[i] \) private caches \( PrC_i \);
  - a set of clients \( W[n] \) which represents the worker modules, which perform requests to \( PrC \), \( ShC \) (when present) and \( M \).
The following PEPA model represents the case of a single-CMP, with a shared level cache that acts as GC with standard mapping (p=n):

\[
W \overset{\text{def}}{=} \quad \text{(req}_\text{Prc}, r_\text{PrC}).W_{\text{Prc}}\_\text{wait} + \\
\quad \text{(req}_\text{ShC}, r_\text{ShC}).W_{\text{ShC}}\_\text{wait} + \\
\quad \text{(req}_\text{rem}, r_\text{rem}).W_{\text{Rem}}\_\text{wait} + \\
\quad \text{(req}_\text{M}, r_\text{M}).W_{\text{M}}\_\text{wait}
\]

\[
W_{\text{Prc}}\_\text{wait} \overset{\text{def}}{=} \quad \text{(reply}_\text{Prc}, \top).W
\]

\[
W_{\text{ShC}}\_\text{wait} \overset{\text{def}}{=} \quad \text{(reply}_\text{ShC}, \top).W
\]

\[
W_{\text{Rem}}\_\text{wait} \overset{\text{def}}{=} \quad \text{(reply}_\text{rem}, \top).W
\]

\[
W_{\text{M}}\_\text{wait} \overset{\text{def}}{=} \quad \text{(reply}_\text{M}, \top).W
\]

\[
\text{PrC} \overset{\text{def}}{=} \quad \text{(req}_\text{Prc}, \top).\text{(reply}_\text{Prc}, 1.0/t\text{PrC}).\text{PrC} + \\
\quad \text{(req}_\text{ShC}, \top).\text{(req}\text{ShC}, 1.0/t\text{sw}).\text{PrC} + \\
\quad \text{(reply}\text{ShC}, \top).\text{(reply}_\text{ShC}, 1.0/t\text{PrC}).\text{PrC} + \\
\quad \text{(req}_\text{rem}, \top).\text{(req}\text{rem}, 1.0/t\text{sw}).\text{PrC} + \\
\quad \text{(reply}\text{rem}, \top).\text{(reply}_\text{rem}, 1.0/t\text{PrC}).\text{PrC} + \\
\quad \text{(req}_\text{M}, \top).\text{(req}\text{M}, 1.0/t\text{sw}).\text{PrC} + \\
\quad \text{(reply}\text{M}, \top).\text{(reply}_\text{M}, 1.0/t\text{PrC}).\text{PrC}
\]

\[
\text{GC} \overset{\text{def}}{=} 
\quad \text{(req}\text{rem}, \top).\text{(l2}\_\text{forward}, 1.0/t\text{lookup}).\text{GC}
\]

\[
\text{ShC} \overset{\text{def}}{=} 
\quad \text{(req}\text{ShC}, \top).\text{ShCserve} + \text{(req}\text{M}, \top).\text{ShClookup}
\]

\[
\text{INPUT} \overset{\text{def}}{=} 
\quad \text{(local}\_\text{stuff} , r\_\text{in}\_\text{loc}).\text{INPUT}_{\text{local}} + \\
\quad \text{(l2}\_\text{forward}, r\_\text{in}\_\text{rem}).\text{INPUT}_{\text{remote}}
\]

\[
\text{INPUT}_{\text{local}} \overset{\text{def}}{=} 
\quad \text{(local}, 1.0/t\text{PrC}).\text{INPUT}
\]

\[
\text{INPUT}_{\text{remote}} \overset{\text{def}}{=} 
\quad \text{(reply}\text{rem}, 1.0/t\text{Rem}).\text{INPUT}
\]

\[
M \overset{\text{def}}{=} 
\quad \text{(m}\_\text{access}, \top).\text{Mserve}
\]

\[
\text{Mserve} \overset{\text{def}}{=} 
\quad \text{(reply}\text{M}, 1.0/t\text{M}).\text{M}
\]

\[
W[n] \overset{\text{def}}{=} \text{PrC}[n] \overset{\text{def}}{=} \text{INPUT} \overset{\text{def}}{=} \text{GC} \overset{\text{def}}{=} \text{M}
\]

Figures 6.6 and 6.7 shows the evaluation of $R_Q$ for each server in the hierarchy:

- $\text{PrC}$, the private level cache of W
- $\text{ShC}$, the shared level cache
- $\text{INPUT}$, the private level cache of INPUT accessed by W
- $W(\text{INPUT})$, the private level cache of W accessed by INPUT
- $M$, the main memory
6. COST MODELS FOR CMP-BASED ARCHITECTURES

(a) $p_{IN} = n + 1$

(b) $p_{W[i]} = 2$

Figure 6.6: Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p = 2$) vs $p = nw + 1$ in Single-CMP with GC implemented at the shared cache level

Figure 6.6 shows the case of a single-CMP, with a shared level cache that acts as GC, while Figure 6.7 show the case a single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs. In both cases, with

(a) a standard mapping strategy, where each W[i] accesses the INPUT private level cache ($p_{IN} = n + 1$)

(b) a low-p mapping strategy, where each W[i] accesses its private level cache and INPUT accesses the various W[i] private level caches ($p_{W[i]} = 2$)

Figures 6.8 and 6.9 shows the corresponding value of $R_Q/R_{Q0}$.

Finally, we measure the impact of multiple-MINFs on chip in a single-CMP, with a shared level cache that acts as GC. The results justify the high value of $R_Q/R_{Q0}$ especially in the corresponding model with $m_M = 1$. The corresponding results are shown in 6.10.

All these results, confirm the results anticipated in the previous section of this Chapter, and we can use it as a good validation for them especially because here we are able to take into account the whole system, with different level of servers that represents each level of the memory/cache hierarchy.

6.3.4 On the resolution of PEPA models

Solving a PEPA model means solving the underlying ergodic CTMC, i.e. computing the steady-state. We wrote and solved PEPA models using the eclipse-plugin for PEPA [1]. This tool provides a lot of different numerical resolution techniques to solve the model. Different techniques can be employed depending on the size of the resulting CTMC: if the number of states is huge (hundreds of thousands) iterative
Figure 6.7: Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs

Figure 6.8: Comparison with PEPA of $R_Q/R_{Q_0}$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ in Single-CMP with GC implemented at the shared cache level
Figure 6.9: Comparison with PEPA of $R_Q / R_{Q_0}$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ a single-CMP, single-MINF, without a shared level cache, with GC distributed among the PrCs

Figure 6.10: Comparison with PEPA of $R_Q$ of each server: low-p mapping strategy ($p_{W[i]} = 2$) vs $p_{IN} = n + 1$ a single-CMP, multiple-MINF, with a shared level cache that acts as GC
yet approximate techniques are preferred. However, the models that we treat are extremely small, thus the steady-state has been directly computed employing a very standard algorithm (i.e., direct solver or conjugate gradient solver for calculating the steady-state probability distribution). In all other cases, e.g. when the number of clients significantly grow, a phenomenon known as state space explosion may arise. However, thanks to the natural structure of our models, we may take full advantage of both state-reduction and fluid-approximation techniques [55]. Briefly, these techniques aim to solve the state space explosion by exploiting potential symmetries in the CTMC. The presence of symmetries can be informally deduced looking at the PEPA expressions: for instance, in our model the set of homogeneous clients (Client[p]) induces replicated sub-Markov chains in the underlying CTMC. These replicated subsystems will be exploited to restructure the CTMC itself and lower the state space size.

6.4 Summary

In this chapter, we deal with the problem of how to evaluate the under-load memory latencies in order to complete the evaluation of the performances of parallel applications executed on CMP-based systems.

In the first part, we used the results of Queueing Theory for a client-server model. Notably, we applied the work proposed in [107], to model under-load memory and cache access latencies in cache coherent architectures.

The results obtained are used to reason about the effect of under-load latencies to parallel program performances. Notably, we discuss the impact of specific choices (e.g., parallel process mapping and cache coherence optimizations) in the run-time support of parallelism forms.

Finally, we used an interesting performance evaluation tool: Performance Evaluation Process Algebra (PEPA), in order to evaluate the base memory and cache latencies defined in Chapter 4, by describing a complete abstract architecture (i.e., with different level of servers for each level in the memory hierarchy) and deriving a complete cost model for various abstract models. With this evaluations we have the confirmation of the impact of the mapping strategies in the performance of parallel program.

Of course, alternative approaches, based on simulation and/or experimental evaluation, are helpful during some design and evaluation phases: for example, a good queuing network simulator of parallel architectures is described in [23].

According to the results obtained, we are able to study the performance of alternative run-time support solutions with different approaches to cache coherence, notably, evaluating the advantages of specific optimization (i.e., low-p mapping strategies).
6. COST MODELS FOR CMP-BASED ARCHITECTURES
Part III

Evaluation of the Proposed Methodology
All the considerations made in the second part of the thesis, are now used to provide an optimized run-time support for structured parallel applications. Our approach aims to design a run-time support for advanced CMP-based multiprocessors. Notably, this chapter, gathers our research group efforts in the implementation of run-time support for CMP-based architectures. Many of the concepts presented here are yet published \[24, 25, 107\] and during this thesis were further refined. In particular, this thesis represents a contribution in the design and study of parallel paradigms’ run-time support with optimization strictly related to the cache coherence problem and its impact in the parallel performance applications.

In the final part of this Chapter we discuss the implementation of the optimization discussed in this chapter on the Tilera TilePro64 processor. This architecture represents a good candidate for the evaluation of our solutions, due to the possibility of implementing or emulating our ideas.

Notably, with this architecture we are able to achieve an improvement of about 50% with respect to the use of the default cache coherence solution.

### 7.1 Optimizations for Parallel Paradigms Run-time Support

In this Section we analyze some relevant optimization techniques that can be used in the development of the run-time support of parallel paradigms. The write operation implementation is a key issue from the performance point of view, in particular for the contention effects on memory modules or caches. As introduced in the previous Chapter, because of the cache coherence protocols, \(PrC\) and \(ShC\) may be regarded as servers in the multiprocessor queueing model: the read/write operations imply
client-server relationships between PEs and caches (e.g., for C2C cache line transfers, write synchronous notifications, and invalidations with acknowledgment). In order to minimize the server utilization factor, the protocol interactions should be designed accurately.

In this Section we describe and evaluate optimization techniques which are presented in, or are an equivalent model of, many systems. In the following we refer to directory-based invalidation-based architectures, though the discussed techniques are valid more generally. Notably, in the following we refer to $PE_{\text{home}}$ or simply home node for a specific cache line as the PE in whose main local memory the line is allocated. If the architecture is not strictly NUMA, it is the PE in charge of controlling a given partition of blocks ($PE_{\text{home}}$ implements the GC). The home node is able to serve a cache line request directly via cache-to-cache transfers, when the line is currently present (thus, valid) in its local cache, or by transferring the requested line from the main (local) memory, if the cache line is not modified in another PE.

### 7.1.1 Flexible home node selection

A first general problem is the global synchronization implied by invalidation (or updating).

In any cache coherence solution, if $n_{sh} \geq 1$ denotes the number of copies to be invalidated, the write operation could have a cost which grows proportionally with respect to this number.

As already discussed in Chapter 5, the invalidation notifications and acknowledgment are performed in parallel; however they contribute to increasing contention and in the case of synchronous writes (e.g., for solving the memory ordering problem) the corresponding cost cannot be entirely overlapped. Thus, the minimization of current copies of the same block is a must in parallel program design.

Often, according to the specific problem semantics and/or to the design strategy, it is possible to recognize a very limited number of copies (one copy at most) to be invalidated: this is a powerful feature of some structured parallel paradigms. For example, in farm or data-parallel computations the input and output channel data structure used by each worker are shared only between the worker itself and the emitter or the collector, limiting the number of copy to be invalidated to one.

Notably, proper strategies for the home node selection can be useful, provided that flexible mechanisms are provided for this purpose.

### 7.1.2 Home-flush techniques

In almost all write operations (i.e., except when it is possible to perform the write locally), the home node is informed/involved by the cache coherence protocols. When the home node does not coincide with the requestor node, which performs the write operation an alternative write operation implementation can be provided.

We called this technique home-flush, because is characterized by the de-allocation
of the referred cache line from the requestor node $PrC_r$ and the whole line is sent or flushed to the home node. The home node uses this data to update the local main memory when necessary ($PE_r$ and $PE_{home}$ have distinct local main memory) and the $PrC_h$ through a cache-to-cache write request communication. Thus, we assume the presence of a special instruction flush with the above semantic.

A synchronous version of the flush instruction can be provided in order to easily solve any memory ordering problems, as discussed in Chapter 5.

The use of this operation reduces home node latency and contention, by avoiding to involve $PE_r$ (which does not hold anymore a copy of the cache line) in subsequent read/write operations executed by the home node itself or from other PE on the same cache line. In other words, the block flush advantage is not only latency saving (though relevant): more important, contention on $PrC$ is reduced, as well as the global synchronization implied by invalidation.

Block flush can be used as an alternative mechanism to invalidation in write execution. When provided, this mechanism represents an important optimization. Similar solutions have been studied in the literature [27, 56], evaluating the advantages of this technique in producer-consumer patterns.

In some architectures the flush mechanism, including the de-allocation effect, is associated to an entire data structure, instead of to a single cache line, which can be even more powerful. For example, the Tilera TilePro64 provides this instruction for the management of software cache coherence, flushing the data to the main memory. While, a pure flush mechanism between $PrCs$ is not explicitly provided, it can be emulated with the write-through semantics of write operation between a requestor node and the home node. In other cases [95] the term flush is used in a different way, meaning that the whole content of one or more cache levels is copied into the main memory: this mechanism has a quite different semantics and it is not of interest for the ensuing discussion.

We can evaluate the cost of this mechanism, considering the use of the cache-to-cache facilities in the abstract model presented in Chapter 4.

A cache-to-cache write request $c2c_write_req$ is composed of $\sigma + 1$ words sent from $PE_r$ to $PE_{home}$ through the interconnection network and $PE_{home}$ sends back an acknowledgment $ack_{c2c}$ answer of few words (1 or 2) through the interconnection network after the data are written in its $PrC_{home}$ and when necessary written back to the local main memory. This write back can be done of course in parallel and the cost possibly overlapped. Therefore, we can evaluate the cost as follows

$$L_{writeC2C}(M, -, -) = T_{net}(\sigma) + T_{PrC} + [T_M] + T_{net}$$
7.1.3 Cooperation mechanisms among cores through inter-processor communications

An interesting aspect of modern CMP architectures, notably (but not just) network processors, is the availability of very specific architectural structures that can be exploited to speed up inter-core cooperation, such as the presence of user-accessible on-chip core-to-core interconnection networks. These networks, referred to as *Messaging Networks*), are used to exchange messages containing packet descriptors, i.e. the packet headers and the initial memory address of the packet in memory. The transmission of packet descriptors is performed over the messaging network by skipping all the shared memory and cache hierarchy levels, thus exploiting the on-chip interconnection to limit the memory contention by sharply reducing the communication latency. Bus, Ring or Mesh networks are available on modern architectures like Broadcom XLP [82], Cavium [59] and Tilera TilePro64 [19].

These mechanisms can be used for our purposes to implement lightweight cooperation mechanisms among cores through inter-processor communications. In particular, this kind of communication is performed asynchronously with respect to the execution of read and write operations. We can consider that the basic communication corresponds to the sending of a message composed of few words \( i \) (e.g., 1 – 4), which will be transmitted to the destination PE in an interrupt message. Therefore, we can evaluate the latency of an inter-processor communication of \( i \) words in terms of the network latency, as follow

\[
L_{\text{IP}} = T_{\text{net}}(i)
\]

This latency, in modern CMP architectures is comparable to the \( PrC \) access latency for single-CMP architectures with a low-latency on-chip interconnection network.

7.2 Communication run-time support

Standard lock-based run-time supports are based on symmetric mutual exclusion of shared data structures, as discussed in Chapter 5. This approach is a generic one, valid for any architecture with *multiprogrammed* mapping and classical low-level scheduling with *passive waiting*. It is typically oriented to the execution of concurrent jobs, not necessarily parallel or highly-parallel, including sequential or concurrent applications and concurrent operating system services. The use of symmetric lock-based techniques is a limitation for low-latency communications, though the design exploits some notable optimizations, notably user-space implementation and communication overlapping. This scheme can be implemented with exclusive mapping too, replacing the low-level scheduling sections with busy waiting synchronization.

Our approach is an optimized, inherently lock-free version, entirely based on asymmetric notify-based Rdy-Ack synchronization for *exclusive mapping* processes. We
aim to design an optimized run-time support for advanced CMP-based multiprocessors with exclusive mapping. The exclusive mapping approach is oriented to single highly parallel programs, in particular structured parallel programs, which exploit the whole PE set. The target is low-latency interprocess communication, possibly associated to communication overlapping. In this way, we are able to apply the optimizations introduced in the previous Section (which derive from the knowledge of parallel paradigms’ structure) to the run-time support of the parallel application. In this Section we define and evaluate this latter approach. Notably, the optimization techniques introduced in the previous section are exploited to define an algorithm-dependent solution to the cache coherence problem in current CMP-based architectures.

7.2.1 The Rdy-Ack Communication Model

We start with a very basic mechanism, that we call Rdy-Ack communication, used by PEs to synchronize and exchange messages. This mechanism provides a point-to-point communication between two partners, sender (S) and receiver (R), with a buffer of one position (vtg). S and R exploit two primitives send and receive that implement the communications as summarized in Figure 7.1.

```
Figure 7.1: Abstract definition of the send-receive operations in the rdy-ack communication model

The pseudo-code of the primitives uses two boolean events:

- the ready (RDY) event, which specifies the presence of a new message, and
- the acknowledgment (ACK) event, which represents the reception of the last transmitted message.

With the signal operation, the corresponding event is set to true, while the reset one sets the event to false. To ensure correctness, the RDY and the ACK events are
initialized to false and true respectively. As discussed in Chapter 5, in a message-passing implementation model, S and R use the send and receive operations to exchange messages, while in a passing-pointer approach the message copied into vtg can be a memory pointer to a shared data structure, exchanging data structures by reference. In the following we refer in both case to vtg as the exchanged unit and, when necessary, we specify if vtg represents either a set of words (a copy of the entire message) or a single machine word representing a memory pointer.

### 7.2.2 Rdy-Ack Based on Shared Memory Synchronizations

To understand the principle which underlies this solution we start with a first implementation of the rdy-ack communication model which consists in a symmetric communication mechanism based on shared memory variables. We define a VTG data structure in order to associate to the target variable vtg the corresponding RDY and ACK events, which are implemented by two boolean flags (initialized to 0 and 1 respectively). Figure 7.2 shows the VTG data structure and the send and receive operations on a rdy-ack communication based on shared memory variables (ra_sm), where the waiting of an event is implemented by a while-loop on the corresponding flag.

Correctness of the Send-Receive Algorithms

Let us consider an abstract multi-processor architecture \( M \) respecting the Sequential Consistency memory model (Section 5.2.3). Accordingly, load/store instructions of the same processor are executed in the program order and they can be interleaved with instructions of different processors in any sequential order. In this case, the following proposition is valid.
Proposition 7.2.1. The send and receive algorithms executed on $M$ implement a lock free single-producer single-consumer shared buffer of one position.

Proof. Initially $(\text{RDY}, \text{ACK}) = (0,1)$. This means that the sender can proceed by executing the send while the receiver is eventually waiting on line 2 of the receive. The sender copies $\text{msg}$ in the $\text{vtg}$.value field and sets the flags such that $(0,1) \rightarrow (1,0)$. Now the receiver is the only one of the two partners that can execute the communication primitive. It reads the message and copies it in a private variable $\text{data}$, and sets the flags such that $(1,0) \rightarrow (0,1)$ going back to the initial condition. It is worth noting that line 3 in the send must be executed after ACK is equal to 1 (otherwise the new message can overwrite a previous and possible unreceived message), and line 5 after line 3 (the ready must be set to 1 after the store of the message in $\text{msg}$ is visible to the receiver). Similarly, line 3 in the receive must be executed if and only if ready is equal to 1, and line 5 after line 3 (saving the message in the private variable before it is overwritten by the sender).

When the architecture adopts a weak memory consistency model, we can ensure correctness by forcing the right order of the operations as discussed in Chapter 5. Notably, in message-passing implementation we need to ensure the atomicity of the copies, while in the passing-pointer solution the send algorithm has to avoid reading of stale data.

Zero-copy Receive in Message-Passing Solutions

In order to avoid the copy (unless the message is very short) and to utilize the $\text{vtg}$.value directly in the receiver computation phase (after the receive execution), we provide an alternative zero-copy version of the receive. The semantics must be equivalent to the basic algorithm with copy: thus the ACK cannot be signaled until the process has terminated to utilize (or actually to copy, if convenient) the $\text{vtg}$.value. Otherwise the sender could modify the $\text{vtg}$.value during its utilization. The solution consists in providing a simple additional $\text{set ack}$ primitive, whose effect is to put $\text{ACK} = 1$. The use and the algorithms of the zero-copy version used by the receiver are summarized in Figure 7.3. In the following, we use this version of the receive operation in the case of message-passing implementations.

Communication with any asynchrony degree

Let us now extend the basic rdy-ack solution to communications with more than one buffer position. We denote by $k \geq 1$ the asynchrony degree, which represents the maximum number of messages that a sender can send without waiting for the first sent message being received. In the basic implementation we have $k = 1$. A higher asynchrony degree can be obtained by using $k$ instances of VTG. The sender and the receiver have two private array $\text{CH}$ of $k$ memory pointers to the
target variable instances and a corresponding private index (initialized to zero). The VTG instances are the only data structures shared between the sender and the receiver, and they are used in a round-robin fashion by using index to denote the next VTG to use. Figure 7.4 shows the data structures used in this solution. Each

time the sender wants to transmit a new message, the VTG indexed by the sender CH[index] is selected and the send primitive is executed on it (according to the same pseudo-code of Figure 7.2) and the sender index is incremented by 1 modulo k. Symmetric actions are performed on the receiver’s side using the private CH and index variables.

This algorithm guarantees the correct synchronization. That is, sender (receiver) is blocked on the current VTG if it contains ACK = 0 (RDY = 0), or it completes the primitive if ACK = 1 (RDY = 1).

A first notable advantage with respect to standard lock-based solutions is achieved in terms of caching exploitation: CH are private data structures which represent a clear opportunity of reuse, and the first cache line of VTG contains all the needed synchronization information. Notably, each CH resides permanently in the PrC of the sender and the receiver, with a consequently very low access overhead with re-
7.2. COMMUNICATION RUN-TIME SUPPORT

spect to the case of \( k = 1 \).

Let us study the detailed implementation of rdy-ack send and receive on a single VTG \( (k = 1) \). As introduced in Section 6.2, we have two possible approaches in the implementation of run-time support based on the use of automatic cache coherence:

1. the use of the basic invalidation semantics, which does not guarantee low-\( p \) mappings, or
2. exploiting the home-flush technique, which aims to achieve low-\( p \) mappings at least for structured parallel paradigms.

Solution 1 is feasible in almost any architecture with standard automatic cache coherence. Solution 2 is feasible when the architecture provides mechanisms for home node selection and for flush with de-allocation as introduced in Section 7.1.

Implementation and cost model for automatic cache coherence

Consider again the pseudo-code of send and zero-copy receive for \( k = 1 \). In this case, we can apply the same consideration made in Chapter 5 to derive a cost model for this rdy-ack implementation based on automatic cache coherence. Notably, here VTG encapsulates both data and synchronization information.

Let \( PE_{\text{sender}} \) and \( PE_{\text{receiver}} \) be the respective processing nodes. Though it is likely that home node \( PE_{\text{home}} \) coincides with one of them, no specific strategy nor optimization is assumed in this first run-time support solution.

Consider the send operation. The first read of the ACK value causes the read of a modified value from \( PE_{\text{receiver}} \). If, at the first test, it is \( \text{ACK} = 1 \), then only one cache line transfer occurs (exploiting in-cache retry), otherwise an additional block transfer is paid (due to the invalidation caused by the receiver).

In a message-passing implementation, we estimate the send latency \( T_{\text{send}} \) as the sum of the setup phase \( T_{\text{setup}} \) and the copy phase \( N_{\text{lines}}T_{\text{transm}} \), where \( N_{\text{line}} \) is the number of cache lines used for the msg/vtg value. The setup phase includes the cache line(s) transfer for the ACK value and the write on the first cache line of VTG for modifying RDY and ACK. Therefore, we can estimate the setup cost as

\[
T_{\text{setup}} \sim (1 + p_{\text{wait}})L_{\text{read}}(I, -, M) + L_{\text{write}}(S, S(1), -)
\]

where \( p_{\text{wait}} \) denotes the probability of the waiting condition.

The message copy involves write operations of modified values which, in the worst case, are still in \( PrC_{\text{receiver}} \). We can assume that the message value (msg) is present in \( PrC_{\text{sender}} \), or, in the worst case (i.e., for long messages) in \( ShC \) when present or \( M \), with the additional transfer latency overhead. Therefore, the copy cost can be estimated as

\[
T_{\text{transm}} \sim L_{\text{write}}(I, -, M)
\]
While, in the case of additional cache line transfer we have

\[ T_{\text{transm}} \sim L_{\text{read}}(I, *, -) + L_{\text{write}}(I, -, M) \]

In the case of short messages, notably when the vtg value is in the same cache line of RDY and ACK value, the setup phase already includes the read latency of the copy phase relative to the vtg value. So in this best case scenario we have

\[ T_{\text{transm}} \sim 0 \]

Therefore, we have the following send latency in message-passing implementation in the medium case

\[ T_{\text{send}}^{MP} = T_{\text{setup}} + N_{\text{lines}}T_{\text{transm}} \]
\[ \sim (1 + p_{\text{wait}})L_{\text{read}}(I, -, M) + L_{\text{write}}(S, S(1), -) \]
\[ + N_{\text{lines}}L_{\text{write}}(I, -, M) \]

In a passing-pointer implementation, VTG is represented with few words (e.g., RDY, ACK and memory pointer), therefore we can assume the same send latency of the message-passing implementation in the case of short messages. As discussed in Chapter 5, passing-pointer implementation in WSO systems requires a memory barrier before the reference is copied into the vtg value. For this reason, an additional (possibly partially overlapped) latency is paid during the send operation.

\[ T_{\text{send}}^{PP} = \left[ N_{\text{lines}}L_{\text{write}}(I, M/E, -) \right] + T_{\text{setup}} \]
\[ \sim \left[ N_{\text{lines}}L_{\text{write}}(I, M/E, -) \right] \]
\[ + (1 + p_{\text{wait}})L_{\text{read}}(I, -, M) + L_{\text{write}}(S, S(1), -) \]

Concerning the receive operation, the same consideration to that required for the send and the ACK value, can be applied to the RDY value. Therefore the \( T_{\text{receive}} \) can be estimated as

\[ T_{\text{receive}} \sim T_{\text{setup}} \]

This evaluation is valid both for message-passing and passing-pointer implementations.

When RDY = 1 the msg/vtg value is available for the compute phase, during which it is read when needed, as discussed in Chapter 5. Notably, in a message-passing solution, when set_ack is executed, we can assume that the first VTG cache line is still in PrC_sender adding a negligible cost of \( L_{\text{write}}(M, -, -) \) to the receive latency.

All latencies must be evaluated under-load and for a specific target architecture. As discussed in Chapter 6, in this solution with the basic invalidation semantics, the ratio \( R_Q/R_{Q_0} \) might be substantially greater than one in some parallel programs.
Implementation and cost model for automatic cache coherence: exploiting the \textit{home-flush} technique

As discussed in Section 7.1, we can provide an optimized version of \texttt{send} and \texttt{receive} run-time support by using the \textit{home-flush} technique, in order to use a low-$p$ mapping strategy.

In parallel paradigms, we can have either a communication between a home node sender and non-home node receiver or a communication between a non-home sender and a home node receiver. Notably, we can say that a channel with home node sender has always a non-home node receiver: this is general for any parallel program (e.g. for a worker-worker channel in a stencil-based data parallel program, just one worker is home node for the channel descriptor). On the other side, a channel with non-home sender has a home node receiver: this is not general (e.g. two partners might be both non-home nodes), however it is very likely.

\textbf{Send/Receive for the home node} For the home node, the run-time support of \texttt{send} and \texttt{receive} operations is the same as for of the basic invalidation solution. The corresponding \texttt{receive-set-ack} (\texttt{receive} in pointer-passing implementation), executed by a non-home receiver, reads the first modified cache line of \texttt{VTG} from the home node and updates RDY and ACK through a \texttt{flush}. As described in Section 7.1 the \texttt{flush} operation executed by a non-home receiver node:

1. \textit{de-allocates} the cache line from the receiver cache,

2. copies the block into the $PE_{sender}$ local memory and into $PrC_{sender}$ through a cache-to-cache write request communication.

According to 2, in the home \texttt{send} operation, the read operation on the ACK value are performed locally on $PrC_{sender}$. Moreover, also writing operations are executed on $PrC_{sender}$ and, according to 1, do not invalidate the receiver cache line(s). Therefore, we have

$$T_{\text{setup}} \sim (1 + p_{\text{wait}}) L_{\text{read}}(E, -, -)$$

for the wait condition.

Also the copy phase has a very low latency, writes are executed locally and do not perform invalidation, resulting in

$$T_{\text{trans}} \sim L_{\text{write}}(E, -, -)$$

Therefore, we can evaluate the \texttt{send} operation latency in message-passing solution as

$$T_{h-\text{send}}^{MP} = T_{\text{setup}} + N_{\text{lines}} T_{\text{trans}}$$

$$\sim (1 + p_{\text{wait}}) L_{\text{read}}(E, -, -) + N_{\text{lines}} L_{\text{write}}(E, -, -)$$
While, for pointer-passing solution we have

\[ T_{h\rightarrow send}^{PP} = T_{setup} \]
\[ \sim (1 + p_{wait}) L_{read}(E, -, -) \]  

(7.4)

In message-passing solution, the non-home receiver will read the modified VTG from \( PrC_{sender} \) in the compute phase. Once used, the corresponding cache line(s) must be de-allocated from \( PrC_{receiver} \). No copy is transmitted to the sender and the sender has not to invalidate such blocks. For the pointer-passing solution, the same considerations are applied to the msg value, which is in the worst case still in \( PrC_{sender} \). The setup latency is also valid for the receive operation, with similar consideration for the RDY value, both in message-passing and pointer-passing solutions, resulting in

\[ T_{h\rightarrow receive}^{MP} = T_{h\rightarrow receive}^{PP} = T_{setup} \]
\[ \sim (1 + p_{wait}) L_{read}(E, -, -) \]  

(7.5)

**Send/Receive for the non-home node** When a non-home sender (receiver) reads ACK (RDY) in VTG, if it finds ACK = 0 (RDY = 0) the corresponding cache line must be *de-allocated* and the request is repeated until ACK = 1 (RDY = 1). That is, the wait condition corresponds to read operations on the \( PrC_{receiver} \) (\( PrC_{sender} \)). This avoids unnecessary invalidations by the home node during the synchronization phase, in order to minimize the contention on non-home nodes. However, this feature is paid for repeated read operations on the \( PE_{home} \). This drawback can be partially alleviated by using a *periodic retry* technique.

\[ T_{setup\rightarrow send} \sim (1 + p_{wait}) L_{read}(I, -, M) \]

Concerning the send operation, in message-passing implementation, the message copy is executed by \( N_{lines} \) flush operations. Therefore, we have

\[ T_{trasm} \sim L_{writeC2C}(M, -, -) \]

For short messages, only the first VTG cache line is used, resulting in

\[ T_{trasm} \sim 0 \]

Therefore, in the general case we have

\[ T_{nh\rightarrow send}^{MP} = T_{setup\rightarrow send} + N_{lines} T_{trasm} \]
\[ \sim (1 + p_{wait}) L_{read}(I, -, M) + N_{lines} L_{writeC2C}(M, -, -) \]  

(7.6)

In a passing-pointer implementation, the same operation can be applied to the msg value, which can be *flushed* if modified to the \( PrC_{receiver} \), resulting in a send latency almost equivalent to the message-passing solution.

\[ T_{nh\rightarrow send}^{PP} = T_{setup\rightarrow send} + N_{lines} T_{trasm} \]
\[ \sim (1 + p_{wait}) L_{read}(I, -, M) + N_{lines} L_{writeC2C}(M, -, -) \]  

(7.7)
Regarding the `receive` operation, as said before, in the `receive-set_ack` (`receive` in pointer-passing implementation) operations the receiver, reads the first modified cache line of `VTG` from the home node and updates RDY and ACK through a `flush`. Therefore, we have

\[
T_{\text{nh}-\text{receive}}^{\text{MP}} = T_{\text{h}-\text{receive}}^{\text{PP}} = T_{\text{setup}-\text{receive}} \approx (1 + p_{\text{wait}})L_{\text{read}}(I, -, M) + L_{\text{writeC2C}}(M, -, -)
\]

Considerations about the under-load latencies

A first analysis shows that all solutions have comparable setup latency, except the home version which performs operations on the local `PrC`. In the same way, for the message copy latency in the message-passing solutions, where the home version still shows the minimum latency.

In message-passing solutions, the setup latency is negligible for long messages (impact on `send`) and for relatively coarse-grained calculations (impact on `receive`) and that the message copy latency is negligible for short messages.

As we said, home-flush solution is characterized by low-`p` mapping, but the reduced contention delays are partially paid with flush latencies of non-home nodes for message copy (message flush in pointer-passing solution) and during `receive-set_ack` (receive in pointer-passing solution) and compute phases. Anyway, often, in structured parallel programs, non-home nodes perform send/receive through nondeterministic commands (e.g. farm emitter/collector), thus busy waiting is not applied to a single `VTG`, thus partially compensating the latency penalty.

Moreover, in the basic invalidation solution, the `compute` phase finds the `VTG` value (and possibly the msg value in a pointer-passing solution) in `PrC_{sender}` (unless it is very large). This base latency saving is paid with greater contention, which must be carefully evaluated case by case.

Finally, in the basic invalidation solution, the careful choice of home node (if provided) can lead to further latency reduction (some of which has been evaluated in the home-flush solution): reads and writes executed by home node are not affected by the cache coherent overhead.

### 7.2.3 Rdy-Ack Based on Inter-processor Communications

Let us now introduce an alternative rdy-ack solution, based on interprocessor communications.

In this solution synchronization is greatly simplified and it is more efficient, since it is implemented by interprocessor communications and private data structures only. That is, no shared variables are used for RDY and ACK values. In a message-passing solution shared memory is used for target variable values only: any cache coherence approach can be used, and the home-flush technique reveals quite natural. In a pointer-passing solution also the reference copy can be implemented by
interprocessor communications and any cache coherence approach can be applied to the msg value. For this reason, this Rdy-Ack solution is suitable for non-automatic cache coherence too.

Consider the basic case of a symmetric communication channel with asynchrony degree \( k = 1 \) in an exclusive mapping architecture. In the message-passing solution, only the vtg value is shared (msg value in the passing-pointer solution), without any additional shared information for RDY and ACK which are implemented through interprocessor communications and private data structures. All the RDY-ACK synchronization is done according to a wait-notify scheme. In the message-passing solution, vtg is write-only for the sender, while it is used by the receiver in the compute phase only. The send message copy is performed by explicit flush of each vtg value cache line. As discussed in Chapter [5], in order to guarantee memory ordering in the sequence message copy notify(RDY), all flushes are synchronous, or a Memory Barrier is inserted before notify(RDY).

In the pointer-passing solution, the notify ready actually corresponds to both the ready event and the send of the msg reference to the receiver. Also in this case, explicit flush of each msg value cache line can be performed before the notify(RDY) ensuring the right memory ordering when necessary.

**Handling multiple communications**

Let us now extend this initial case to the possibility of having a generic number of communication channels per process/thread and any asynchrony degree for each channel.

Consider a parallel program and the set of the communication channels used by each module. We can associate a unique identifier to each channel \( CH_{1}, ... CH_{n_{ch}} \). For each process, a private Channel Table \( TAB_{CH} \), indexed by CH identifiers, is provided. Notably, \( TAB_{CH}[CH_{i}] \) is the pointer to the corresponding channel structure, which is represented in Figure 7.5 for the message-passing solution.

For each channel \( CH_{i} \), in addition to the \( k_{i} \) shared VTG instances and to the private pointer structures \( CH \) for the sender and the receiver, two private data structures \( EVENT_{ACK} \) and \( EVENT_{RDY} \) are provided for sender and receiver respectively. Each of these event structures has \( k_{i} \) entries with binary values = \{0, 1\}. Each entry is the (RDY,ACK) pair of the corresponding VTG instances used in the wait-notify scheme. As in the shared-memory Rdy-Ack solution, VTG instances are used in a round-robin fashion, and for each instance the same wait-notify synchronization technique of the basic case is applied. The complete definition of send and receive-set_ack operations for the Rdy-Ack communication based on interprocessor communications is shown in Figure 7.6, where index is the current index value of CH. The definition and implementation of wait and notify operations is the following and summarized in the pseudo-code in Figure 7.7. Each interprocessor communication consist
Figure 7.5: Rdy-Ack channel structure for the message-passing solution based on interprocessor communications

Figure 7.6: Pseudo-code of send, receive and set_ack operations for Rdy-Ack channel structure for the message-passing solution based on interprocessor communications
in the communication with the \textit{notify} operation of the pair \((CH_i, index)\) in order to perform the corresponding synchronization on the \((RDY_{index}, ACK_{index})\) of \text{TAB} \text{CH}[CH_i].

The reception of this message is treated as an interrupt: a run-time support interrupt \text{-handler} is called each time a new message arrive. The message is inspected to determine the corresponding communication channel and the corresponding event is set (i.e., \text{TAB} \text{CH}[CH_i] \rightarrow EVENT[\text{index}] = 1).

The \textit{wait} operation checks the corresponding event and if it is not RDY or \(ACK = 1\), waits for new interprocessor communications also buffering other possible event received for other channels used by the same process. For pointer-passing solution, we can use similar data structures and algorithms as summarized in Figures 7.8. In this case, no VTG are \textit{shared}; when a new interprocessor communication is received by the receiver the message pointer communicated is copied in a local private buffer associated to the CH of the receiver in the corresponding \text{TAB} \text{CH}[CH_i].

---

Figure 7.7: Pseudo-code of \texttt{wait}, \texttt{notify} and \texttt{interrupt-handler} operations for Rdy-Ack channel structure for the message-passing solution based on interprocessor communications

```
wait(ra_io, event, index) {
if(ra_io->event[index] == 1)
  ra_io->event[index] = 0;
else {
  while(receive_IP_comm(ch_id,ev,id))
    if !(event==ev & ra_io->ch_id==ch_id & index==id)
      tab_ch[ch_id]->ev[id] = 1;
}
}
```

```
notify(ch_id, event, index) {
  send_IP_comm(tab_ch[ch_id]->PE,ch_id,event,index);
}
```

```
interrupt(ch_id, event, index) {
  tab_ch[ch_id]->ev[id] = 1;
}
```

---

Figure 7.8: Rdy-Ack channel structure for the pointer-passing solution based on interprocessor communications
Cost model for rdy-ack based on interprocessor communications

In the message-passing solution the setup phase corresponds to the execution of the wait and notify operations. All private data structures are reused in the local PrC, possibly in the higher level.

The most likely situation in the wait execution is that the event (RDY/ACK) has already been set in CH; otherwise, if rcv_com is executed, it is likely that the received message is consistent with the waited condition. The condition relative to the reception of another event is rarely verified, and the probability drops rapidly with the number of the loop iterations.

The notify operation has a negligible impact too, because of the asynchronous latencies of the interprocessor communication, which is overlapped. Concerning VTG, it is used in write-only mode in the send operation, so we can force a non-allocation policy in order to avoid the read operation latency. VTG is not accessed at all in the receive operation, it is used only in the compute phase.

Finally, in the setup phase we need to consider the contribution of the interrupt-handler, which, as discussed in Section 7.1, in architecture with an efficient interprocessor communication system, can be comparable to the PrC access latency. Therefore we have for both send and receive-set_ack operations

\[ T_{\text{setup}} \sim L_{\text{read}}(M/E/S, -, -) \]

In old-style machines which implement interprocessor communications in kernel-mode, the interrupt-handler has a substantial cost, resulting in a larger \( T_{\text{setup}} \). In this case, this rdy-ack approach may prove not convenient.

The message copy phase, with a non-allocation policy and according to the automatic or non-automatic implementation is paid during the send operation in the case of the home-flush performed by the sender, or during the compute phase by reading from the GC if no cache-to-cache write requests are used. Again, we can assume that the message is present in PrC_sender. Therefore, we have for solutions which adopt the home-flush technique the following \( T_{\text{trasm}} \) latency

\[ T_{\text{trasm}} \sim L_{\text{writeC}2C}(M, -, -) \]

While, for non-automatic cache coherence systems in which cache-to-cache write requests are not supported, we have

\[ T_{\text{trasm}} \sim L_{\text{write}}(I, M, -) \]

which is paid during the compute phase.

Analogous considerations can be made for the pointer-passing solution. Notably, the cache-to-cache write requests can be exploited when possible for the msg value. Therefore, in the general case we have

\[ T_{\text{IPcomm}} = T_{\text{setup}} + N_{\text{lines}}T_{\text{trasm}} \]

\[ \sim L_{\text{read}}(M/E/S, -, -) + L_{\text{writeC}2C}(M, -, -) \]
In general, for each specific architecture and parallel program, we are able to estimate $T_{\text{setup}}$ and $T_{\text{trans}}$ accurately. This rdy-ack solution shows a very simple and efficient way of implementing synchronization mechanism and exemplifies the potential simplifications and optimizations achievable with the non-automatic, or flush-based automatic, cache coherence solutions. Potentially, this run-time support is characterized by a base communication latency which is comparable to, or even lower than, the shared memory rdy-ack version, provided that the interprocessor communication system is implemented in an efficient way (like in modern CMP-based architectures). For this reason, this approach is particularly suitable for fine-grained computations too.

**Considerations about the under-load latencies**

The absence of using shared-variables for RDY-ACK values during the send and receive operations, and the application of optimizations (e.g., flush strategies) for the access to vrg and msg values, avoid some critical situations of contention. Therefore, this solution is able to achieve low-p mappings using the general principles of Chapter 6 and the optimizations of Section 7.1. On the other hand, we need to study the impact of RDY and ACK notifications. All PEs exchange interprocessor messages in order to perform synchronizations. For example, in a farm the Emitter and Collector receive a number of notifications which is equal to the number $nw$ of workers (analogously for data parallel INPUT and OUTPUT modules). We cannot model this situation using the classical client-server model with request-reply behaviour. In fact, RDY/ACK notifications are asynchronous. In this case the system is modeled as an acyclic graph (Section 6), so some nodes might potentially become bottlenecks.

The critical parameter is now the service time of nodes for serving asynchronous notification requests, which is determined by the interrupt-handler operation service time (or the equivalent computation in the wait operation). As we already said, this service time is very low in modern CMP-based systems. In this way, the notification interarrival time to the most stressed nodes, which is equal to the stream interarrival time (for the Emitter/Input), or to the ideal service time of the whole parallel paradigm, is actually greater than the notification service time. This results in a utilization factor of the corresponding module $\rho < 1$.

Otherwise, in no way the parallel program would be able to achieve the ideal bandwidth, independently of the notification implementation and evaluation issue. A final condition is that also the interprocessor communication implementation provides a sufficient asynchrony degree/buffering capability in order to not add an overhead in the queuing delay of the corresponding modules.

In conclusion, asynchronous notifications do not affect system performance on condition that the architecture provides a low overhead interprocessor communication mechanisms with sufficiently large asynchrony degree.
7.3 Asymmetric Rdy-Ack Communications

Non-determinism in asymmetric communications is made easier to design, and efficient, in the Rdy-Ack model with exclusive process mapping and busy waiting. Support for many-to-one communications can be simply achieved with the receiver process/thread testing the selected channels RDYs in a round-robin fashion until RDY = 1 is met. Let $CH_1, ..., CH_m$ be the channel set used for the asymmetric communication. If $CH_i$ is the most recently used channel, the round-robin scan starts from $CH_{(i+1) \mod (m)}$.

With the exclusive process/thread mapping, we avoid any global synchronization of senders, as confirmation of the real lock-free nature of this rdy-ack model. Instead, with multiprogrammed mapping lock-based global synchronization or alternative approaches based on CAS instructions are necessary \cite{79, 104} for a correct execution of process low-level scheduling. Therefore, the latency overhead is relatively small, and it is further reduced if proper data structures are used to allow testing of several RDYs simultaneously.

Support for one-to-many communications can also be efficiently supported and the implementation is straightforward: ACKs are tested (simultaneously and) in a round-robin fashion. For example, an on-demand farm Emitter tests the task channels’ ACKs in a round-robin fashion until ACK = 1 is met. The selected channel is a good candidate for a load-balanced distribution of the next task. The non-determinism implemented with this solution avoids the available communications from workers to Emitter. In this way we reduce the communication traffic, and, notably, we are able to reduce contention also in automatic cache coherence approaches without the home-flush optimization.

One-to-many and many-to-one communications are used for data distribution and data collection, respectively, in farm and data parallel paradigms. In both cases, the flush-based communication run-time support is even better exploited. As introduced in Section\cite{72}, several channels are scanned, thus no retry is performed on the same channel by non-home node sender (receiver), respectively Emitter and Input process (Collector and Output process) in farm and data parallel paradigms. Therefore, we can assume the costs of send and receive operations also for asymmetric communications with an additional overhead due to the overhead of testing multiple RDY/ACK values.
7.4 Implementation and Evaluation on Tilera TilePro64

In this Section we present our experience in the implementation of the rdy-ack communications as inter-thread interaction mechanisms on the Tilera TilePro64. Although it is a domain-specific parallel architecture, this architecture represents a notable example of how advanced architectural structures, such as user-accessible on-chip interconnection networks and configurable cache coherence protocols, are of great importance to design lightweight cooperation mechanisms enabling efficient parallel implementations.

As introduced in Chapter 1, the Tilera TilePro64 is equipped with 64 identical PEs (called tiles) interconnected by an on-chip network named iMesh. Each link consists of two 32-bit-wide unidirectional physical links carrying the traffic in both directions. The iMesh network is composed of five independent 2D meshes each one carrying a different kind of traffic. Notably, the User Dynamic Network (UDN) supports the explicit transfer of small messages (up to 128 32-bit words) among tiles under application programmer control. Each tile has five UDN hardware queues connected directly to the processor registers. Special assembler instructions are provided to perform the enqueue/dequeue and the transmission over UDN. The UDN serves user-land processes or threads, providing a flexible and low latency cooperation mechanism.

The Tilera TilePro64 provides also a flexible cache subsystem named Dynamic Distributed Cache (DDC) which implements the automatic cache coherence protocols according to the abstract model [AM2b] described in Chapter 4.

Figure 7.9 summarizes the actions performed by the automatic cache coherence in read and write operations. This architecture allows to finely control the cache coherence mechanism offering the following features:

1. flexible home node selection
2. write-through C2C between the requestor node and the home node
3. disabling of the automatic cache coherence with explicit flush and de-allocation mechanisms

All these characteristics, make Tilera TilePro64 an interesting candidate for the comparison of the performance of structured parallel applications with the different run-time supports presented in this Chapter.

First Results on Tilera TilePro64

Before analyzing the implementation of the rdy-ack solutions studied in the previous Sections, we report the first experiences of our research group in the implementation
Figure 7.9: Tilera TilePro64 automatic cache coherence protocols
of run-time supports for parallel patterns on Tilera TilePro64 architecture.
In [24], we discussed the porting of the FastFlow [15] framework on this architecture. Notably, Fastflow is a passing-pointer solution which provides programmers with predefined and customizable stream parallel paradigms such as task farms and pipelines. In our porting, we deal both with the implementation of stand-alone applications and applications executed by using the Tilera TilePro64 as a software-accelerator. We obtained very interesting results, related to the cache coherence problems, by encapsulating at the skeleton level three alternative cache coherence allocation strategies for the task exchanged between the modules of a farm skeleton:

- **Hash Home Node** (HHN), which corresponds to the default cache coherence protocol defined by the architecture: a hash function is used to uniformly distribute home nodes among all the caches. HHN guarantees a uniform usage of all the caches, although it may increase the network traffic and reduce the effective amount of cache usable per tile with high parallelism degrees.

- **No Home Node** (NHN), which disables the automatic cache coherence, resulting in incoherent memory pages. Coherency is ensured only when a task is passed to a different concurrent entity. When the work on the local task is finished and before sending the task to another concurrent entity, the FastFlow run-time automatically and transparently adds memory flush operations to enforce cache coherence.

- **Fixed Home Node** (FHN), which specifically selects, for each task, a PE that becomes its home node. This strategy aims to remove most of the performance overhead of the DDC mechanism. This characterization is actually possible for the farm paradigm, where each task is entirely processed by a single thread. Although theoretically very promisingly, the main problem relative to this policy arises considering that the destination thread for a specific task is usually defined late at runtime. This means that, in a pointer passing environment such as FastFlow, it is usually necessary to copy the task on a new memory area after the worker is elected, to select the proper home node (thus voiding all the effect of pointer passing).

We executed a matrix multiplication $A[N][N] \times B[N][N]$ written in FastFlow exploiting the farm skeleton on a stream of 3200 matrices. Figure 7.10 shows the results for two test cases: one using matrices of integers with $N=64$ and the other with $N=128$. For each one we tested the three cache coherence strategies supported in the farm paradigm. With high parallelism degrees we can actually see very different results depending on the strategy used.

In the case of 64x64 matrices, each matrix takes 16KB of space, so that the entire working set of each worker for each task is 48KB (two input matrices plus an output one). This means that the working set is small enough to fit in the L2 cache of one tile and therefore the number of memory transfers are minimized. We are expecting an
7.4. IMPLEMENTATION AND EVALUATION ON TILEPRO64

![Graph](image)

(a) 64x64 integer matrices  
(b) 128x128 integer matrices

Figure 7.10: FastFlow stream matrix multiplication (AixBi) using different cache coherence strategies: Hash Home Node (HHN), No Home Node (NHN) and Fixed Home Node (FHN)

The graphs above illustrate the performance of FastFlow stream matrix multiplication using different cache coherence strategies: Hash Home Node (HHN), No Home Node (NHN) and Fixed Home Node (FHN) for both 64x64 and 128x128 integer matrices.

extremely high scalability, that is in fact verified with the FHN and NHN strategies. In contrast, the standard cache coherence protocol works very well up to \( \sim 20 \) nodes, then suddenly stops scaling. This is because with a large parallelism degree, the L2 home must contain and manage an update copy of cache lines for which is responsible for. Thus, the cache available for each tile is less than the required. In this case, the working set of the algorithm does not fit in the cache and the performance of the sequential code executed by the workers suddenly decrease. On the other hand the NHN implementation is indeed very good, as it is able to obtain aligned results with the best option for automatic cache coherence.

By using larger matrices we expect the working set to not fit the caches in any of the policies. Still, it represent an interesting experiment as we are stressing the memory, and thus we can actually see if the coherency protocol helps or aggravates the situation.

In this test is possible to observe the benefits of using the automatic cache coherence when using the HHN policy: when running sequential programs or parallel ones with small parallelism degree, we may have that the sum of all L2 caches is large enough to contain the working set of the application, so that the performance can be much better than the other two strategies. This also means that the speedup (calculated with respect to a standard sequential version which uses HHN strategy) is indeed an unfavorable metric for the other two modes.

The incoherent policy works surprisingly well: by removing the cache coherency protocol we reduce the amount of memory requests, or at least the amount of traffic on the network, ending with far better results with respect to any implementation that exploits automatic cache coherence.
Rdy-Ack implementations

The results obtained with FastFlow were very promising. The communication mechanisms between the modules of the parallel paradigms is lock-free and wait-free, where synchronization data and the exchanged value are strictly coupled. This restriction poses a limitation in space on the possible optimizations offered by current CMP-based architectures. On the other hand, with a rdy-ack communication model we are able to provide:

- specific and possibly various home selection strategies for each data structure used by the run-time support
- specific home selection strategies for each parallel paradigm
- different synchronization mechanisms according to the grain of the parallel computation

Moreover, with the rdy-ack solutions we provide alternative message-passing and pointer-passing solutions, in order to choose the better implementation model according to the performance parameters of the specific application.

Rdy-Ack based on Shared Memory Synchronizations

The API provided by the Tilera Multicore Library (TMC) \cite{35} offers an explicit home node selection with which the user can choose the PE which will be the home node for a specific cache line or data structure. As discussed, the cache coherence protocols in this architecture define the write operations in terms of write through communications between a generic PrC and the \( PrC_{\text{home}} \) for the corresponding cache line. We can exploit these mechanisms to emulate the home-flush techniques in the implementation of the rdy-ack run-time support. Notably, consider the activities performed by the two partners on a rdy-ack communication channel on the different fields of VTG, which are shown in Table 7.1.

<table>
<thead>
<tr>
<th></th>
<th>RDY</th>
<th>ACK</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>WRITE-ONLY</td>
<td>READ-WRITE</td>
<td>WRITE-ONLY</td>
</tr>
<tr>
<td>Receiver</td>
<td>READ-WRITE</td>
<td>WRITE-ONLY</td>
<td>READ-ONLY</td>
</tr>
</tbody>
</table>

Table 7.1: Reading and writing activities on the VTG fields

Our solution consists in partitioning the fields of the VTG in different cache lines:

- a cache line contains only the ACK flag and its home node is \( PE_{\text{sender}} \)
- one or more cache lines (depending on the message-passing vs passing-pointer solution) contains the RDY and the vtg value, which are homed on \( PE_{\text{receiver}} \)
7.4. IMPLEMENTATION AND EVALUATION ON TILEPRO64

Figure 7.11: Rdy-Ack based on Shared Memory synchronizations with home-flush optimization on Tilera TilePro64

Figure 7.11 summarizes this solution. During the send operation, the RDY flag and the vtg value are transmitted directly from $PE_{sender}$ to $PE_{receiver}$, which is the home node of that cache line(s) owning the updated copy. The opposite behavior occurs during the execution of the receive, when the new value of the ACK flag is transmitted directly from $PE_{receiver}$ to $PE_{sender}$.

In order to avoid invalidation messages, as happens in the home-flush techniques, since the de-allocation is not possible without the side effect of the de-allocation also from the home node $PrC$ with the no allocation policy. Notably, the write operations on the write-only fields of VTG can be performed with the no allocation policy available on this architecture. In this way, the sender and the receiver do not need to transfer their write-only part of VTG into their $PrC$.

The run-time support for asymmetric communications (i.e., many-to-one and one-to-many) is straightforward, according to the definition of the asymmetric rdy-ack communication described in Section 7.3. Figure 7.12 summarizes the data structures used in both types of asymmetric communications.

Finally, send and receive operations are defined in order to ensure the correctness of the protocol, by means of memory fence instructions, since Tilera TilePro64 adopts a weak memory consistency model.

Rdy-Ack based on Interprocessor Communications In addition to the facilities exploited in the shared memory case, this implementation relies on the UDN on-chip network for the interprocessor communications. Every tile can transmit a message composed of one header word and the payload by specifying the destination tile and a tag associated with the message which is used to forward the message to the corresponding UDN hardware queue of the five available. As described in Section 7.2.3, we transmit a payload with 2 or 3 words depending on the message-passing/pointer-passing solution.
Figure 7.12: Asymmetric Rdy-Ack based on Shared Memory synchronizations with home-flush optimization on Tilera TilePro64

Figure 7.13: Rdy-Ack based on interprocessor communications on Tilera TilePro64

Figure 7.13 summarizes the data structures used in both message-passing and passing-pointer solution. The run-time support for asymmetric communications (i.e., many-to-one and one-to-many) is straightforward, according to the definition of the asymmetric rdy-ack communication described in Section 7.3, with similar data structures to the previous shared memory case.

As in the previous case, send and receive operations are defined in order to ensure the correctness of the protocol, by means of memory fence instructions, since Tilera TilePro64 adopts a weak memory consistency model.

Finally, since the buffering space of a UDN queue is of 128 32-bit words, in order to give a proper asynchrony degree for an entire parallel program, we exploit all the UDN queues for the interprocessor communications.
7.4. IMPLEMENTATION AND EVALUATION ON TILEPRO64

First results  A first approach to the implementation of the rdy-ack communication model on Tilera TilePro64 has been presented by our research group in [25]. As stated, it was a first step toward the evaluation of the communication model presented in this Chapter. Notably, this work studies the implementation of rdy-ack communications based on shared memory and interprocessor communications only for passing-pointer solutions and for one-to-one and many-to-one communications. We briefly report some important results that can be useful to the evaluation of the rdy-ack implementations provided in the following part of this Section.

A set of micro-benchmarks are used to study the communication latency of different implementations using the shared memory and the UDN supports. The latency benchmarks are carried out using a ping-pong scheme. A sender transmits a one-word message to a receiver executed on a different tile and waits for a reply from the receiver. The receiver receives the message, and sends back a reply to the sender. The benchmark consists of many iterations \( I \). The execution time of a pair of send/receive operation (named \( T_{exchange} \)) is measured as the completion time of the benchmark \( T_C \) divided by the number of iterations (i.e., \( T_{exchange} = T_C/I \)). The communication latency to execute a single communication operation can be estimated by \( L_{com} \sim T_{exchange}/2 \). Figure 7.14 summarizes the results for symmetric communications, comparing the rdy-ack based of shared memory synchronizations using automatic cache coherence (ch_sym_sm) and the emulation of the home-flush technique (ch_sym_cache) with respect to the solution based on the UDN interprocessor communications (ch_sym_udn).

These experiments confirm the evaluation made with the cost model in the previous Section. The home-flush techniques, even if emulated on this architecture, improve the latency of the shared memory support. Notably, for write-only cache lines it is more convenient to use the non-home run-time support for the sender, while, for read-write data used by a single thread the home run-time support is the best solution to reduce the automatic cache coherence overhead. This optimization leads to a 50-65% improvement with respect to the standard automatic cache coherence solution. More important are the results obtained using the UDN on-
chip network. The corresponding solution considerably outperforms the best shared memory variant. The result is that the communication latency is one third of the one of the best shared memory implementation. For each solution, the communication latency is influenced by the distance (1/8/14 hops in the figure) between $PE_{sender}$ and $PE_{receiver}$. The longer the distance the greater the latency, except for the standard automatic cache coherence solution in which distance between $PE_{sender}$ and $PE_{receiver}$ is alleviated by the allocation of the $PE_{home}$.

To evaluate the many-to-one communication mechanism, the same ping-pong benchmark is used to evaluate the additional overhead of the asymmetric mechanism in communications between two threads. Another benchmark is used to study the overhead of the non-deterministic selection by changing the number of the senders. Notably, one sender sends messages to the receiver following the ping-pong scheme, while the other senders are idle. Figure 7.15a summarizes the results for asymmetric communications, where (ch_sym_sm) represents the results for rdy-ack shared memory solution with the emulation of the home-flush technique.

In the average case the gain of using the UDN support compared with the shared memory version is more than 50%. As discussed in [23], the UDN support shows that the many-to-one mechanism features a penalty of 20% in communication latency with respect to the one-to-one case. The reason is that in the asymmetric communications (as discussed previously) the UDN message is composed of two 32-bit words (ch_id, msg_ref), while in the symmetric case the first word is not necessary because the communication is between a static pair of PEs.

Figure 7.15b shows the important result that the latency offered by the many-to-one communication does not depend on the number of senders with the UDN support. In contrast, the shared memory solution has a cost proportional to the number of senders.

Figure 7.15: Rdy-Ack many-to-one communication latency for passing-pointer solution evaluated with the ping-pong micro-benchmarks.
Implementation of parallel applications  Consider the 3-stage pipeline computation represented in Figure 7.16, in which:

- the first stage (S1) represents a module which encapsulates M matrices of integer values A[N][N] and generates a stream of these matrices. Let A[i] be the generic stream element.
- the second stage (S2) is defined with the pseudo-code in Listing 7.1
- the third stage (S3) receives the results from the previous stage and stores all the received matrices of integer values B[N][N] into other M matrices.

Although this computation is very generic, an increasing number of emerging applications use this pattern for different purposes, such as network traffic and sensor data processing, e-business transactions monitoring and real-time analysis of data streams form social media.

Listing 7.1: Pseudo-code of Stage 2 in a 3-stage pipeline computation

```c
int A[N][N], B[N][N];
ra_comm input_stream, output_stream;
while (true) {
    receive(input_stream, A);
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            B[i][j] = F(A[i][j], ...);
    send(output_stream, B);
}
```

Depending on the average service time of function $F(T_F)$ and the architecture/runtime supports parameters (i.e., $T_{send}$ and $T_{receive}$), the second stage S2 can represent the bottleneck of the 3-stage pipeline. As studied in the previous chapters, in order
to eliminate the S2 bottleneck, we can evaluate the optimal parallelism degree of S2 as follow:

\[ n_{opt} = \left\lfloor \frac{T_{S2-id}}{T_{A-S2}} \right\rfloor \]

where

\[ T_{S2-id} = T_{receive} + T_{calc} + T_{send} \simeq T_{receive} + N^2 \cdot T_F + T_{send} \]

We study the performance of the computation and its parallelization with the farm paradigm. In order to compare the different run-time support solutions, we studied different versions of this computation, changing the following parameters:

- \( N = 32, 64, 128 \) in order to study the computation with different \( T_{calc} \);
- with the same \( N \) values, we study the effect of coarser grain computation; notably, Tilera TilePro64 does not have floating-point units and decimal operations are emulated by software, thus, by using single precision decimal numbers as data type, we study the effect of coarser grain computation with the same number of cache misses;

We report in table [7.2] the most important latencies studied for the corresponding abstract model [AM2b] and the evaluations obtained by the benchmarks presented in Chapter 4. We use the write latencies without overlapping in order to consider the impact of the memory fence instructions used in the run-time support to solve the memory ordering problem in this WSO architecture.

Using this base latency we are able to evaluate \( T_{send} \) and \( T_{receive} \) for the various run-time support solutions, considering \( \sigma = 16 \), as studied in Section 7.2. Notably, we have

- for rdy-ack based on shared memory synchronization with automatic cache coherence (ra_sm), varying the \( PE_{home} \) allocation, in the best case scenario where \( p_{wait} = 0 \)

\[
\begin{align*}
T_{MP_{send}} & \simeq 172 - 224\tau + N^2 \cdot (3 - 5\tau) \\
T_{PP_{send}} & \simeq \left[N^2 \cdot (2 - 4\tau)\right] + 172 - 224\tau \\
T_{MP_{receive}} & \simeq T_{PP_{receive}} = 172 - 224\tau \\
T_{MP_{nh-send}} & \simeq T_{PP_{nh-send}} = 160 - 204\tau + N^2 \cdot (0 - 1)\tau
\end{align*}
\]

- for rdy-ack based on shared memory synchronization with the emulation of the home-flush technique (ra_sm_home)

\[
\begin{align*}
T_{MP_{h-send}} & \simeq T_{PP_{h-send}} = 2 - 8\tau + N^2 \cdot (0 - 1)\tau \\
T_{MP_{h-receive}} & \simeq T_{PP_{h-receive}} = 2 - 8\tau \\
T_{MP_{nh-send}} & \simeq T_{PP_{nh-send}} = 160 - 204\tau + N^2 \cdot (0 - 1)\tau
\end{align*}
\]
### 7.4. IMPLEMENTATION AND EVALUATION ON TILEPRO64

<table>
<thead>
<tr>
<th>Read/Write and CC state</th>
<th>Cache Block Read Base Latencies</th>
<th>Benchmarks Results (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_{\text{read}}(M/E/S, -, -) )</td>
<td>( T_{\text{PrC}} )</td>
<td>2-8</td>
</tr>
<tr>
<td>( L_{\text{read}}(I, M/E/S, -) )</td>
<td>( T_{\text{LC}} + L_{\text{net}} + T_{\text{GC}} + L_{\text{net}}(\sigma) + T_{\text{PrC}} )</td>
<td>40-70</td>
</tr>
<tr>
<td>( L_{\text{read}}(I, -, M) )</td>
<td>( T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + L_{\text{net}} + T_{M} + L_{\text{net}}(\sigma) + T_{\text{PrC}} )</td>
<td>160-204 (non-home) 120 (home)</td>
</tr>
<tr>
<td>( L_{\text{write}}(E, -, -) )</td>
<td>( L_{\text{net}} + T_{\text{GC}} + L_{\text{net}} + T_{\text{PrC}} )</td>
<td>5-28 (non-home) 2-7 (home)</td>
</tr>
<tr>
<td>( L_{\text{write}}(I, M/E, -) )</td>
<td>( T_{\text{LC}} + L_{\text{net}} + T_{\text{GC}} + L_{\text{net}}(\sigma) + T_{\text{PrC}} )</td>
<td>45-73 (non-home) 2-7 (home)</td>
</tr>
<tr>
<td>( L_{\text{write}}(I, S(n_{sh}), -) )</td>
<td>( T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + L_{\text{net}}(\sigma) + [L_{\text{inv}}(n_{sh})] + T_{\text{PrC}} )</td>
<td>52-332 (non-home)</td>
</tr>
<tr>
<td>( L_{\text{write}}(S, S(n_{sh}), -) )</td>
<td>( T_{\text{LC}} + L_{\text{net}} + T_{\text{lookup-GC}} + L_{\text{inv}}(n_{sh}) + [L_{\text{net}}] + T_{\text{PrC}} )</td>
<td>12-292 (non-home) 7-285 (home)</td>
</tr>
<tr>
<td>( L_{\text{write-C2C}}(M, -, -) )</td>
<td>( L_{\text{net}}(\sigma) + T_{\text{PrC}} + L_{\text{net}} )</td>
<td>30-60</td>
</tr>
</tbody>
</table>
A STRUCTURED PARALLELISM APPROACH TO CC

\[ T_{nh-receive}^{MP} \simeq T_{nh-receive}^{PP} = 160 - 204\tau + N^2 \times (0 - 1)\tau \]

- for rdy-ack based on interprocessor communications (ra_ip), we used the results in [25] to have a good estimation of the \( L_{IP} \) latency

\[ T_{send}^{IP} \simeq 50 - 129\tau \]
\[ T_{receive}^{IP} \simeq 20 - 69\tau \]

Of course, we need to consider the corresponding under-load latencies, in the various cases. Notably, for the ra_sm solution the architecture is modeled as a NUMA, independently of the global organization of nodes and external memories. The NUMA-equivalent characterization derives from the client-server interactions of PEs for performing the cache coherence protocol actions. This solution is not characterized by the \( low-p \) mapping strategies, \( p \) and, consequently, \( R_Q/R_{Q_0} \) grows with the parallelism degree (\( nw \)) of the farm parallelization (Chapter [9]). Notably, \( pE = nw + 1 \), while for the initial pipeline computation is at most \( p = 4 \) if \( PE_{home} \) does not coincide with \( PE_{S1}, PE_{S2}, PE_{S3} \).

We initially compare the results obtained with integer matrices varying the \( N \) dimension. Figure 7.17 shows the results for the message-passing and pointer-passing solutions with the various run-time support respectively for \( N = 32/64/128 \). The results are very interesting.

In general, the finer the computation the more the gain of using the UDN support for interprocessor communications. Notably, for the case \( N = 32 \) with the message-passing implementation we achieve an improvement of about 50% with ra_sm_io with respect to the ra_sm_home solution.

We can also notice the particular behavior of the ra_sm solution, especially in cases \( N = 64/128 \), in both the implementation solutions. This behavior is due to the problem discussed previously: with a large parallelism degree the default hashing home strategy used by the automatic cache coherence solution causes the decrease of the availability of the L2 cache for each node. Of course, the bigger are the matrices the smaller is the parallelism degree at which this phenomenon starts. In the case \( N = 64 \), the problem is a bit alleviated in the message-passing solution, probably because the working set of each node is composed of the msg value and the vtg value, requiring more accesses to the main memory which in some cases can be nearer with respect to the home node selected by the default mechanism.

With the ra_sm_home this problem is clearly solved by selecting the proper home node. In this case, we can see that the passing-pointer solution does not offer appreciable improvements with respect to the message-passing solution due to the copy required to select at run-time the home node, as discussed before.

Another important consideration is in the differences between the message-passing and the passing-pointer solutions in the ra_io case. In the passing-pointer
implementation the messages exchanged through the UDN network require an additional word for the message reference with respect to the message-passing solution. This has the side effect of reducing the effective asynchrony degree of the communications with high parallelism degree. Although, all the UDN queues are exploited we have to use an asynchrony degree \( k = 1 \) due to the number of communications channels used by the parallel application. This problem was anticipated and its effect discussed in Section 7.2.3.

As we can expect the speedup increases with bigger problem sizes, when the communication overhead, that influences the emitter’s \( T_A \), is a smaller portion of the overall execution time. However, it is still far from the ideal one. The reason is that the parallel efficiency is limited by the available memory bandwidth. In fact, though the exploitation of the four on-chip memory controllers (MINFs) and the corresponding memory macro-module to store the matrices, the memory bandwidth is not sufficient to sustain a high number of working nodes. To demonstrate this fact we show in Figure 7.18 the results obtained with coarser grain computations by using single precision decimal numbers as data type. In this case, MINFs are subjected to a lower pressure from the nodes and the parallelization achieves better speedup with \( N = 32 \) and near optimal speedup with \( N = 64/128 \). The advantage of using the UDN for interprocessor communications is remarkable with small matrices, while it decreases with larger problem sizes up to achieving the same performance of the ra_sm_home solution. These results confirm our ideas about the rdy-ack communications based on interprocessor communications, about how exploiting such architectural feature we achieve scalable parallelizations of fine-grained problems. Moreover, the home-flush techniques, even if emulated in this architecture, offers significant advantages and a more predictable behavior with respect to the default automatic cache coherence solution.

7.5 Summary

All the considerations made in the second part of the thesis, have been used in this chapter to provide an optimized run-time support for structured parallel applications for advanced CMP-based multiprocessors. With this chapter we contribute in the design and study of parallel paradigms’ run-time support with optimization strictly related to the cache coherence problem and its impact in the parallel performance applications. Notably, a complete set of run-time supports have been provided and evaluated, such as message-passing vs passing-pointer solutions, as well as solutions based on automatic cache coherence or based on specific optimizations guided by the cost models derived from the previous chapters. In the final part of this section we discuss the implementation of the optimization discussed in this chapter on the
Tilera TilePro64 processor. This architecture represents a good candidate for the evaluations of our solutions, due to the possibility of implementing or emulating our ideas. Notably, with this architecture we are able to achieve an improvement of about 50% with respect to the use of the default cache coherence solution.
Figure 7.17: Speedup of the farm computation with the various run-time supports: integer values matrices with N=32/64/128 Tilera TilePro64
Figure 7.18: Speedup of the farm computation with the various run-time supports: float matrices with N=32/64/128 Tilera TilePro64
With this thesis we studied performance models and optimizations for CMP-based architectures, with particular attention to the cache coherence problem. In particular, we focused on the performance prediction of parallel patterns, in order to evaluate the impact of off-the-shelf automatic cache coherence solutions. To achieve this result, we developed an abstract model for cache coherent CMPs which is a simplified view of a concrete target architecture able to describe the essential performance properties and abstract from all the others that are useless. It provides the base access latency for memory and cache operations in terms of the automatic cache coherence solution adopted. The model provides a first result toward the evaluation of the impact of automatic cache coherence on parallel program performances, by analytically defining the base memory and cache access latencies of reading and writing operations in terms of the coherency protocol adopted. Starting from this model and by using performance evaluation solutions, such queuing networks and process algebra (i.e., PEPA) we are able to estimate the average response time of the various level of the memory hierarchy according to the parallel applications defined through well-known parallelism paradigms. This cost model is fundamental in the definition of the parallel paradigms run-time support, showing for example how a specific mapping strategy can improve performances by minimizing the under load latencies. Moreover, the results obtained with the resulting cost model, allow us to compare the impact of different cache coherence solutions, e.g., automatic vs algorithm-dependent solutions. Notably, the latter are designed in order to provide an optimized run-time support for structured parallel applications. This optimizations represent the results of the considerations made through combined analysis of the cost model of the specific architecture and the parallel application implementations.
A Rdy-Ack run-time support for current CMP-based architecture is presented and its cost model is defined for message-passing and passing-pointer implementation models, both based on standard automatic cache coherence, optimized cache coherence solutions based on the use of the *home-flush* technique and on interconnection communications for synchronizations.

This latter solutions represent a lock-free run-time support which is able to provide better performance results with respect to solutions based on standard automatic cache coherence, especially for fine-grained parallel computations. Notably, we are able to reduce the number of memory accesses, cache transfers and synchronizations, and increasing computation parallelism with respect to the use of the automatic cache coherence alternative.

Finally, Tilera TilePro64 architecture has been used to provide a validation of the results obtained for the general case with the cost model. This architecture represents a good candidate for the evaluations of our solutions, due to the possibility of implement or emulates all our ideas in terms of cache coherence optimizations. Notably, with this architecture, in some cases, we are able to achieve an improvement of about 50% with respect to the use of the default cache coherence solution.

This thesis has to be considered a small, yet very important, part in our long-term project. In particular, the use of performance models is pervasive in our approach, as they are used both at compile- and at run-time: first to select the best implementation, and then to drive the adaptation policies [77] for dynamic run-time system. With this thesis we demonstrated the possibility of how a specific problem like cache coherence can be studied and modeled in order to capture the effects on parallel programs' performance. Both from the performance modeling and optimization point of view, many other aspects may be addressed in the future. In particular, we believe that a further study of the trends in hardware technologies and of new parallel paradigms definition and implementation is required.


[62] J. Jeers and J. Reinders. Intel xeon phi coprocessor high performance pro-

D. Shippy. Introduction to the cell multiprocessor. IBM J. Res. Dev.,

[64] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. Power7:

[65] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication

[66] Leonard Kleinrock. Queueing Systems, volume I: Theory. Wiley Interscience,
1975. (Published in Russian, 1979. Published in Japanese, 1979. Published in
Hungarian, 1979. Published in Italian 1992.).

[67] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu,
Jürgen Michel, Lionel C. Kimerling, and Anant Agarwal. Atac: A 1000-
core cache-coherent processor with on-chip optical network. In Proceedings of
the 19th International Conference on Parallel Architectures and Compilation
Techniques, PACT ’10, pages 477–488, New York, NY, USA, 2010. ACM.

[68] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo
queues. In Rachid Guerraoui, editor, Distributed Computing, volume 3274 of
Lecture Notes in Computer Science, pages 117–131. Springer Berlin Heidel-

[69] Leslie Lamport. Specifying concurrent program modules. ACM Trans. Pro-

[70] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C.
Sevcik. Quantitative System Performance: Computer System Analysis Using
Queueing Network Models. Prentice-Hall, Inc., Upper Saddle River, NJ, USA,
1984.

of the 4th International Workshop on Parallel and Symbolic Computation,
PASCO ’10, pages 18–19, New York, NY, USA, 2010. ACM.

[72] M. Leyton and J.M. Piquer. Skandium: Multi-core programming with al-
gorithmic skeletons. In Parallel, Distributed and Network-Based Processing
(PDP), 2010 18th Euromicro International Conference on, pages 289–296,
Feb 2010.


[82] Netlogic. The netlogic xlp processor family. http://www.netlogicmicro.com/Products/MultiCore/XLP.asp


