Synchronization Architecture in Parallel Programming Models

PhD Thesis

Arturo González Escribano

July 2003

Dpt. Informática

University of Valladolid

PhD Supervisors:

Dr. Valentín Cardeñoso Payo (Universidad de Valladolid)
Dr.Ir. Arie J.C. van Gemund (Delft University of Technology)

Copyright © 2003 by Arturo González Escribano

To my family, the small one and the big one

" ni tibi caveris, istud
non sinet intactum chaos, Antipodumque recessus,
alteriusque volet naturae cernere solem [...]"

"if you're not on guard, he will disturb this chaos of
yours, the Antipodes deepest places, he will try to
discern another world's sun [...]".

"Alexandreis" sive "Gesta Alexandri Magni", 1855
Gautier de Châtillon

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I want to thank my parents, who started my education, those who taught me how to act, how to be, how to learn, how to search in my soul when lost, and those who forgot so few things to teach me when I was fighting for adulthood; my brother, who led me in the way of watching, dreaming and questioning; and with all my love, my grandmother Dolores, who taught me many things about the real meaning of life, especially the secrets of laughther, sadness and joy.

Now, I include here both professional and personal acknowledgements, in no specific order: I want to thank Miguel and Fina for helping me to start over and for fighting for good sense, no matter the troubles around; Pablo and Ceci for making common life interesting over and over again; Marco for being a rare light in the depths of night; Jenny, who supported me with her sweet and pleasant company; Jesús for being a stone in the eye of a hurricane (even in his hurricane); Felipe who lives in our minds, dreaming our dreams; Barbi with all my tenderness, for bringing a marvelous scent back into my life; Fernando, Toro, Beltza and Sito, never to forget why; Rafa who is always around even when far away, and who is a remarkable man; Juan and Loli for being worry, so worry, so much worry, that even they tried to press me; Isidro for being such a normal person; Ramon for sharing his insanity with mine at crazy times; Carmen, Roberto and other friends for keeping me on the line; many people from GMV just for being around with their cheerfulness; César for reminding me what a normal pleasant life was, and for so many nice times in Delft; Rebeca for keeping my life warm in a chilly age; Cristina, Rosalía, Toño, Dani, Ramón, Alfonso, Zacarías and so many other nice people to share things when far away from home; Chus, for keeping me part of her life; Andrei and Lin for their help and for their professional understanding; Catherine and many other friendly people from the EPCC; Mike O'Boyle, for sharing his experience with me; Mark Bull, for being always at hand, even keeping no time for him; Oliver for his support in hard times; Georg because he was "interested in everything that was interesting", no matter if one hundred miles away; my department colleagues, those who helped me, those who worked with me, those who talked with me about my work, those who asked me a thousand times if the thesis was already finished, and those who were around when needed; especially Diego for his support and help, not only proffesional, during this difficult last year; Valentín for sharing his glitters of wisdom and for both: his stubbornness and his steadiness; and last but not least, I want to thank Arjan, for being a mentor, a colleague and a friend from the beginning of this work, someone who experiences life with joy, and so many times a lighthouse in the middle of the mist.

Nevertheless, the full list of people to thank will be too large to be included in such a small volume. If you should be here and you are not finding your name around, you know you are close to my heart.

The author would like to acknowledge the support of the European Commission through EARI grant number HPRI-1999-CT-00026 (the TRACS programme at EPCC); the Rekencentrum, Rijksuniversiteit Groningen, and its staff for their support during the EC summerschool of 1998 (TMR program) and afterwards during our use of their machine. This work was partially supported by CICYT grant contract no. IN97-65, for the use of the Origin2000 machine at the University of Valladolid.

Chapter 1

Introduction

He holds him with his glittering eye–
The Wedding-Guest stood still,
And listens like a three years' child :
The Mariner hath his will.

The Rime of the Ancient Mariner, 1798
SAMUEL TAYLOR COLERIDGE

1.1 Motivation

The *effectiveness* of a parallel application has been traditionally measured only in terms of its achievement of high performance as compared with its sequential implementation counterpart. From this point of view, the typical scenario has been one of a *high performance quest*, especially in the field of scientific computing. This situation is quickly changing nowadays, since general purpose parallel machines have become an affordable alternative to classical supercomputers, and network connectivity improvements have enabled parallel computing based on heterogeneous clusters, NOWs, and GRIDs. Concepts such as portability, predictability, evolution or correctness, genuinely related to software development methodologies, play now a role as important as performance improvements. As a consequence, the construction of high-quality parallel software at a reasonable investment of effort has become one of the main objectives in the development of parallel applications. Software construction methodologies, verification, ease of debugging, interoperability and reusability become key challenges for new generic supercomputing environments [52, 169, 175]. The continuous hardware evolution, and the lack of an established and commonly accepted parallel computing model or reference architecture results in a maturity level identical to the one in sequential computing before everybody assimilated the compatibility

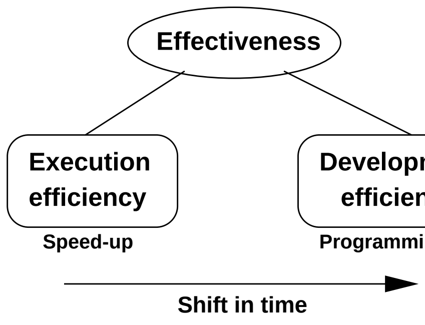


Figure 1.1: Shift of interest in parallel development efficiency

ideas imposed by the introduction of widespread common architectures based at the more abstract level on the Von Neumann model [134]. The development of quality parallel software is compromised by the lack of well-established models and by the common design techniques for the low-level tuning which is necessary to get maximum performance.

In the parallel programming field an important research activity is thus focused on the introduction of tools and methodologies for parallel software development. The objective is to create an appropriate framework to develop *effective* parallel applications (efficient and portable). However, parallel computing challenges are being faced from three basically different perspectives which are not yet mature enough as to bridge the gap between them. These are the following:

Parallel semantic models

Formal models of parallelism have been proposed and studied for a long time (see e.g. [165, 166, 190]). They are aimed to reason about correctness and concurrency possibilities. However, the models that explore unrestricted synchronization concurrency reveal many undecidable problems. Their inherent complexity prevents formal analysis and the amount of tracing information about a system evolution becomes intractable. As a consequence, there are no practical programming languages or environments which fully integrate the complete set of formal properties prescribed by these models.

Parallel computation models

Parallel computation models were introduced as a means to reason about computability, and to derive complexity measures of parallel algorithms. In sequential programming, a Turing machine is a universal model whose complexity measures are not modified, but in small constant factors, when an implementation is generated using the bridging model proposed by Von Neumann. However, in parallel computing, complexity measures and performance prediction are complicated by the lack of an established cost-theoretic model or a widely accepted bridging model for parallel computers [122]. As a consequence, parallel computation models proposed are either too abstract, too worried about implementation subtleties, or even too restricted to describe many real situations.

When the abstraction level is too high, the implementation in a real machine requires complicated transformations, specific for any new architecture and in general not decidable, that can modify the complexity orders of an algorithm (e.g. PRAM model). At the same time, too abstract parallel models do not encourage programming techniques that deal with synchronization problems.

On the other hand, some models are focused on modeling the low-level details of communication/synchronization costs (e.g. LogP). In these cases, the synchronization structures are completely unrestricted and analysis problems arise. The software development is not intuitive and the cost model (if not untractable), cannot be used in reverse to determine which heuristic implementation decisions produce given results.

Finally, there are models designed to provide a convenient and simple cost model. However, they use unnatural synchronization restrictions that limit the types of algorithms that maps directly into the model, and no clues are given for the mapping of other types (e.g. BSP).

Parallel programming models

Given the previously discussed problems, an important part of the parallel programming community is focused on the development of practical programming interfaces that allows the programmer to exploit the parallel and high-performance characteristics of actual machines. In search of the maximum expressive power and flexibility, many of these programming models allow the creation of structures that are dynamic, complex, or impossible to analyze. Poorly structured synchronization is the origin of many current difficulties of parallel programming [89]. For unrestricted synchronization structures the optimal mapping (problems to programs) is almost humanly impossible, cost models are not affordable, and good scheduling algorithms are extremely expensive. The programmer must take low level detail decisions, including sometimes machine dependent optimization solutions hardwired in the code. On the other hand, other models sacrifice expressive power, restricting the synchronization structures available, so as to keep analysis properties that lead to better cost models and formal developing, verification, mapping and debugging techniques.

From the above discussion, we identify the synchronization structures available in a parallel system as the key factor for a trade-off between expressive power and engineering ability. This trade-off has been, for a long time, an issue of still on-going debate. In this study we introduce the concept of *synchronization architecture* to classify the synchronization structures in terms of their properties related to software engineering and high-performance. This new approach lead us to identify the minimum restrictions needed to bound the complexity of relevant analysis problems, and to evaluate the potential problems to express parallel applications into these restricted structures.

1.2 Parallel programming models

As already pointed out, a *parallel programming model* (PPM) can be defined as a programming interface which can be targeted to any computer architecture and lets the programmer express parallelism in terms of a set of primitives given by the underlying parallel model of computation. There exist PPMs from the highest (formal specifications) to the lowest (close to implementation details) abstraction levels.

Many new design decisions take part in the creation of a parallel programming interface. Compared to sequential programming, new degrees of freedom are to be considered. Parallel computations are much more complicated to create, track and analyze. The mapping of a parallel algorithm to a program is a complex task. The resources-to-activities assignment (scheduling), and the partitioning of data or activities that minimize communication costs are optimization problems in the tuple *(time, space)*, typically translated to graph problems, which optimal solution are not known or are NP-complete [4]. A generic, heterogeneous and evolving framework requires flexible mapping methods to create efficient and portable applications. Thus, abstraction is a more important feature in parallel than in sequential programming. On the other hand, a parallel programming model is not practical if it proposes a so abstract interface that it is too difficult or expensive to find efficient ways to implement it in real hardware architectures (existing or evolving). The choice of a parallel programming model involves a trade-off between portability and efficiency.

We distinguish two main categories of characteristic for a PPM. They are related to two properties of the semantics involved in the model (see Fig. 1.2). A model should propose an interface abstract enough to minimize the human effort to learn it and use it for software developing (SEC, *software engineering characteristics*). At the same time, the model should be simple and close enough to the low level details to make it possible efficiently bridging programs to actual parallel computations in a machine (AC, *analysis characteristics*). However, expressive power may be lost if too much simplification or abstraction is used to improve the quality of these SEC and AC characteristics.

The search of a convenient programming environment is the current Holy Grail quest in parallel computing. The problem has been approached for a long

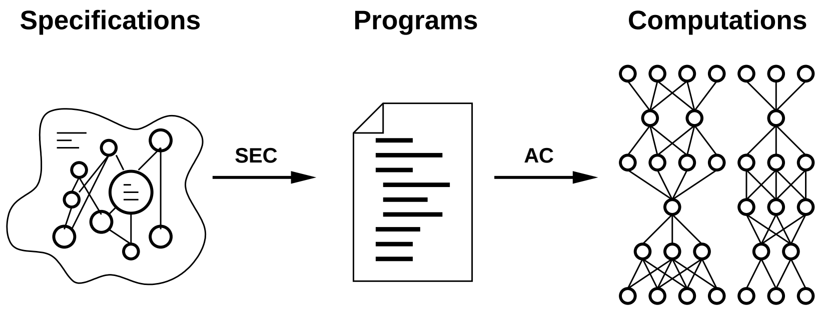


Figure 1.2: Software engineering (SEC) and analysis (AC) characteristics.

time. In the 80s, there was an important gap between theoretic parallel algorithm design (mainly based on PRAM model) and efficient implementation on real machines. Data-parallel languages and automatic parallelizing compilers based on data analysis were the major trend of research for easy and efficient parallel programming. Nonetheless, the restrictive data-parallelism model and the impossibility of reconstructing certain parallel structures from sequential code[1] led to the development of alternative systems for direct and free expressions of parallelism (mainly message-passing interfaces), that were failing to provide analysis characteristics [89].

In the 90s, the introduction of the restricted but portable and cost driven computation model BSP [185], and the more detailed but less restricted communication cost model LogP some years after [49], brought new attention to parallel programming models in general, and to those aimed to cost measurement in particular [16, 57, 76, 133, 161]. Performance modeling became an important issue. In the middle of 90s, the performance analysis study of Van Gemund lead to the introduction of the parallel programming model SPC [70, 71]. In this model the dependence structures that can be generated are restricted to nested-parallelism structures (those that can be represented by a Series-Parallel (SP) graph [184]), extended with a simple contention mechanism. This basic restriction in the synchronization structures allowed by the model is related with the possibility of using a new performance analysis technique, with adjustable accuracy in terms of a machine model. SPC provides a simple parallel software development framework. However, some synchronization structures are not directly representable in nested-parallelism, and they must be reprogrammed, possibly adding dependences that were not in the original problem. Thus, the utility of a restricted

[1]When a naturally parallel application is programmed in a sequential model, the concurrency *space* is *compressed* in only one point of the concurrency axis. Concurrent tasks are pushed into the time axis, imposing an arbitrary order on them. Thus, important information about concurrency could be lost.

synchronization programming model is endangered by the potential loss of performance in certain type of applications. We will come back to this important issue along the next section.

The study and comparison of unrestricted and restricted parallel programming models (like SPC) brought to light an important feature of PPMs. Most of their properties (SEC,AC) are related to the ability to detect and evaluate properties of communication and synchronization. Restricted models that have good analyzability characteristics for both communication and synchronization, can achieve all the SEC and AC requirements.

1.3 Synchronization Architecture

We propose the concept of *Synchronization Architecture* (SA) to be the abstract description of the synchronization structures and communication processes which characterize a given PPM, together with their fundamental properties. PPMs can be classified in terms of their synchronization architecture, defined by the mechanisms which are provided for expressing synchronization, and the structures that can be created by them.

To classify parallel programming models, Skillicorn and Talia have proposed the following criteria [174]. (1) PPMs with support for dynamic process or thread structures; (2) PPMs with only static process or thread structures, but no syntactic limits on communication; (3) PPMs with only static process or thread structures and syntactic limits on communication. They claim that: “Models that allow dynamic process or thread structure cannot restrict communication [...] even models that restrict communication within a particular syntactic block cannot limit it over the whole program. Thus such models cannot guarantee that the communication generated by the program will not overrun the total communication capacity of a designated parallel computer. [...] some programs that can be written in the model will perform badly, and it is not straightforward to detect which ones”.

Although we do agree this is a good candidate for a general classification of PPMs, we think that it can be clearly refined. In fact, restricted synchronization models can impose structure on the way threads are created and synchronized, to derive dynamic but restricted thread structures with a predictable number of communications. Even for some dynamic thread structures, it is still possible to obtain cost measures and find an efficient way to map the computations onto the machine, as shown in Table 1.1.

Thus, we propose new more detailed criteria, including different classes of synchronization restrictions. First, we distinguish two types of synchronization [9]: (1) Condition synchronization (CS), which implies an order to be preserved in the execution of two tasks or statements, and that is typically associated to *data*

(a)

Thread structure	Communication	
	Restricted	Unrestricted
Static	Predictable	Predictable*
Dynamic	Unpredictable	

(b)

Thread structure	Comm. & Synch.	
	Restricted	Unrestricted
Static	Predictable	Predictable*
Dynamic	Predictable*	Unpredictable

* Depending on the exact structures and restrictions of the model

Table 1.1: PPMs classifications: (a) Skillicorn & Talia [174], and (b) our proposal.

dependences or *communication*; and (2) mutual exclusion (ME), that prevents that two tasks or statements to be executed at the same time, although the order in which they are finally executed is not relevant. These types of synchronizations can be considered orthogonal, in the sense that a PPM can support one or both of them independently. Some models simply lack any form of explicit synchronization (e.g., HPF [1, 27]), some do not provide any explicit dynamic synchronization mechanisms (e.g., Fortran-M [67]), while others impose restrictions on the form of the static synchronization structures (e.g., BSP [185]).

In section 2.2 we will introduce a classification of the synchronization space in terms of three different characteristics: (1) CS structures to be allowed, (2) the ME mechanisms, and (3) the data-dependence of synchronization structures. In particular, we distinguish two complementary classes of CS structures: one unrestricted and one restricted to a specific compositional form called *nested-parallelism*, SP or *Series-Parallel*. SP structures are restricted to nested-parallel task control structures or, in other words, to the recursive application of primitives with the semantics of *cobegin-coend* [9]. Models which allow only SP restricted structures are called SP-models (e.g. BSP, SPC). The associated task graph of these structures is in the class of Series-Parallel graphs or SP-graphs for short.

Beside the previously mentioned engineering aspects (SEC) the introduction of restrictions on a PPM's synchronization architecture has a favorable effect on its analysis characteristics (AC). For instance, improved scheduling techniques have been designed for SP restricted DAGs [63, 20, 142, 159, 157]. One of the reasons behind their advantages is that the number of edges in an SP graph is bounded to be a linear function of the number of nodes ($O(|E|) = O(|V|)$). Another reason stems from the compositional nature of SP-graphs, which allows a recursive local analysis of properties. Thus, many scheduling algorithms show a very low complexity measure when applied on SP graphs. Moreover, many combinatorial problems which are NP-hard for generic graphs are known to be linear on Series-Parallel graphs [179]. Cost analysis (e.g. critical path analysis) is also improved. Even more, analytical closed form cost expressions can be derived for SP graphs [71]. At a more abstract level, there exists a formal algebraic characterization of the languages constructed on SP semantic models (called *SP-languages*), with their extended recognition automata [125, 126, 127].

However, the restrictions imposed on the static synchronization structures do eliminate some expressive power from the model. In some situations, tasks that could be theoretically executed in parallel must be serialized as a consequence of nested synchronization. This could lead to a performance loss which, unless carefully estimated, would clearly compromise the use of these models in parallel programming.

To illustrate this point, let us consider the task graph associated to a 1D cellular automata with just 3 cells, where a function dependent on parameters evaluated at neighbor cells (the ones given by a stencil) is applied in parallel across all the cells along 3 consecutive iterations. In Fig. 1.3*(a)* a task graph associated with a generic PPM computation is presented. Each edge represents a communication or a synchronization. The version in Fig. 1.3*(b)* is a task graph associated with the choice of a model which restricts communication so that it can only synchronously take place at a barrier synchronization. The dashed line represents the barrier. The black nodes can be executed in parallel in the first example, but are serialized in the second. If execution times of the black nodes is $t_b = 10$, and white nodes $t_w = 1$, the total execution time (without communication costs) is $T_1 = 12$ in the first example, and $T_2 = 30$ in the second one, which give a performance penalty of almost 3.

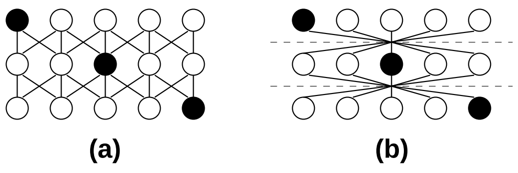


Figure 1.3: Example of parallelism loss at programming level

This potential loss of parallelism is introduced at a programming level when in the design phase, as a consequence of the restrictions of the abstraction level we are using to describe the problem, and it will be readily propagated through the following development phases. We are specifically interested in any potential computation time penalty forced by the restricted expressive power of a PPM. On the other hand, the quality of the low-level implementation phases can be improved with restricted CS models. Specifically, SP-restricted programming shows interesting features for scheduling and mapping, not found in non SP-restricted models, as mentioned earlier.

1.4 Problem statement and thesis

From the previous discussion, a number of interesting open questions arise which we will address in this work. It is not yet clear what *type* of programming models are more convenient for nowadays and near future parallel programming. It will be highly interesting to find objective characteristics that we can use to evaluate or classify the potential benefits and drawbacks of a given model. As mentioned earlier, Skillicorn and Talia proposed in [174] a set of interesting properties for ideal parallel programming models that promotes low cost software development and maintenance, efficiency and portability. According to them, a model should "be easy to program, have software development methodology, be easy to understand, guarantee performance, and provide accurate information about costs". These criteria are mainly subjective, and can be difficult or impossible to agree about the adequacy of a given model to it. On the evolution of future parallel programming models clear directions and requirements must be proposed. Theoretical comparisons between well-known parallel computing models has been shown (see for example [16, 161, 128]). However, no rationale has been offered in the more abstract level to explain the similarities and differences, Quantitative evaluation of parallel programming models has been tried previously focusing on efficiency and performance evaluation accuracy [114]. Related design characteristic are studied in [112]. Although the experimental approach is similar to ours in the low level, we are more interested in determining the origins of these quantitative differences at more abstract levels and to predict the effect of design decisions in parallel programming languages and models in both, software engineering and analizability characteristics (SEC,AC).

The problem we want to tackle is detecting any relation between the SA of a PPM and its software development and analizability characteristics, in order to present classification criteria of SAs it terms of their characteristics for parallel programming. If restricted SA classes appear to have advantages over unrestricted classes, a related question is if there are methods to map known applications in unrestricted SA classes to restricted ones, and how much performance impact may impose such high level transformations.

In the thesis proposal we are presenting here, we identify, first, the synchronization architecture (SA) as a key property of PPMs with respect to its suitability for software engineering and analysis, and a good criterion to classify PPMs. Some classes of restricted SA leads to good characteristics in software engineering as well as analysis, while others prevent them. The most important feature of an SA is the class of condition synchronization it allows (NSP vs. SP).

The choice of a restricted SA may entail a loss of parallelism at the programming level of abstraction (possibly propagated to lower levels). We propose an empirical evaluation system of PPMs to grade them in terms of this loss of parallelism as a function of their SA. Based in our value system, we promote the class of SP-restricted PPMs as a promising PPM for general-purpose parallel computing. SP restricted SA models present a good trade-off between expressiveness and software engineering and analizability characteristics. Moreover, we show that most applications can be mapped to SP (nested-parallelism) structure with minimal performance impact.

1.5 Approach

In this dissertation we study the problem from three different perspectives. A conceptual review of the SA of parallel architectures, computation and programming models, programming languages, and applications is needed to identify the best criteria for classifying the synchronization structures found at any detail level of a parallel system. Once the classes are determined and the restricted SP class arises as the class with the most promising features, a further study of the properties of its structures is needed. Then, the second step is a theoretical study, based on graph theory, of the properties of NSP and SP structures, including an study of the transformation of structures in different classes. The third step entails an experimental framework: ideas and techniques developed on the theoretical study can be used to experimentally compare the behavior and performance of application structures in different classes. This empirical study validates results and proposals analyzed in the previous steps and reveals the real parameters and behavior of real applications when programmed in different classes of SA.

Conceptual approach: After defining the SA concept and establishing the different abstraction levels of study, it includes a classification of the SAs found at any level: From parallel architectures and well-known parallel computing or programming models to the applications space. In this approach we relate the SA class with the programming models expressive power, analysis characteristics and the virtues and flaws associated for mapping applications to them. The NSP vs. SP classification appears as the more relevant feature of a PPM. We also present a conceptual discussion of the possible mapping strategies of applications, to PPMs in a different and more restricted class.

Theoretical approach: This approach is based on a theoretical study of the modeling capacities and restrictions of SP models in an abstract level, and their significance. A formal analysis of the NSP and SP graphs, their relation and the distance from NSP to SP forms is introduced. We present heuristic transformation techniques and algorithms to convey NSP structures into SP approximations that introduce minimum changes in topology or performance. We develop an analysis framework to predict the loss of performance introduced at the programming abstract level as function of SA. The framework is based on the use of graph theory, topology classes, and task workload metrics. We measure performance differences in terms of critical path.

Experimental approach: We present a comparison of using programming models or languages in different SA classes to implement real applications, including the effects of typical implementation trajectories. Here we do not restrict ourselves to the highest abstraction levels, but we use the above framework to discuss the performance effects of various mappings and implementation issues at lower level. Thus, two different frameworks are studied:

1. Oriented to the whole program space.

   We study the results of enforcing SP restrictions on a sample of the whole graph space, and on synthetic graphs, relating the modeled performance loss to generic and simple graph and workload parameters.

2. Oriented to applications.

   Based on our parallel applications classification presented in the conceptual approach, we select a collection of representative applications from all relevant SA classes. We compare execution times and performance effects produced when real codes programmed in generic and restricted SP models are run in several machines models. The impact of SP restrictions is empirically predicted and compared with previous results.

As we will show along the following chapters, the results of this three approaches will fully support our theses: presenting the SA as the key factor in the analysis characteristics of a PPM, and consequently in the software engineering of parallel applications, and promoting the SP restriction of condition synchronization as one of the most relevant choices in the design of a PPM.

1.6 Outline

This dissertation is organized following the three different approaches presented in the previous section. Chapter 2 presents the conceptual approach. After introducing some concepts and terminology (including SA), we present our classification criteria for SA. Then, we travel bottom-up along the different abstraction levels studying the SA of parallel architectures, programming models and applications. Interesting conclusions are discussed for each new layer. In Chapter 3 we present the theoretical approach. We formally define SP graphs and study their structures. Simple transformation techniques and problems are discussed, after which two new heuristic algorithms are introduced. The impact of the transformation is studied from different perspectives. Chapter 4 includes an exposition of the motivations and decisions taken to build our experimental framework. Graph application modeling techniques are introduced in this phase of the study. A broad summary of the results obtained in each phase is presented and discussed. In Chapter 5 we recall the results and ideas presented along the whole work, and we present our conclusions.

Chapter 2

Conceptual approach

I did not paint it to be understood, but I wished to show what such a scene was like.

J.M.W. Turner, 1775-1851

This chapter is an attempt to bring the reader a travelogue through the parallel programming world. After the introduction of some concepts and terminology, we will initiate a trip along the fields of parallel programming, a land full of sight spots where the synchronization structure colors are showing up from inside everything that blossom. From the rough and changing oceans of parallel architectures and low level execution models, we will fly up to the low-lands of mapping, where the implementation floods are directed by the river coasts of compilation. In the upper valleys we will find the programming models which allow this compilation techniques and the abstractions that hide the details to the programmer. Finally, we will climb up the high abstraction peaks to find their snow crowns, where applications and parallel algorithms dwell, nurturing the waterfalls where all the implementation line begins. All around, synchronization structure will be a friendly guide that will show us secrets beneath what the untrained eye catches. Throughout this trip we will learn how synchronization structure helps us to understand the roots of advantages which show, and the difficulties to be tackled, when different parallel programming models are used.

First, we will discuss about models and modeling, to propose general definitions for parallel programming and computing models, and describe the different detail levels involved in parallel computing. Then, we will introduce the *synchronization architecture* concept, presenting classification criteria, useful for detecting the good and bad properties of synchronization structures regarding analyzability and expressive power. These criteria are used in the following sections to classify models and applications, showing the relevance of synchronization architecture at any level of detail.

2.1 Parallel models and definitions

In this section we begin to prepare the luggage we will need for our trip. We establish some terminology that sometimes have confusing meanings, typically when coming from different communities related to parallel computing. We also define the main concepts about parallel programming models we will use from now on.

2.1.1 Parallelism and parallel computing

Although parallel computing is somehow a complementary concept to sequential computing, they share a main substrate. They solve problems applying a *programmed* collection of actions, chosen from a reduced set, where each of them modifies a well-defined environment in a deterministic way.

The important difference between parallel and sequential worlds is how these actions interact with the environment and how they are ordered in the time space. In sequential computing the programmer is responsible for the order in which the instructions are executed and only one of them can modify the environment in a given instant of time. When the restrictions of time order are relaxed, and many (a given number) of actions can be executed simultaneously or in no specified order, the programmer has new freedom degrees to exploit, but she/he faces new associated problems. When two given actions modify an initial environment state in non-compatible ways, they must be prevented to execute simultaneously to preserve the consistence of the computation until it arrives at a known state where the problem is solved. Thus, we distinguish actions that **can** be executed in *parallel* and actions that **must** be executed *sequentially.*

We call *parallelism* to the possibility of exploiting time ordering relaxation and simultaneous execution of actions for problem solving. Thus, *parallel programming* is related to uses and techniques to express a solution to a given problem in a computational environment where parallelism is possible. And *parallel computing* refers to the evaluation of solutions in such environments.

Parallel computing and programming is historically associated with high-priced machines and high-performance. However, parallelism is a broader word that can be associated with many terms, most of the times with unclear boundaries among them. For example:

Concurrency: Typically associated with the basic problems of parallelism, like analyzing *mutual dependences* and using *synchronization mechanisms* to access *shared resources.* Sometimes, concurrent computing refers to mechanisms of time-sharing to provide simulated parallel execution in multitask environments with restricted number of processing elements.

Distributed computing: More related to the techniques for using parallelism in environments where active elements are loosely-coupled and/or have diverse nature.

High-performance computing (HPC): While mainly using parallelism, HPC is focused on the extraction of high performance peaks from specific (scientific) applications with new or recent computing technologies.

Parallel computing: Mostly related to the programming and use of real parallel architectures, where several processing units operate with a hardware or software layer that allows interaction among them.

We will use *parallel computing* in the broadest sense, referring to the exploiting of parallelism in any computational environment.

2.1.2 Modeling

Human beings use *modeling* or *models* to abstract reality in order to represent it in a simplified way which allows them to reason about it, developing theories. However, the exact meaning of the terms *model* and *modeling* depends on many issues related to the nature of what is being modeled, the purpose of the model, the level of detail required and the intended techniques to be applied. Thus, talking about, and especially defining what a *programming or computing model* is, is technically difficult, as different people understand or think differently about them.

In the parallel computing world, there does not yet exist a reference architecture or programming model accepted as universal. We present in this section a tentative distinction of modeling levels and their relations in a parallel computation environment, that will be useful for conceptually analyze both, well-know models of parallel computation, and the scope of our study.

The first distinction we must introduce is that programming and computing are not the same thing. While *programming* is the activity oriented to express or prescribe a solution to a problem (or family of problems) with a constrained set of actions, *computing* is the activity oriented to evaluate programmed solutions in a computational environment. Thus, *parallel programming models* and *parallel computing models* are not exactly the same, although the boundaries are blurred, as programming and computing are inter-related activities. Many models partially cover aspects of both programming and computing activities, and they try to fill in the gap between them. They are usually called *bridging models*. We will discuss about them in the next section.

2.1.3 Parallel computing and programming models

Now we will walk through the abstraction levels, from the highest to the lowest of the computing/programming activities, giving names to what we find on our way. The reader can follow our trip in Fig. 2.1.

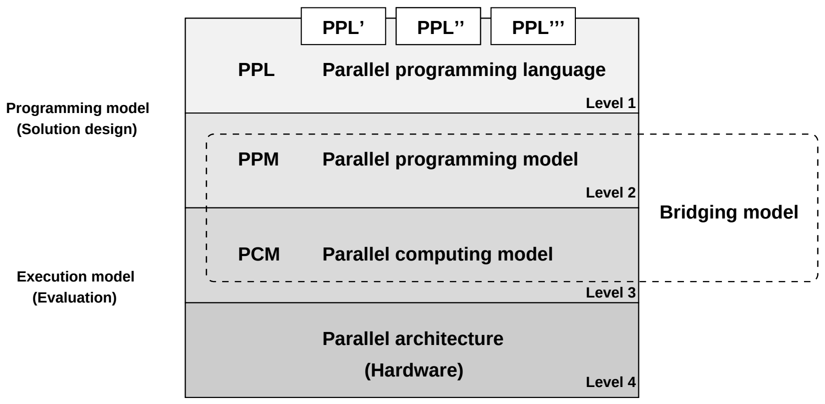


Figure 2.1: Abstraction levels of modeling

The reality that we try to model is the programming and computing task with real machines able to use parallelism. The programming task is done by implementing an algorithm or application specification in a programming language with capabilities to express parallelism. A programming language is an instance (with given words and syntax) of a programming model (which provide specific semantics).

Thus, a *Parallel programming model (PPM)* is an abstract description (or abstract virtual machine) to express parallel actions independently of the underlying execution level. Message-passing interfaces, concurrent object-oriented programming and other similar tools are mainly focused at this level. We call an instance or specific notation for a PPM a *Parallel programming language (PPL)* Thus, a PPM defines a family of PPLs (a family of possible languages that allow the programmer to express exactly the same parallel semantics).

A complete *virtual machine* includes the definition of a basic information unit and a concise instruction set with clear semantics. In parallel programming the instruction set must include synchronization operations. The reader must notice that a PPM defines an *abstract virtual machine* that in fact induces a computation model. A PPM is a programming interface that hides some execution details and issues of the underlying *execution model*. That is what we call a *Parallel computing model (PCM)*. An effective parallel computing model must be affordable to be efficiently implemented in real parallel machines and many times it is highly influenced by real architecture model capabilities. At the same time, it is expected that a good parallel computing model provides a reliable cost analysis technique to test the behavior and performance expectations of programmed solutions. Examples of this level include abstractions such as PRAM [65] or LogP [50] models.

Finally, at the lowest level, machines are abstracted by *parallel architectures*. They propose a model for hardware capabilities of the machine, defining a kind of low-level virtual machine. A parallel architecture may include a specific machine description (e.g. The Connection Machine [103]), or a more generic model (e.g. NUMA architectures [51], Beowulf systems [177]). PPMs and PCMs should be abstract enough to provide an easy interface to the programmer, and at the same time, they should be portable (efficiently implementable) across the most relevant parallel architectures.

For example: MPI [48] or PVM [178] are different languages (level 1) that implement the semantics of the same PPM: Message-Passing (level 2). Message passing assumes an underlying PCM based on a bounded number of processors running asynchronously and exchanging point to point messages, such as LogP (level 3). At the lowest level, such a PCM can be directly implemented on a NOW, a cluster or even in a shared-memory architecture (level 4), with possibly different low-level implementation mechanisms on each.

PPMs and PCMs are highly related. Since most of the times they are different only in the point of view (from the programmer or from the implementation level), they share many common problems, and the solutions to them may be similar. This is the reason why nowadays there exists a wide concern about *bridging models* for parallel computation. These models include the main features of a PPM, offering a high-level parallel programming interface, and give details about performance cost modeling and low-level implementation issues associated with the PCM (typically representing a given architecture or real machine with a small number of parameters). They try to jump over the gap between two communities: The architecture design community (concerned by efficiency and implementation) on one hand, and the parallel solution design community (concerned with programming techniques and parallelism exploiting) on the other hand. Many models can be considered bridging models, although the concept was proposed with BSP [185]. We will review some bridging models and their characteristics in section 2.4.

In the following paragraphs we review other definitions and ideas found in the literature about what parallel programming and computing models are or should be.

Skillicorn & Talia define a PPM in [174] as an interface separating high-level properties from low-level ones. It is an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all the architectures below. It is designed to separate software-development concerns from effective parallel-execution concerns and provide both abstraction and stability.

A similar idea introduced by Maggs in [129]: A PCM defines an abstract execution engine, powerful enough to produce a solution to relevant classes of problems, which must reflect the salient computing characteristics of practical parallel computing platforms. The model is both, descriptive and prescriptive. It describes realistic platforms behavior, and at the same time it suggests hints and directions for new hardware development, as it models features desirable for interesting programming and computing techniques.

2.1.4 Detail levels

From the more abstract specifications of a problem solution, to the real implementation and program, there exist several detail levels that can be considered and included in a model. Most of them are clearly related to the abstraction level where they typically can appear. We introduce here a classification of these detail levels from McColl, as presented in [35]. From the maximum abstraction to the lowest level of detail, a PPM/PCM can include or model the following levels (we present some examples of models that include a given level):

Specification: Unambiguous description of a computational problem (e.g. Z, CSP, π-calculus).

Programming: Notation for a precise, high-level description of correct and efficient solutions to a given computational problem (e.g. HPF, Occam).

Cost analysis: Basis for evaluation and comparison of efficient methods for a programmed solution to a computational problem (PRAM, BSP, LogP).

Architectural (also called mapping level): Framework for the description of implementations of programs (e.g. monitors, semaphores, RPC, message-passing).

Physical (also called machine level): Description of a real machine characteristics in which to implement and solve a program (e.g. distributed-memory vs. shared-memory models, NOWs).

A programmer typically walks through these levels top-down during the design and implementation until the program can be executed in a real machine. The term *implementation* is sometimes used for the whole process of transforming a problem specification in real code for a given machine. In our framework, it typically means the process of transforming a *program* (specified in a PPM notation for the programming level) into a ready-to-run executable.

2.1.5 Requirements of PPMs

What characteristics should have a PPM/PCM to be a good candidate for general all-purpose parallel programming? We discuss here a proposal from Skillicorn and Talia [174]. They propose six main requirements for a PPM/PCM:

Easy to program: A PPM should conceal details about *decomposition* of the computation in threads, *communication* and *synchronization* between them, and any mapping decisions to adapt the computation to the underlying hardware model.

Software development technology: A firm semantic foundation is needed to bridge from specifications to programs.

Easy to understand: To educate existing software developers.

Architecture independent: Even with new evolving or future technologies. (In [129] we also read that a PPM should be somehow prescriptive, and point into new interesting directions for hardware development).

Guaranteed performance: Although it is not needed to exploit it to the best possible in each architecture, especially at the expense of much higher development and maintenance costs. "Implementations should aim to preserve the order of the apparent software complexity and keep constants small".

Cost measures: They should cover execution time, process utilization, development, etc. They must be compositional and convex.

These requirements can be divided in two broad categories. The first three requirements are focused on the software development characteristics (more related with the PPM), and the last three ones are focused on the good mapping characteristics (more related to the induced PCM). The achievement of the requirements depends on the modeling decisions taken in the design of a PPM/PCM at the different detail levels (see section 2.1.4). These decisions define the power of the model *expressiveness* and *analyzability*, being foundations of the feasibility of the software development and good mapping requirements categories respectively.

Specific restrictions at the programming level, that somehow reduce the number of applications that have a natural mapping from specifications to the structures accepted at this level, may produce benefits for the lower levels. Specifically, advantages may appear on cost analysis techniques and implementation transformations to map applications into the architectural level.

The programming model, formally, provides a set of rules or relationships that defines the meaning of a set of programming abstractions. Its objective is to allow reasoning about program meaning and correctness [129]. Thus, a model must be simple enough to allow analysis and stable software developing techniques. At the same time it must provide meanings to express problems in a natural way (obvious to any programmer), complying to the original specifications of the problem solution and obtaining efficient implementations and good performance in real machines. Such *mapping decision* should be helped by a performance cost model, based on a sufficient detailed but abstract enough machine model. Cost models that allow to plug different machine models in a standardized description language or formalism are the best candidates. The programmer may trade complexity and accuracy in the process to determine the best implementation of an algorithm for a given machine [70].

Our study is mainly focused on the cross relationships between the programming, cost analysis and architectural (or implementation) detail levels and their impact on the expressive and analysis power of the model. We have identified the synchronization structures supported in the programming model as a basic component of a PPM design. We have found it responsible for an important trade-off between expressiveness and analyzability, which are foundations for the two PPM/PCM requirements categories. This matter is discussed in the following sections.

2.2 Synchronization architecture

Applications that exhibit the same synchronization structures usually have properties that can be exploited through the programming and implementation pipelines. PPMs can restrict or support specific kinds of structures in order to offer advantages in software engineering, programmability and portability (automatic or interactive performance analysis, verification, etc.) Identifying important classes of programming structures with interesting properties becomes a challenge for parallel software engineering.

We propose the concept of *synchronization architecture* to classify parallel systems regarding its main synchronization structure properties. In this section we propose and describe a classification of the different main types of synchronization structures.

Definition 2.2.1 *A* Synchronization architecture *(SA) is the formal description of the properties that define the communication structures and synchronization mechanisms either present in a specific application or supported by a given PPM.*

2.2.1 Types of synchronization: CS, ME

Although several names are used in the literature, we distinguish only two main types of synchronization (see e.g. [9, 122]).

Condition synchronization (CS): It is used when an operation or process must be delayed until a certain condition is satisfied. It is typically associated to *data dependences*, *communication* or other processes ending. It implies an execution order in the processes or operations involved for the computation to be correct. It is also called *static, deterministic* or *event synchronization.*

Mutual exclusion (ME): A *critical section* is a sequence of statements that must be executed as an atomic operation. When two or more critical sections or processes cannot be executed at the same time (in parallel), we say they are *mutually exclusive.* If two or more mutual exclusive processes try to begin their executions, only one of them can proceed, but the order in which they are executed is not relevant for the computation correctness. It is also called *dynamic* or *non-deterministic synchronization.*

These types of synchronization are orthogonal in the sense that a PPM can support both or either of them independently. Nevertheless, they are only different from the programming point of view. In the execution model, the ME synchronization is transformed in CS, creating an order of execution for the mutual exclusive critical sections. This transformation is done by scheduling algorithms in the PCM implementation or directly by the hardware (e.g. through communications contention). The difference is the freedom for the critical sections to be scheduled in any order, that allows the underlying execution layer to detect or apply a different order for a particular execution of the code. This order is chosen to maximize the performance and must be determined by the computation status, the execution times of other tasks, and previous scheduling results.

2.2.2 Mutual exclusion, mapping and bounded resources

In this section we discuss the relation of ME nature with mapping tasks at low levels of detail. Thus, ME appears to be highly related to implementation details oriented to deal with restricted resources. Processors are typically a restricted resource. The discussion evolves to the relative importance of supporting bounded or unbounded number of logical processing elements in a PPM, that is related to the parallelism granularity supported.

A PPM must include CS mechanisms. Although some problems can be solved with only ME, there are many others whose solutions need an specific order of execution in some operations for the computation to be correct. On the other hand, ME exclusion may be implemented by a programmer in terms of CS creating an unnecessary order in the tasks involved. The programmer faces the risk of degrading performance if the order chosen is not the optimum for a specific execution of the program, but many times she/he has a good heuristic to decide what should be an acceptable order. Furthermore, many times ME is introduced by programmers to solve mapping problems in environments where the PCM or execution model cannot solve them directly.

The main purpose of ME is to let the programmer deal with restricted resources. These resources can be of any nature, but they are intrinsically related to the architecture, model, or design of real machines (e.g. shared-memory accesses in shared-memory architectures that do not provide an implicit contention mechanism). At the more abstract level of specification ME scarcely appears. Only when the programmer (or compiler designer) is facing mapping problems, considering a restricted number of resources (e.g. a restricted number of processors), ME becomes really important. Explicit ME can be used by the programmer to annote the tasks which can produce contention problems, for its implementation in architectures that do need it, and for being used in a cost model during the mapping.

Consider the following example. A classical parallel solution for load balancing in many irregular problems is the *farm* paradigm, also called *work-stealing* strategy (see e.g. [43, 189]). In problems solved with this strategy, there are k work providers and n workers. The workers repeat a simple cycle until the computation is finished: Get work from a work provider and do the work. The work providers act as resources that must be accessed through contention by the workers. See a graphic representation of the generated structure in Fig. 2.2. For work balancing reasons and simplicity, in most examples there is only one centralized work provider $k = 1$. In some applications the work done can produce many other pieces of work to do in the future which are sent to a work provider when the exclusive access for this operation is obtained.

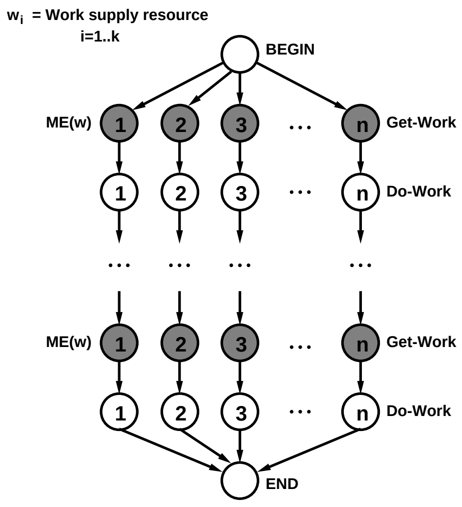


Figure 2.2: Workers-Farm scheduling strategy

This description corresponds to a mapped and free scheduled solution for the problem. The original problem only considers that many workers can do pieces of a job in parallel. The original problem solution only specifies that many workers can get a piece of job and do it iteratively. If the number of processors is not bounded, then, m simultaneous workers can do one of the many m pieces of the job. If the work produces more m' pieces, then, m' new workers in m' new processors can start processing these new m' pieces, as soon as they are available. Nevertheless, the number of processors is typically a limited resource. When mapping to n processors one logical choice is to start n workers, and let them process the pieces of work iteratively. In a second mapping phase, if we consider $k = n$ work providers, each worker has an exclusive font of work pieces, and the computation does not need mutual exclusion. However, for the kind of irregular and data-dependent problems that this strategy is oriented to solve, a worker can produce many more pieces of work than others. We want to balance the load such that no worker is idle while others have still many items to process. Thus, workers that become idle should contact the work providers to get more job pieces. One or more centralized sources of work are needed, acting as resources and needing mutually exclusive access to avoid several workers creating race conditions when downloading or uploading job pieces.

This load balancing strategy is a mapping decision that works appropriately when the computation is highly irregular and the computation time of a worker doing a job piece compensates the communication and contention delays. The number of work providers can be selected depending of many cost factors and load predictions. All these mapping issues are faced by a programmer implementing a farm directly, when the original problem definition was much simpler. In fact, the original solution structure is hidden or even lost in the mapped-scheduled code generated. We argue that this mapping decision must be postponed to the mapping phase, done by the PCM implementation, guided by information provided by the programmer either, on the code or interactively.

An interesting question derived from the previous discussion is whether a PPM should force the programmer to work with a fixed number of logical processors or with an unbounded number of them. As is discussed in following sections about existing PPMs, working with an unbounded number of processors allows the programmer to exploit the maximum level of fine-grain parallelism in the problem. However, in most situations, this is not an efficient solution in real implementations. Fine grain parallelism can create a huge amount of small tasks with too frequent communication, reducing the *parallel slackness*[1] and unbalancing the communication/computation ratio, incrementing the communication costs over the computation. On the other hand, if the programmer must take in account that he is working with a fixed number of processors, sometimes he is lead to deal with this restricted resource directly, facing and solving the data partition, scheduling and other mapping details. This could compromise portability and the possibility of using powerful software development techniques. In section 2.4 we discuss PCMs that try to face this problem from different points of view.

[1]The granularity of the computation partition among tasks [185].

The best solution is to find a good mapping technique that transforms fine-grained parallelism expressed in an abstract form by the programmer in coarse-grained parallelism in the best possible form, adapted to the number of processors and other machine details. An accurate, minimum cost model that detects at least the asymptotic performance alterations of a given data-layout, scheduling or other mapping transformations is a key for this kind of techniques.

From the above discussion, we suggest that an ideal PPM should abstract the programmer from the number of processors that he is going to use, letting him only to show hierarchically the different levels of parallelism in the problem solution (from the coarsest to the finest). The PPM/PCM should include an automatic or interactive procedure to map this kind of programs to the restricted resources of a given architecture using: (1) ME, (2) an asymptotically accurate cost model supplied with the target machine model and parameters, (3) a proper scheduling technique to transform fine-grained parallelism to the proper granularity, eliminating unnecessary communication and leading to the proper parallel-slackness needed to obtain an efficient program.

We conclude that ME is not needed at the highest abstraction level of specification, but it is helpful to express some solutions to specific problems and to help the PCM implementation to take decisions about where and how to deal with contention problems that are not solved by typical underlying architectures.

2.2.3 Classification criteria for SAs

We can classify SAs according to the different properties (in expressiveness vs. analizability trade-off) they induce in a PPM/PCM or application. We propose criteria based on three orthogonal axis as shown in Fig. 2.3. The two first axis correspond to the two orthogonal types of synchronization (CS and ME synchronization). They are orthogonal in the sense that a PPM can support both or either of them independently. The third axis is based on a criterion that distinguish data-dependent from non-data-dependent synchronization structures. CS and ME are combined by the programmer to create the appropriate synchronization structures for a given application. Some applications will always create a given synchronization structure or combination. However, applications that are data-dependent may create processes and any type of synchronization (ME or CS) dynamically. Thus, it is possible that the exact synchronization structure created by an application will be not known until execution time. This third criterion becomes important to detect if synchronization structures may be analyzed and manipulated statically at compile time or only dynamically at run-time.

The relevant classes identified in each of the three axis are:

1. CS synchronization subtypes:

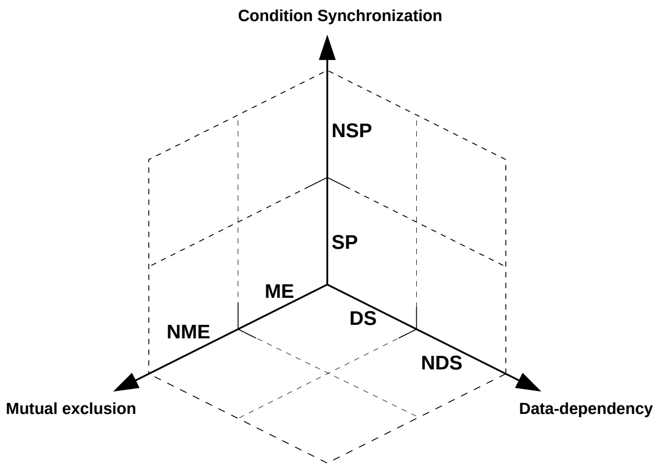


Figure 2.3: SA classification criteria

We propose only two complementary main categories of CS structures regarding the properties of the PPM that derivate from its class: (1) Hierarchical, SP, or Series-Parallel (also known as nested parallelism); (2) Non-hierarchical, NSP, or Non-Series-Parallel.

2. ME synchronization subtypes:

   We consider two classes: (1) PPMs not supporting mutual exclusion (NME), or applications which do not need it; and (2) another complementary class for PPMs supporting, or applications which use, ME.

3. Data-dependency subtypes:

   We distinguish between: (1) Non-Data-Dependent synchronization structures (NDS), and (2) Data-dependent synchronization structures (DS), created by a PPM which allows dynamic thread creation [174] or data-dependent synchronization structures (determining which and when processes communicate at run-time). Parallel algorithms may also be designed with non-data dependent structures or may use semantics that need data-dependent (dynamic) synchronization.

Thus, we propose eight SA classes, where some of them can be empty at some abstraction or modeling levels if no useful parallel computations (PPMs or applications) present such synchronization structures. Each class will be named by a triplet $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, where a will be the class of CS, b will indicate if ME can be exploited and c if data-dependent synchronizations are possible. In the following sections we will fully describe each axis subclass, presenting examples of PPMs and programs for each one.

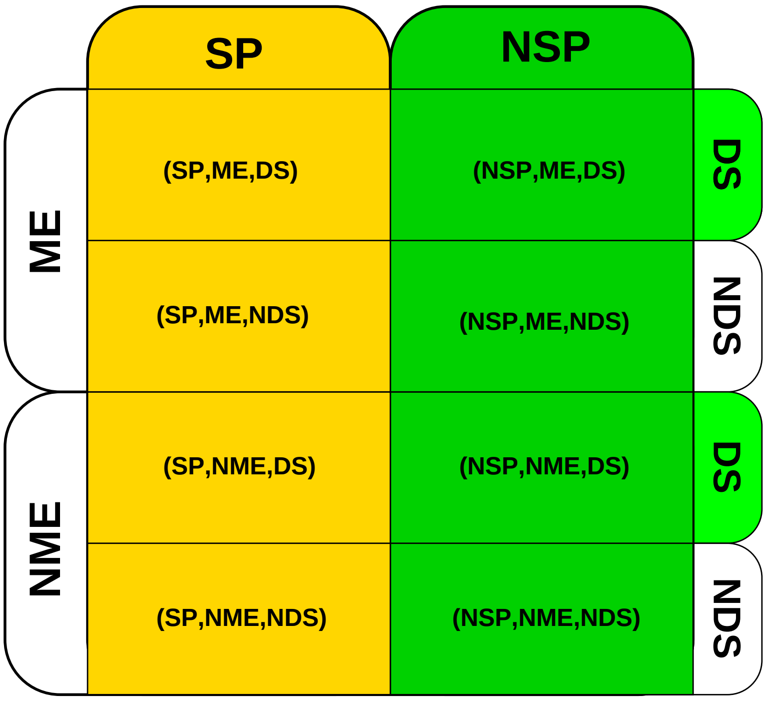


Figure 2.4: SA classification

We will conveniently represent the SA space in two dimensions as in Fig. 2.4. In this graphic representation, the less restrictive SA classes are in the top right corner, and the more restrictive are in the bottom left corner. Fig. 2.5 shows the idea of increasing restrictiveness from one class to another with small arrows, and the intuitive idea of general increasing restrictiveness from the top right corner to the bottom left corner with a big arrow.

2.2.4 Condition synchronization: CS classes

PPMs that do not support condition synchronization must base all solutions in ME. They cannot solve many concurrency problems that need *fairness*, should ensure *no-starvation* or should avoid *dead-lock* conditions. We consider this case a degenerated class of PPMs not fully useful for general parallel computation. Applications based mainly in ME for problem solving typically include some form of CS at least to create processes or threads and to wait for them to end before another stage begins, or the application finally ends (see e.g. section 2.5.5).

We will present now the two classes of CS structures with an example of a possible parallel programming language and a possible program for each class.

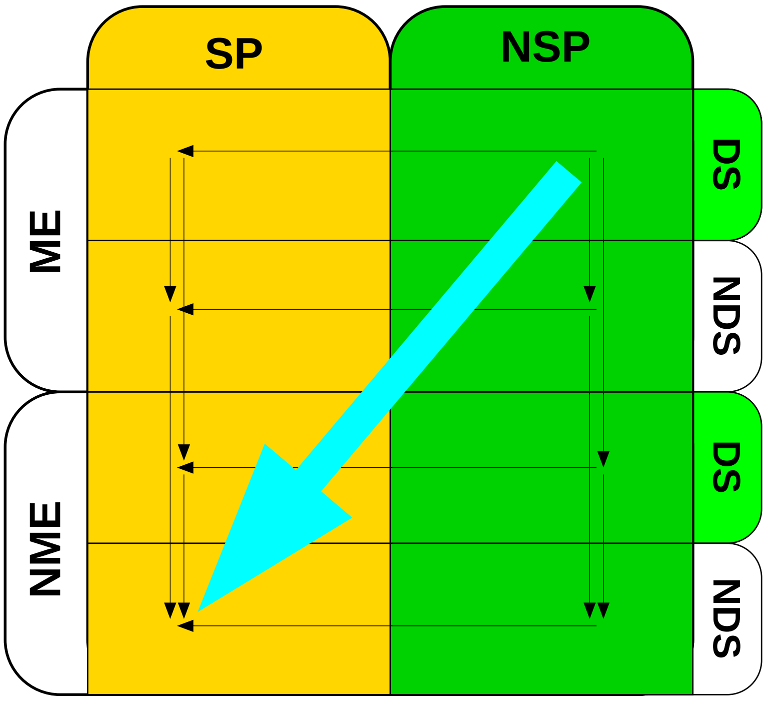


Figure 2.5: Restrictiveness increase of SA classes

A. SP (Series-Parallel)

This class contains the SAs which only allow CS structures which dependences can be represented by a series-parallel partial order set or series-parallel directed acyclic graph (see section 3.2.1). Series-parallel structures are generated by the so called *nested-parallelism*, nesting (or recursive applying) series and parallel compositions. They appear in PPMs with language primitives with the same semantics as the `cobegin`, `coend` constructors [9]. The end of the parallel section implicates a barrier synchronization before it proceeds. Next tasks are dependent on all of the tasks in the previous parallel section. Communications are implicit and only occur during the fork and join phases of a parallel section.

A formal definition of SP languages based on SP partial order sets and SP-algebras is presented in [126]. Automata theory can be extended for recognition of SP languages (see [127, 125]).

For example, consider a PPL where the only condition synchronization that can be expressed is implicit in `cobegin`, `coend` statements. Arbitrary computation blocks are identified by an integer in a `do <integer>` statement. An example of a possible code and its task graph representation is shown in Fig. 2.6. The numbers in the task graph represent the numbers of the computation blocks in the code. The synchronization structures that can be created are constructed by recursive application of spawning parallel sections, and serial composition. The use of global variables or data in tasks from different threads of parallel sections could compromise the program correctness, as the model considers the tasks in different subthreads completely non-dependent.

```
begin
    do 1;
    cobegin
        t1: do 2;
        t2: do 3; do 4;
            cobegin
            t1: do 5; do 6; do 7;
            t2: do 8;
            t3: do 9; do 10;
            coend
            do 11;
        t3: do 12; do 13; do 14;
    coend
    do 15;
end
```

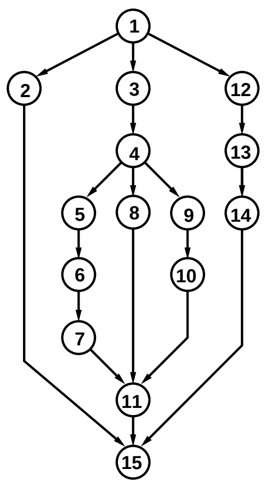


Figure 2.6: Example of series-parallel code and structure

Subclasses of SP class

In SP class of SAs we can distinguish two subclasses associated with well-known concepts related to the synchrony in PPMs, and widely used in the literature. Presented in order of decreasing synchronization restrictiveness, they are:

Lockstep: Each computation step is synchronized among all processing elements in the system. SIMD machines in Flynn's classification [64] works with these SA (see also PRAM model discussion in section 2.4.1). Typically, lockstep mechanism assume unit cost for the operations and no cost for the synchronization mechanism.

Bulk-synchronous: Each processor executes a series of local computational steps or tasks before all processors synchronize together in a full barrier. Communication or accesses to shared memory only occur such that the results are only available in the next phase, after the full synchronization. (See BSP, QSM and some PRAM derivate models in sections 2.4.2, 2.4.3 and 2.4.1 respectively). The recursive application of bulk-synchronizity creates SAs that are in the full SP class.

The relation between these subclasses is presented in Fig. 2.7. SP and NSP classes are complementary. Lockstep is a more restricted class than bulk-synchronization that is in turn a subclass of SP synchronizations.

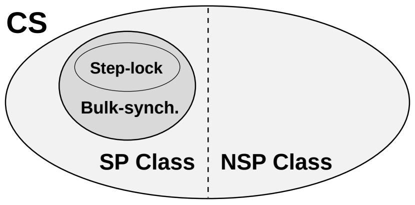


Figure 2.7: Classes of condition synchronization

B. NSP (Non-Series-Parallel)

This is the class of the SAs which allows static structures whose dependences can NOT be represented by an SP partial order set or SP directed acyclic graph. Any kind of dependences combination expressed with CS can be found in an application programmed in this kind of model.

PPMs in this class are also called *asynchronous*. Non-series-parallel models are related to the concept of synchronization by point to point message-passing or mechanisms as signal, wait primitives. Consider a toy PPL where arbitrary

```
(1)   do (> 1 > a,b )
(2)   do (> 2 > c )
(3)   do (b > 3 > e,f )
(4)   do (a > 4 > d )
(5)   do (e > 5 > g )
(6)   do (c,f > 6 > h,i,j )
(7)   do (d,g > 7 > k,l )
(8)   do (i > 8 > m,n,o )
(9)   do (j > 9 > p,q )
(10)  do (n,p > 10 > r,s )
(11)  do (o,q > 11 > t )
(12)  do (l,r,s > 12 > u )
(13)  do (s,t > 13 > v )
(14)  do (k,u > 14 > )
(15)  do (m,v > 15 > )
```

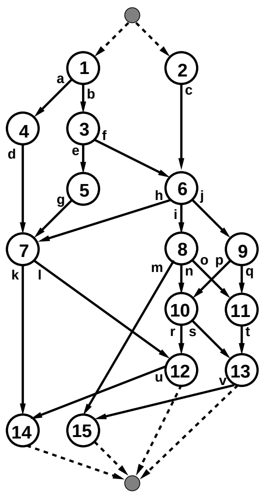


Figure 2.8: Example of non series-parallel code and structure

computation blocks are identified by an integer number, and they are executed provided that a collection of preconditions, identified by a name, are true. At the end of the computation a collection of postconditions can be issued. The syntax used will be the statement do (preconditions_list $>$ taskNumber $>$ postconditions_list). Any task with no precondition will be executed (in parallel) at the beginning of the computation. In Fig. 2.8 we show an example of a program which generates a complex non-SP graph. Any kind of synchronization structure or generic graph can be generated with such a language. (For this example toy language it is possible to create computations that never end, due to infinite cycles or conditions impossible to be satisfied).

2.2.5 Mutual exclusion: ME classes

We distinguish two classes in the ME axis: PPMs that do, and PPMs that do not provide ME mechanisms. We include a discussion about the different natures and ME mechanisms in the description of the related class. Application definitions also may or may not use ME semantics. In section 2.2.6 we discuss the problems of simulating ME semantics with CS.

A. NME (No mutual exclusion)

Models and applications in this class do not support or need ME mechanisms. In previous sections we have shown examples of PPLs and programs which do not use ME (see Fig. 2.6 and Fig. 2.8).

B. ME (Mutual exclusion)

In this section we discuss the different mechanisms that support ME. We will use as example a simple problem where n threads need to access a global variable (accessible in shared-memory or through communication mechanisms across threads) to use it as a counter. ME must be used to avoid race conditions. The typical mechanisms are:

1. Shared-Variable paradigm with mutual exclusion primitives:

   Some PPMs provide the programmer with mechanisms or primitives that have implicit ME semantics. In this case the programmer can directly specify which tasks cannot be executed in parallel (simultaneously), without specifying any implicit order. Any one can be executed before the others.

   The exact mechanisms can be of any nature: Atomic operations on variables, atomic transactions, critical sections specification, monitors, ... The main advantage of direct ME primitives is that the compiler can easily detect and reason about the effects of the unordered synchronization in the program performance. An approximation technique to the cost modeling of ME is given by Van Gemund in [70].

```
a=0
!$OMP_PARALLEL, shared(a)
    myId = OMP_GET_THREAD_NUM()
    !$OMP_CRITICAL
    a=a+1
    WRITE(*,*) "Thread ",myId," scores ",a
    !$OMP_END_CRITICAL
    WRITE(*,*) "Ending thread ",myId
!$OMP_END_PARALLEL
WRITE(*,*) "End of computation"
```

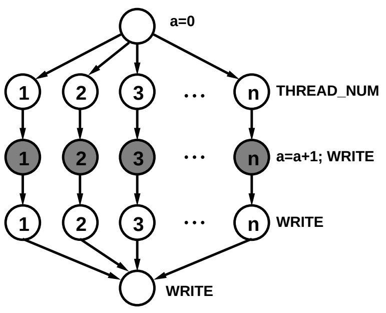


Figure 2.9: Example of code and structure with ME primitives

Consider for example OpenMP [149]. It provides a parallel section pragma OMP_PARALLEL and another pragma called OMP_CRITICAL to specify a part of code that is a critical section. Critical sections of code are mutually exclusive for all the threads in the same parallel section. The simplified code in Fig. 2.9 shows an example using OpenMP in FORTRAN language. In the associated task graph we cannot use normal oriented edges to represent this dependence, as it does not induce any order in the tasks. We use shaded nodes to represent this dynamic dependence. The shaded nodes will be executed sequentially but in no specific order. The number in a node represents the number of the thread executing the task. The screen results of this code execution depend on the order in which the threads get access to the critical section, but they will be consistent as no race condition in the $a = a + 1$ statement can be produced.

2. Message-Passing paradigm with programmed ME:

   Other models do not provide primitives with ME semantics, but they have a contention mechanism that can be use to manually program mutual exclusion.

   Consider for example a SPMD parallel language which begins a parallel section with a parallel() statement, a facility to get the own thread number get_id(), and has a message-passing interface with send(p,i), recv(p,i) operations, where p is the number of processor to send to or to receive from, and i is an integer. Suppose we allow the receive operation to get a message from any processor, the first that arrives at the in-port. For our example language, if $p = -1$, then the recv(p,i) operation will return in p the number of the processor from which the next message comes. We are allowing a kind of contention between arriving messages in the in-port of the receiving processor. We consider a case in which if several messages are sent simultaneously the order of arriving cannot be predicted. In the code in Fig. 2.10 we show an example of using this feature to produce mutual exclusion, using one thread as a (monitor like) dynamic synchronization server. Again, the number in a node represents the number of the thread executing the associated task. The results will be similar to those of the previous example for ME primitives.

```
parallel(n+1) {
    myId = get_id();
    if (myId==0) {
        a=0;
        for (i=1; i<=n; i++) {
            p=-1; /* From any */
            recv(p,foo);
            a=a+1;
            send(p,a);
        }
    }
    else {
        send(0,foo);
        recv(0,result);
        printf("%d reads %d",myId,result);
    }
}
printf("End of computation");
```

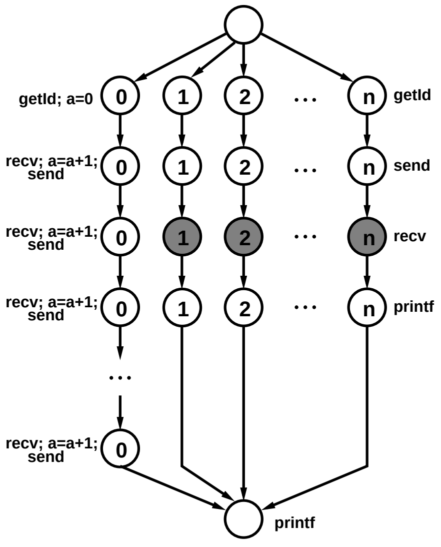


Figure 2.10: Example of code and structure with programmed ME

Although it is still possible to produce similar results as using ME primitives, the programs get more complicated, the programmer must face semi-scheduling issues, and the global effect of the mutual exclusion is hidden to the compiler. Typically, in these programming models, the analysis of the contention must be done at a very low level, where the original semantics of the mutual exclusion are lost, complicating the overall cost analysis with new low-level parameters.

2.2.6 Mutual exclusion vs. condition synchronization

Some PPMs do not include any mechanism for mutual exclusion (NME). When such a model needs to deal with a problem like the one proposed as example in section 2.2.5, the only possible solution is to use condition synchronization between the tasks that cannot be executed in parallel, creating a specific order, that may be not the optimum schedule.

Consider a SPMD language extension of C, with an explicit parallel region construct, with a facility to identify the number of the current thread get_id() and with semaphore-like operations: wait(c) that waits until c condition is signaled, signal(c) that signals the condition c. Conditions will be identified by an integer number. Thus, the code in Fig. 2.11 shows how to use condition synchronization to avoid race conditions in the access to the shared variable a.

```
a=0;
parallel(n) {
    myId = get_id();
    if (myId==1) { /* thread 1 */
        a=a+1;
        printf("%d reads %d",myId,a);
        signal(2);
        }
    else if (myId < n) {
        wait(myId);
        a=a+1;
        printf("%d reads %d",myId,a);
        signal(myId+1);
        }
    else { /* thread n */
        wait(myId);
        a=a+1;
        printf("%d reads %d",myId,a);
        }
    }
printf("End of computation");
```

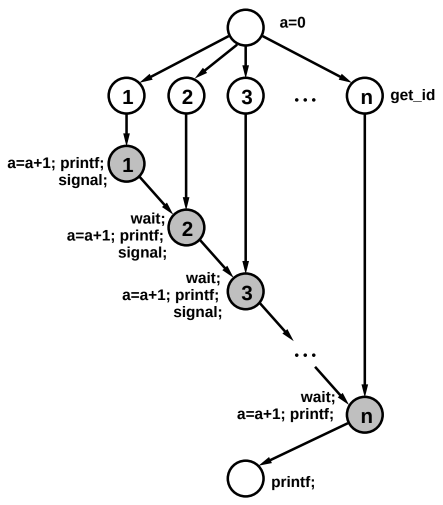


Figure 2.11: Example of none ME synchronization code and structure

However, condition synchronization creates an ordering over-specification not really coming from the original problem. In cases of not perfectly balanced situations, where the contending threads may arrive at the critical section in random order, this over-specification could delay threads prepared for execution until the previous threads in this false order arrives and finish the critical task. Fortunately, not many parallel problems present this kind of unbalanced behavior.

2.2.7 Data-Dependency: DS, NDS classes

This classification axis is related to the creation, from the same program, of potentially different synchronization structures at run-time (data-dependent). We distinguish only two classes.

A. NDS(Non-Data-Dependent synchronization structures)

Many applications create the same synchronization structure independently of the input data (no thread creation or communication target is decided as a function of the data values). Although not common, PPMs may support only this kind of data-independent structures. In this case, synchronization mechanisms are provided with explicit information about which processes or threads communicate at compile-time. A PPM that is restricted to only non-data-dependent synchronizations must have a predetermined number of named processes running. The name of the process to which a communication or synchronization is issued must not be able to be determined at run-time.

```
initialize( M(1:1000,1:1000) )
numIterations=3
do i=1,numIterations
    dopar
        cellAutom(M(1:251,:), M(1:250,:))
        cellAutom(M(250:501,:), M(251:500,:))
        cellAutom(M(500:751,:), M(501:750,:))
        cellAutom(M(750:1000,:), M(751:1000,:))
    end-dopar
end-do
write(M)
```

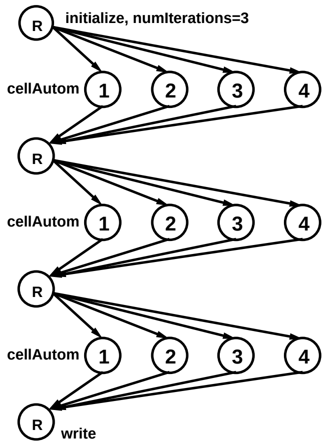


Figure 2.12: Example of static synchronization code and structure

Let us consider an example PPL, where the parallelism can be only expressed by a dopar, end-dopar construction that contains no code, but a maximum of p function calls with one input and one output parameter. Each function is executed in an independent process that receives the input parameter from the root process and communicates the output parameter back to the root process. The semantics of the language do not allow conflicts by syntactically forcing that the variables which receive the output parameters int the root process must be in non-overlapping memory cells. The functions inside a parallel construction must not contain other parallel construction. In this model, the synchronization structure is completely non-data-dependent if the dopar construction may not be inside a conditional statement. Hence, no run-time decisions may affect the parallelism or communication structure. An example of a static cellular automata like program in such a PPL is shown in Fig. 2.12. The input parameter of each function include the frontier lines of the matrix, while the output parameters do receive only the computed part, with non-overlapping lines.

B. DS(Data-Dependent synchronization structures)

Almost all PPMs allow an implicit or explicit form to create data-dependent synchronization structures. Typical cases of these synchronization mechanisms are communication/synchronization primitives inside conditional statements, communication channel names selected at run-time by a computed value, data-dependent asynchronous communications, wildcards for message receiving primitives in message-passing, dynamic creation of processes inside conditional or loop statements, etc.

```
MPI_INIT(err)
MPI_COMM_SIZE(MPI_COMM_WORLD, numP, err)
MPI_COMM_RANK(MPI_COMM_WORLD, myId, err)
IF (myId==0) THEN
    read(*,*) s
END-IF
MPI_BCAST(s,1,MPI_INTEGER,0,MPI_COMM_WORLD,err)
DO i=1,s
    neig = MOD(myId+i,numP)
    CALL MPI_SEND(myId,1,MPI_INTEGER,neig,0,MPI_COMM_WORLD,err)
END-DO
DO i=1,s
    neig = MOD(myId+numP-i,numP)
    CALL MPI_RECV(lec,1,MPI_INTEGER,neig,0,MPI_COMM_WORLD,status,err)
    write(*,*) myId, "receive: ", lec
END-DO
MPI_FINALIZE(err)
```

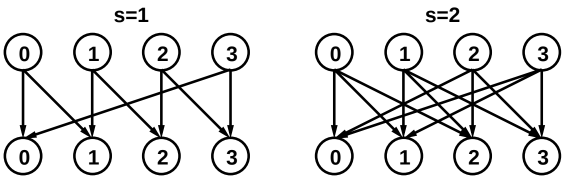


Figure 2.13: Example of dynamic synchronization code and structure

In Fig. 2.13 we present an example of a FORTRAN-like MPI based code that produce different synchronization structures depending on a run-time value. The value is read from an input device and determines the number of communications, and the processes to which they are sent. Two examples of the generated graph are shown for values $s = 1, s = 2$ and executions with 4 processors.

2.3 Execution-level models

In the following sections we will use the SA classification to show that benefits and disadvantages found at different modeling levels are strongly related to the concepts used for our SA classification criteria. Our trip along the parallel programming abstractions begins in the lower level, where the unknown oceans of parallel program execution are shaking the dangerous cliffs of machine models. The rocks fight with the fierce waters, trying to resist in the middle of the moaning winds to form an established coast line. People working in parallel architectures try to rule this broken seaside, in constant change, applying all new affordable technologies. In their efforts, some machine models have been acknowledged and are being used as abstractions for development of higher level programming tools.

The machine models we review in this section are more or less established ideas. Sometimes they are thought as equivalents of Von Neumann architecture for parallel computing, but many of the times they are considered little abstractions of current technology trends in the concurrency and high performance race. Nevertheless, there exist a convergence of parallel machine models at hardware and organization levels [51].

2.3.1 SA class of machine models

Most machine models are designed to provide full capacity of communication and synchronization among processes. Thus, they are mainly in the SA class that presents no restriction (NSP,ME,DS). The two main trends of parallel architectures have been *shared memory-address space* and *distributed memory-address space* or *message-passing* models. We discuss also the *data-flow* machine model, because it is a different and interesting graph-based approach to generic parallel computing. There are other non-generic models that are not considered in our study, e.g. systolic arrays (simple lock step application oriented circuits), vector machines, and data-parallel machines. Their architectures are specifically designed to obtain better performance for specific types of computations. Thus, their SAs are highly dependent on them. The following descriptions are mainly based on [51].

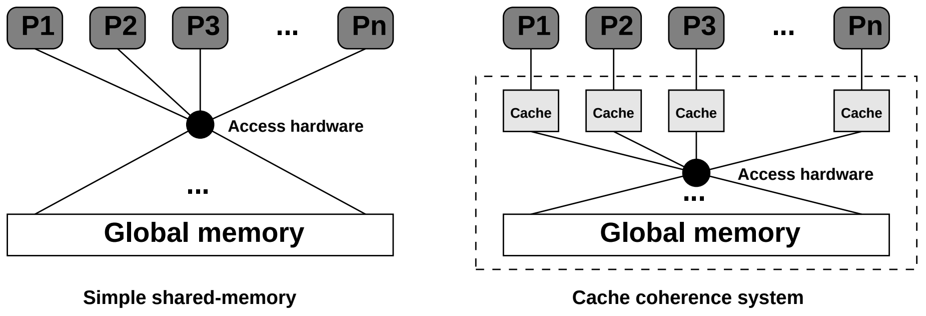


Figure 2.14: Shared address space machine models

Shared address space

Shared address space systems have hardware support for global access to any memory cell from any processor. The latency of memory access can be uniform (UMA) or non-uniform (NUMA) depending on the physical configuration of the memory across the machine, the presence of caches with a coherence system, and the processor to memory access hardware (see a block diagram of two typical configurations in Fig. 2.14). But it is anyway transparent to the upper levels. This kind of machines provide different mechanisms to prevent race conditions when accessing memory cells concurrently. However, the programmer is responsible for using the synchronization and contention mechanisms provided by the architecture (operative system or hardware) to create programs with fixed semantics and no stochastic behavior. ME is then programmed with explicit primitives that implement lock systems. CS is created through similar primitives also hardwired in the operative system (e.g. semaphores) or the hardware itself (e.g. CrayT3E provide even a hardware barrier mechanism, and cache coherence hardware may be exploited in ccNUMA machines for the same purpose [102]). As they are based on some kind of *flag set, flag test* mechanism, the CS structures created by processes are not restricted.

Message-Passing

Message-passing (distributed address space) machines are based on a model where processors only have access to a local memory, and communicate with other processors to obtain remote data by exchanging messages. There exist many different message communication mechanism, all of them abstracted as an interconnecting network from the machine model point of view (see a block diagram of these machine models with two example configurations of the abstract node elements in Fig. 2.15). Messages are used to create CS in a natural way (when the precondition is activated, a message is sent to all the processes waiting for it and the reception of the message fires the action). Messages are in transit through the communication network for an unknown and typically unpredictable time (usually depends on network traffic). Thus, the order of several messages sent from different processors at different times cannot be predicted. The programmer may program ME using messages. The processes that want to execute a mutual exclusive task (critical region) must send a *request* message to a resource server process and receive a *confirmation* message from it before they proceed. After the execution of the critical region, the process send an *ending* message to the server to indicate that it can send a confirmation to other requesting processes. Thus, in this model ME must be manually programmed.

As presented in [51], there exist a convergence in these two main trends of parallel machine models. Traditional message passing operations are supported

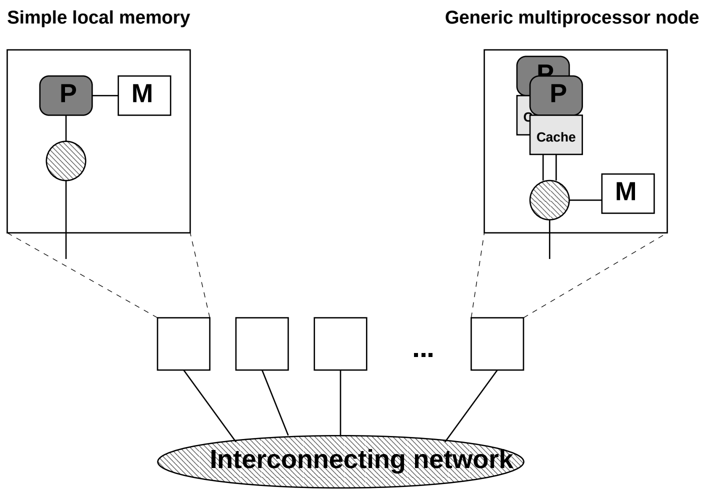


Figure 2.15: Message-passing machine models

by shared-memory machines using hidden shared buffer storage with a proper API. On the other hand, over a message-passing system it is possible to build a more abstract layer where a global address space hides the necessary messages to exchange data. Thus, in upper abstraction levels, even if different PPMs seem to be more oriented to a specific machine model, all of them can be implemented in both types of architectures.

Data-Flow machines

These machines architecture are based on a hihgly abstract execution model. The programs are specified as static task graphs. A node is a basic operation to be executed when all *precondition* (input parameters) are available. After execution, a node throws its *post-conditions* to successor nodes. The processors are based on a matching mechanism that identifies ready to run graph nodes (those which inputs are already computed) and spawns new threads to execute them. The execution graph has the same topology as the input graph, which structure is not restricted. Thus, any kind of NSP CS is possible, although it must be static. In a more dynamic version, the nodes can be function invocations with *context* information. Dynamic synchronization structures are possible. As information generated as node output may be used or modified by different successor nodes, lock mechanisms to create ME are provided to access memory elements (by hardware) or entire data structures (by operative system). Thus, ME is implicit in the low level data access system.

2.3.2 Conclusions about execution models

All machine models are in the less restricted SA class (NSP,ME,DS); see Fig. 2.16. Parallel machine developers try to satisfy all possible consumer requirements. Hence, most machine models proposed have the capacity to create any kind of CS structure. Moreover, ME is needed at low levels for shared resource control. It is a basic feature for distributed and parallel operative systems. They should also have mechanisms to create or destroy processes and threads to attend new user jobs and system requests. These elements, that appear and disappear at hand, may communicate or synchronize among them. Although an specific installation of the operative system may limit this capacity, the parallel machine models are fully dynamic and support data-dependent synchronization structures.

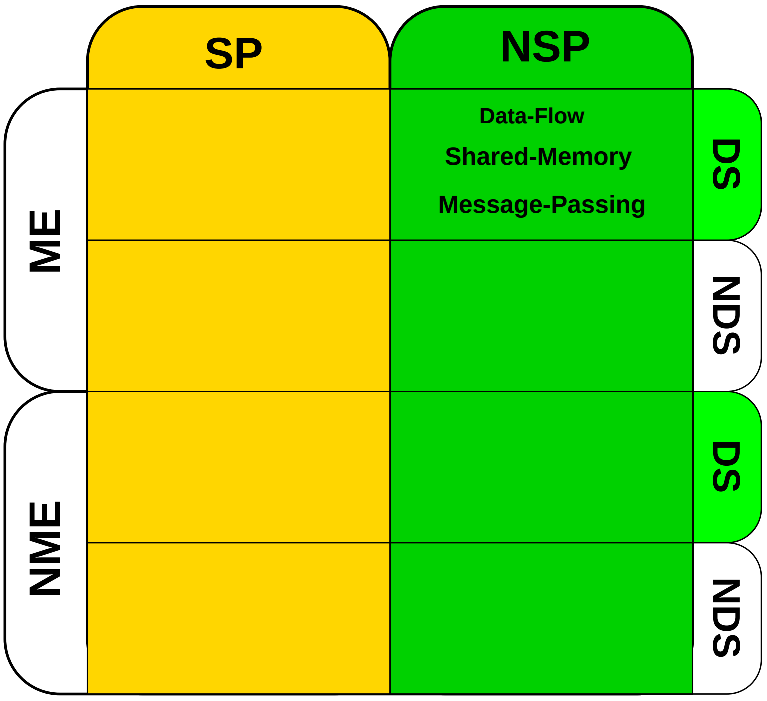


Figure 2.16: SA classification of machine models

In generic environments, such as NOWs or GRID computing, it is common to have only software mechanisms to synchronize. Especially in these environments with mixed architectures and high latencies that inefficiently increase the synchronization time with the number of processors, structured and hierarchical synchronization highly increases performance. Hierarchically splitting the computation in subsets of processors improves locality, and maps well to big heterogeneous or hierarchical clusters (see e.g. [119, 187]). Thus, more restricted synchronization architectures (specifically in SP class) will be found in higher abstraction levels, to improve software development on these generic execution models.

2.4 Bridging models and cost models

Walking up from the execution level coasts, we will travel through the wide lands of cost evaluation and bridging models (PCM/PPM). We will find pleasant slants of greenery where new proposals flourish, but most of the time we will cross vast fields which old well-known PCMs have ploughed long ago, and where the crops so many of experience are now hanging on their heads.

We review several models frequently found in the literature. All of them propose a PCM based on an abstract parallel machine, give a performance cost model (at least for asymptotical complexity measures) and prescribe a SA for the PPM. Some of them are more focused on the solution design point of view, but most of them are introduced as *bridging models*, proposing a trade-off between programmability and efficient mapping for any machine. We examine here the most popular ones, focusing on the features relevant to our study, to show how SA is highly related to the analyzability properties of a model. (For a more complete survey of parallel computation models see e.g. [129, 35, 4]).

2.4.1 Class (SP,NME,NDS): PRAM

In this class we find an important family of PCMs with a common origin. The PRAM parallel computing model [65] has been used for parallel complexity measurement during more than two decades. In PRAM, a parallel computer architecture is highly abstracted, leading to a very simplistic model for easy programming. Although in many references (see e.g. [73, 156]) it has been presented and used as the equivalent of a data-parallel programming model, based on the *SIMD* (Single Instruction, Multiple Data-flow) machine model of Flynn's classification [64], the PRAM model has indeed more expresive power and it is a full *MIMD* model.

Description

A PRAM machine [65] consist of a control unit that *synchronously* activates the execution of *one machine level instruction* on an *unbounded number of processors* that, apart from their private memory, work with an *unbounded global memory space of uniform time access* (see Fig. 2.17). When the execution begins the same program is loaded in each processor (SPMD model). The processor P_0 is the only processor activated when the computation begins, but the instruction set includes a *fork* operation to activate other processors which may evolve in different ways as they have their own program counter, acumulator register and an unbounded number of private memory cells. Note the conceptual similarity with the shared-memory machine model block diagram in Fig. 2.14.

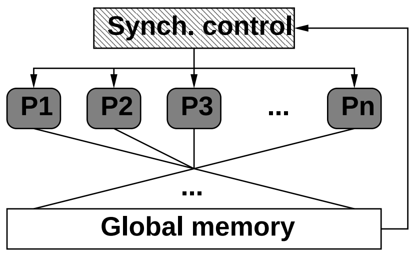


Figure 2.17: PRAM computing model

The cost of a PRAM algorithm is the product of the parallel time complexity by the number of processors used. Time complexity is easily measured as the processors operate synchronously and global memory accesses have uniform latency included in the processors step.

Derivates of the basic PRAM model exist to cover the problem that appears when more than one processor issues simultaneous accesses to a cell in the global memory (see e.g. [156]). They affect the programmer in the techniques available for algorithm design, but the stronger the model (the more expressive power) the further from easy implementation of the model in a real machine. The PRAM model can be considered:

EREW (Exclusive read, exclusive write): Two processors are not allowed to read or write at the same memory cell simultaneously.

CREW (Concurrent read, exclusive write): Only simultaneous reads are allowed in the same cell, but only one processor can write. This is the default PRAM model.

CRCW (Concurrent read, concurrent write): Reads and writes to same cell are possible in the same step. A policy for handling concurrent writes must be specified, leading to more subclassifications of the model (see e.g. [156]).

PRAM models can be simulated, sometimes efficiently, in other variants of PRAM or in other parallel computing models (see e.g. [97, 128]).

Synchronization architecture

The basic PRAM model presents a extremely restricted SA. The CS is restricted to a synchronous advance step by step, or *lockstep* (a subclass of SP). All tasks in one step *depend* on all tasks of the previous step. Whether all processors execute the same instruction or they execute different ones does not affect the synchronization structure. The most important simplification for the cost model is the assumption of similar latency in memory accesses and computation steps, forced by the synchronization control.

We consider PRAM to be in the NDS SA subclass because even if data-dependent applications may be programmed, the synchronization structure is transformed to a static lockstep structure. Although processors may activate other processors at any moment, the PRAM algorithms are typically designed in two phases [156]: In the first phase a sufficient number of processors are activated, and then, in the second phase, all processors activated are used to execute the program with no new activations. In this case, the number of processors used for a given computation is fixed. Let us analyze now the allowed synchronization patterns among them. The number or index of the processor and the values of memory cells may be used in conditional statements. Thus, in principle, the exact instructions and memory locations accessed in one computation phase can be completely data-dependent. From this point of view, the communication structure can be also data-dependent. However, from a more abstract point of view, the lockstep mechanism transforms any communication structure in a full all-to-all synchronization structure. On one hand, programmer do face an static programming model, as the synchronization structure is synchronized and memory latencies or communication problems are transparent. On the other hand, data-dependent applications may be programmed, being the lockstep mechanism the responsible to deal with the dynamic behavior during implementation.

The objective of PRAM is to simplify the cost model assuming unit cost for computation step and communication. One can see PRAM as synchronized only because of equal latency on the operations. This simplifies the algorithm design, but the implementation should keep the communication structure expressed in the algorithm, in the presence of real latencies and even with asynchronous execution in each processor. In this case, the implementation of a PRAM algorithm can express regular but not fully-synchronized patterns between each layer of computation (a SA in NSP and DS class). As it is shown in the discussion below, these differences between specification (using a cost model in a highly restricted SA) and implementation (in unrestricted SA), is one of the reasons why PRAM fails to provide good mapping features.

The basic PRAM model lacks ME mechanisms, as they are not needed in a lockstep SA. The only shared resources are the memory cells. The EREW model do not allow writing algorithms that need contention control, while the CREW model assumes the possibility of simultaneous reading but no writing contention is allowed. In the CRCW model, a contention policy for conflicts prevents the need of ME. However, if an arbitrary non-deterministic policy is assumed, ME may be explicitly programmed. One processor may be used as resource (critical region) server, using the non-ordered contention in a memory cell to communicate the requests. Thus, for non-deterministic contention policies, CRCW is in class (SP,ME,DS).

Discussion

The simplicity of the model allows immediate cost measures. The parallel time complexity is in the order of the number of instructions executed (as any operation is synchronized among all processors), and no more parameters are considered. However, the simplicity of the model makes it to ignore important details of real parallel and distributed programming.

First, global memory access in uniform time is not portable. It is not easily simulated in non-uniform memory access (NUMA) machine, and the model does not consider the cost of full communication in a distributed memory architecture. Thus, the model does not discourage the design of algorithms with a very fine grain of parallelism. Communication patterns can produce bottlenecks that completely neglects performance improvement and scalability. The time needed for contention solving in real concurrent accesses to memory cells is also disregarded.

Second, the number of processors is unbounded. It is considered that a fixed number of processors can simulate a set of PRAM processors, but the implementation of the synchronization system, a load balance mechanism when PRAM processors are dynamically switched on and off, and the cost of the simulation with concurrent memory accesses are difficult issues and can completely modify the complexity bounds of the original algorithm.

The conclusion is that PRAM model is adequate for basic theoretic complexity measurement, or gross classification of algorithms. However, it is so unconcerned about real machine details that the mapping problem of PRAM algorithms is far from direct, and many details must be still considered by the programmer to keep the original features of the algorithm for a specific machine. However, for its simplicity, and for assuming unit resource costs, it encourages the algorithm designer to expose all possible parallelism in the problem (even if this fine-grained parallelism will have a non-efficient or even a non-affordable cost). Thus, it surely will survive as an interesting tool for theoretical purposes.

PRAM extensions

Many extensions of the original PRAM model have been proposed to solve the model shortcomings. They typically try to tackle one of the main important features not contemplated in basic PRAM, although some of them try several at the same time. Some are still too simplistic and they do not usually map well in real architectures. Others lead to much more complicated or even non-practical cost models. In general they try to preserve simplicity, by assuming restricted

SA. Evolution to real bridging models can be noticed in some of them. Consider these few examples (see [129, 35] for a detailed survey of more alternatives):

Contention problems: An extended family of PRAM models known as the QRQW-PRAM [75] (Queue read, queue write PRAM) deals with the contention problem in memory accesses. This model is better suitable for architectures with pipelining contention rules in cells, and sufficient processors-to-memory bandwidth. Efficient implementations in other architectures are not supported. This model support programmable ME using the contention queues, moving the SA to ME class (still in lockstep subclass of SP CS).

Asynchrony: It is another important issue in PRAM model extensions. Some examples of *partial* asynchrony are in the *Asynchronous PRAM* [74] and the APRAM [46] models. In these models different processors may execute at different time rates, skipping the lockstep mechanism. Nevertheless, explicit synchronization is needed to keep consistency in write/read operations. Thus, these models propose global or partial synchronization mechanisms. *Communication* through write/read operations between synchronization points is limited to eliminate dependences (e.g. no read after a write in the same global memory cell before a synchronization point). There exists several variants:

APRAM: Synchronization occurs in fixed rounds. SA moves to *bulk-synchronous* SP subclass.

Phase Asynchronous PRAM: Full synchronization is explicitly used by the programmer for consistency in read/write operations: Bulk-synchronous SA.

Subset Asynchronous PRAM: The programmer can use full synchronization in hierarchical subsets of processors. SA moves from lockstep and bulk synchronous subclasses. SP synchronization structures are allowed. As the subsets of processors may be created dynamically by data-dependences, the synchronization structures are change to DS class. The SA is in (SP,NME,DS).

All these models still keep an affordable cost model due to the SP-restricted CS structures.

Another model called Asynchronous QRQW-PRAM [77] combines contention in cells and real asynchrony, where dependences through accesses to global memory can appear in any form. Thus, SA moves to (NSP,ME,DS) class. However, to avoid the problems of complexity, reducibility and analysis in the cost model, derived from unstructured CS, it forces the programmer to construct the program in a way that it assures correctness under the worst case assumption on the finite delays incurred by the processors in queuing global memory accesses. The cost model uses an optimistic synchronous assumption. Thus, the complexity introduced by the NSP SA, is moved not to the cost model (that works properly for bulk-synchronous structures), but to the programmer decisions. Many PRAM algorithms must be reconsidered and reprogrammed to get profit of this model, and to assure correctness if the simplified cost model is to be used.

Communication latency: Several variants consider different latency values for accessing local or global memory. Some well-known examples are the LPRAM and BPRAM models.

LPRAM model [3] distinguish only two latency times: One for accessing local memory (unit time) and one for accessing global memory cells (a new latency parameter). It is suggested that LPRAM algorithms should restrict their behavior to perform two different kind of steps. Communication steps (where the accesses to global memory has a high fixed cost), or computation steps (where processors work in local memory in unit time). Thus, the cost model includes two types of steps with different costs, but the SA does not change and the analyzability is not affected.

The Message-Passing Block PRAM (BPRAM) [2] includes a startup cost for a message (or access to a global memory block) and a constant cost for any word in the message (pipelined read/write operations). Thus, it rewards the sent of long messages, and encourages the design of algorithms that exploit data locality to form cohesive blocks that can be moved fast. A processor can send and receive at most one message in a step. This model does not greatly modify the SA. As long as different block accesses can have different costs, the lockstep is inherently substituted by a bulk-synchronous activity. The cost model of a step is a little more complex due to new parameters for more accurate predictions. But the overall cost model simplicity is similar because of the still SP-restricted SA class.

We conclude that many extensions of PRAM model try to cover features ignored in original PRAM to jump over the implementation gap. Some try to improve accuracy by adding new parameters and a little complexity to the low level details of the cost model, but keeping a restricted SA to make the overall solution simple and easy to handle. Others move to unstructured SAs, leading to cost models that become too complicated. Some of them are so far from original PRAM model that no algorithm developing techniques and practice have been yet exerted. In general we notice how newly introduced features that seriously modify the SA lead to important changes in the cost model or mapping properties of the model.

2.4.2 Class (SP,ME,NDS): BSP

The *Bulk Synchronous Parallelism* model [185] was introduced as a more realistic bridging model for a parallel machine. BSP and variants have been studied for more than a decade and its introduction has produced a lot of expectation and interest. However, its acceptance is not complete due to its restricted parallel expressive power. We focus on several key features of BSP and especially in extended BSP models that support nested parallelism, as they provided the nearest framework to a pure SP parallel programming model.

Description

A simplified model of a parallel computer, called the *bulk-synchronous parallel computer* (BSPC) consists of: (1) A fixed number p of processors with local memory; (2) an interconnecting network with limited bandwidth and simple bounded latency parameters; (3) a fixed cost barrier synchronization system.

The BSP computer works in *supersteps*. In each superstep every processor works independently with its local memory and data. During the computation phase every processor sends or receives at most h messages of little size (typically one word) to other processors (if $h = p$ every processor communicates with all the others). This is called an *h-relation* (see e.g. [173]). Data received from other processors are not available until next superstep. After the computation/communication phase, a full barrier synchronization is issued. Every processor begins the next superstep at the same time (the full synchronization can be inherent to the communication phase when $h = p$).

Two main interpretations of how BSP superstep works and its cost model exist (see Fig. 2.18). The main premise for the model is a *consistency statement* that assures that data coming from other processors during superstep s are not used for computing before the beginning of superstep $s + 1$. Thus, communications could be issued during the computation phase at any moment, provided that transfered data arrived during the current superstep are not used in the target processor before the beginning of the next superstep.

Interpretation 1: Completely horizontal model. The messages are delayed until the end of the computation phase (all processors end their computation for this superstep), and sent during a communication phase. See for example [114, 173].

Interpretation 2: Overlapping model. The messages are sent during the computation phase, overlapped with computation. Examples of this interpretation can be found in [76, 133].

The cost parameters of the model are:

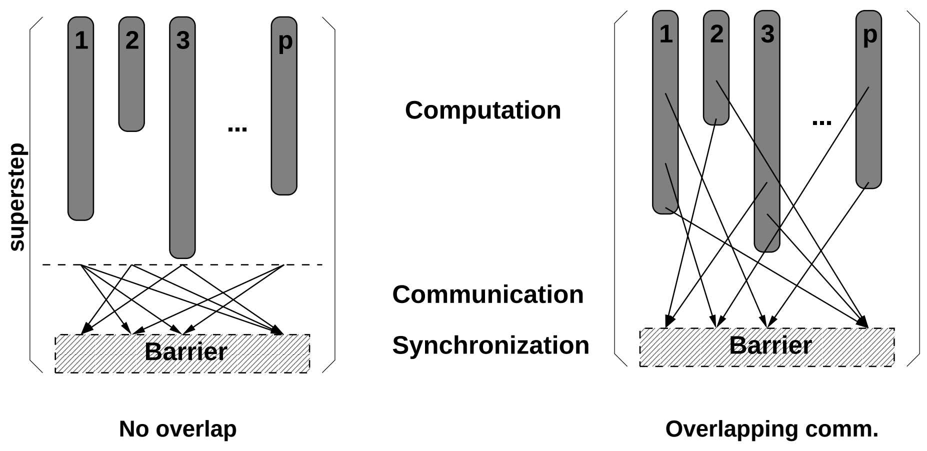


Figure 2.18: BSP computing model

p: Number of processor elements.

g: The cost per communication (the basic throughput of the communication system while in continuous use).

Some simplifying assumptions are made here. The cost of establishing initial communication is generally ignored, as long as the minimum number of communications per processor in a superstep is considered to be enough to neglect this innitial latency, in comparison with the total communication cost. Global communication structure can also be exploited to hide it.

Another simplification assumed by the cost model is that messages are small (in the order of the processor word), in order to always have similar sending latency time.

L: Periodicity. The computing unit for a superstep.

As originally proposed by Valiant [185], each L time units the system checks if every processor has finished its superstep activity. Then, communications are finished and a new superstep begins after a full synchronization. During a superstep each processor can do L computation steps, and can send/receive at most $\lfloor L/g \rfloor$ messages.

L parameter has other meanings depending on the interpretation of the model used. See following discussion about the cost model.

The cost measurement is easy as any computation works in supersteps of $cL + gh$ time complexity (c is the number of periods of L time used by processors before they end the computation phase).

Authors using the horizontal interpretation typically consider another parameter l for the cost of the barrier synchronization, and they completely split computation and communication costs. In this case, w_i represents the work/computation time of each processor during the superstep and h_i the total number of messages sent by the processor i. The cost model of a superstep is:

$$T = \max_{i=1\ldots p} w_i + \max_{i=1\ldots p} gh_i + l$$

Other authors using this interpretation consider L to be the minimum cost of a superstep. Thus, L represents the time for synchronization and activation of next superstep and it substitutes l in the formulae.

However, in the second interpretation, computation and communication overlap. Typically L is considered a *minimum latency* parameter that represents the minimum time length of a superstep imposed by the hardware. Let be $W = \max_{i=1\ldots p} w_i$ and $H = \max_{i=1\ldots p} h_i$. Thus, the superstep cost model is:

$$T = \max(W, gH) + l$$

Or, in other versions:

$$T = \max(W, gH, L)$$

The parameters L, l, g are empirically measured for a given architecture and a given number of processors p. The cost model can be used to test how an algorithm maps to a range of values for the parameters combination (the BSP space). Thus, if the cost model shows to be accurate enough, the programmer can easily predict which algorithm is going to perform best for an specific machine.

In [49] we read that being the L parameter (the duration of a superstep or periodicity) calculated as a function of h it must be considered for the worst possible h. In this case, the cardinality of interprocess communication can be different in different supersteps leading to performance losses in some steps. The *Extended BSP* (E-BSP) model [113] includes an extended cost model based on more complex and variable *routing relations*. It provides reliable measures for unbalanced communication patterns in different supersteps and models *locality* (network proximity) in communications.

Implementations

Implementations of the BSP model in generic architectures have been developed since 1993. Mainly the Oxford BSP library [138], the Green BSP library [92] and the BSPlib library [101] which includes Direct Remote Memory Access (DRAM) and Bulk Synchronous Message Passing (BSMP). BSPlib has almost become a standard or at least a reference point for BSP implementation research and programming. New implementations with nested parallelism approach are discussed below.

Synchronization architecture

The SA of the BSP model is highly restricted. The only condition synchronization structures allowed are sequences of supersteps, and a superstep is a parallel composition, always with the same degree of parallelism. Each parallel thread is a series of tasks of any length. These structures are in the *bulk-synchronous* subclass of SP CS class. The restriction of using always p processors in each superstep is not important for the SA point of view. The latency parameters L, l, g are typically dependent on p. The purpose of fixing p is to use fixed and known values of parameters throughout all supersteps for cost formulae simplification. At the same time it is a reasonable choice to use as many processors as possible during all the program execution.

Some kind of contention is produced by the arbitrary arriving of messages sent to the same processor. Thus, even if no ME primitive is considered in the model, ME can be programmed and the SA is in the ME class.

For the same reasons discussed in the PRAM model in section 2.4.1, the SA is static and non-data-dependent, in the sense that the programmer do not face the problems of dynamic communications among processors. She/he sees only one bulk synchronization and communication step, independently of the internal dynamic structure created in lower implementation levels. Programming data-dependent applications is possible, but the bulk synchrony barrier system is responsible for transforming the dynamic structure into an static one, making it transparent for the programmer. Thus, we consider BSP to be in the (SP,ME,NDS) class.

Discussion

The thesis of Valiant [185] is that when the programmer uses enough *parallel slackness*[2] the model behaves neutral with respect to the number of processors, and the programs run efficiently as long as the communication is at least balanced with the computation. The value of L can be pre-calculated for any machine and h value combination, for any program to run with optimal efficiency (in constant factors) for this model.

It is claimed by Valiant that the implementation of this model in any architecture is possible loosing only little efficiency (no logarithmic losses). Successful implementations of BSP models and applications confirms it for many cases (see references in [173, 100, 91]). The model lets the programmer determine which algorithm is better suitable for any machine simply checking the results of the cost model for the given parameters measured for the machine, and knowing the h-relation cardinality of the algorithm.

[2]Programs are written for v virtual processors to run on p physical processors where v is much larger than p (e.g. $v = p \log p$)

Nevertheless, objections and counter-objections to this model are stated. The accuracy of the cost model is not so high, although it is “very reliable in modeling the overall behavior of an application, including the prediction of *breakpoints* at which the performance changes” [91]. In the same paper it is also claimed that the accuracy could be increased by adding new parameters, but this will made the model more complex and the algorithmic trade-offs less obvious. Nevertheless, as far as the SA does not change, the main analyzability properties that leads to an affordable cost model will not change. The choice of modeling parameters of the underlying machine is a trade-off between accuracy and complexity that can be applied to the same cost modeling techniques [70].

BSP cost model ignores possible delays due to contention problems derivated by many processors sending messages to a given processor at the same time. The solution is to use specific message ordering adapted to the computation. Implementations of BSP can do it internally, but most of the time the programmer should be aware of the problem and provide a solution changing the order in which messages are sent in the algorithm [112]. Thus, the programmer is facing a mapping problem derivated by the limited number of resources (processors and network interfaces).

At the same time full barrier synchronization is claimed to be an expensive mechanism that most machines do not provide by hardware, and a mechanism with no fixed cost, which scales-up with the number of processors. Hill and Skillicorn studied the practical implementation of barrier mechanisms in [102]. The performance of the different mechanisms available in shared memory architectures is good enough, but difficult to predict without very low-level detailed knowledge. For distributed memory architectures, which rely on message-passing models, performance of barrier synchronization is predictable and reliable, but poor in general. However, better synchronization systems are constantly developed and it is reasoned that synchrony is an important feature to improve analyzability and correctness proofing. The model suggests this direction for hardware developers. Software alternatives to direct barrier synchronization exist:

1. When $h = p$ and every communication is delayed until the end of the superstep, the communications can be optimized and the barrier is implicit in the $h = p$ information exchange [57].

2. A special system of zero-cost emulation of a barrier that can be used in special circumstances was proposed in [62, 8]. It is implemented in the PUB library with the name *oblivious synchronization* [25]. When every process knows exactly the number of messages that other processes are going to send to it during a superstep (the exact receiving h-arity), when it receives that number of messages it can proceed to the next superstep. The consistency is maintained, as long as no process uses data received during a superstep until next local superstep begins.

3. A relaxed barrier synchronization emulation by a handshake protocol only between communication processes is proposed in [121]. It uses the number of the superstep (s) in the sending process as control information in the message to keep the consistency statement (data are not used before the $s + 1$ superstep in the receiving process). The efficiency of the system is improved due to the relaxation of the synchronization phase.

However, these systems only work under special assumptions (e.g. known number of receiving messages), and the relaxation of the synchronization compromises the simplicity of the cost model because of a change in the SA. It is difficult to deal with the lack of synchrony and still keep cost measures tight, especially in irregular or not highly-balanced problems. A new cost model should be devised, but the lack of synchrony can lead to NSP SA and non-reducible cost models. An approximation to this problem has been presented in [81].

Another profitable feature is that BSP model is mainly used in the most general case where $h = p$ (assuming a full interprocess communication in each superstep). For this case, implementations may exploit the implicit and explicit knowledge of the communication global structure. Repacking, destination scheduling and pacing techniques used in an implementation of the BSPlib improve performance to a factor of approximately four comparing with a generic message-passing interface (MPI) [57]. This is a good example of how restricted and structured synchronization architectures lead to performance improvements in implementation.

An interesting proposal for increasing the expressive power of BSP and maintaining or even improving the performance, somehow related to the idea of subdividing the BSP machine, is the Collective Computing Model (CCM) [163]. In this model the number of possible communication patterns at the end of a normal superstep is limited to a chosen subset that includes all typical collective communication schemes. Accurate cost measures can be obtained for them, and specific efficient implementations are possible for such a limited number of well-know communication patterns. At the same time they propose a new special kind of superstep, the *division* superstep, that splits the processors in groups, distributes data among them, computes specific tasks in each group, and redistributes the results, always trying to benefit from the reduced number of efficient communication patterns. However, the division steps are rigid and cannot be nested to extent the SA to an SP class. The model keeps the great simplicity of the BSP cost model even in the division steps as the SA is still *bulk-synchronous*. This direction is mainly focused to the integration of BSP with the efficient and performance predictable collective communication operations, that are so commonly used in message passing environments (see section 2.5.4).

Nested parallelism in BSP

It has been said that “global barrier synchronization is an inflexible mechanism for structuring parallel programs” [135]. Trying to keep the good properties of BSP model but getting more expressive power, some new versions of BSP include support for the concept of nested parallelism (SP synchronization architecture) using the nested BSP computers concept. The whole BSP abstract machine, with p processors can be recursively subdivided in k BSP submachines, each with k_i processors that work like a small BSP machine, synchronizing their processors independently of the other sub-machines. When a subset of (perhaps also subdivided) machines end their work, they must wait to be synchronized together. Examples of the implementation of this idea can be found in the Paderborn University BSP (PUB) library [25], NestStep [119] (that also includes support for virtual shared memory), H-BSP [39], and NBSP [80].

The BSP model assumes that the computer has a global synchronization mechanism (a *bulk* property). It has been argued that synchronizing a subset of executing processes can be a complex issue [173]. However, the same report states that architectures in which barrier synchronization is implemented in software can make to it without any problem. Moreover, many works oriented to fine-grained parallelism indicate that nested parallelism can be implemented efficiently (see e.g. [180, 131, 19]).

Nested BSP has basically an SP SA class. Thus, the cost model of a nested BSP can use the compositional analysis properties of SP class over the local BSP cost models. For example, a simple nested theoretic BSP cost calculus named miniBSP was introduced in [172].

If the subsets of processors may be chosen dynamically by data-dependences, the synchronization structures are no more static. Nested parallelism move the SA to real SP class, where dynamic construction of the nesting is possible. Thus, nested BSP is in (SP,ME,DS) class. SP languages map without much trouble in any implementation of a nested parallel BSP model.

Conclusion

The BSP model proposes a highly restricted SA (bulk-synchronous) to obtain a very simple and easy-to-use cost model. At the same time, full synchronization helps in software development because it makes much easier to reason about correctness [91]. For example the *refinement calculus* can be used to check correctness in BSP program building [171]. In the same report it is also said that this technique can be also used for nested BSP. Refinement calculus works in a recursive framework, being useful for all SP class models.

Although the programming discipline imposed by the bulk-synchronous architecture is very user-friendly and easy to understand [91, 72], no software engineering techniques that helps the programmer to *flatten* more complex SA schemes to only one-dimensional parallelism exist. Automatic flattening by the compiler has been only achieved for SIMD parallelism [119], as e.g. in NESL [18]. Simulations of other models (as PRAM) are possible in BSP, but for real efficiency direct BSP algorithm design is desirable. At the same time the programmer is faced with data-partition problems, as the point-to-point message system forces to explicitly know where data are and where they must be moved to be used. Data-layout is then fixed in the final algorithm.

It is an interesting question to determine which range of applications can be efficiently programmed in a bulk-synchronous scheme [91]. No measures of the potential loss of parallelism inherent to the full barrier synchronization have been previously shown. Measuring the distance from BSP programming to a more expressive or generic model is an important issue in this dissertation. Although we focus in the more broad SP synchronization architecture class, we show that most of our results are applicable to BSP programming.

2.4.3 Class (SP,ME,NDS): QSM

The *Queue Shared-Memory* model is the evolution of the QRQW-PRAM model (see section 2.4.1) to a bridging parallel computation model based on latency-contention in a shared-memory environment. It tries to keep the simplicity of use of shared-memory with the same cost model features of BSP or LogP models (see section 2.4.4). QSM detailed description and rationale can be found in [76].

Description

The QSM machine model has a fixed number p of processors with local memory and connected to a shared memory global space. Every cell has a queue of read/write operations that deals with the contention of many processors trying to read/write the same cell.

Processors execute synchronized phases. A phase is an arbitrary interleaving of three possible operations:

Local computations: Each processor i performs c_i RAM operations in its local memory.

Shared-memory reads: Each processor i reads r_i shared-memory cells, copying their contents into the local memory. Shared-read operations are not guaranteed to complete until the end of the phase. Thus, values cannot be used before the next phase begins.

Shared-memory writes: Each processor i writes to w_i shared memory cells.

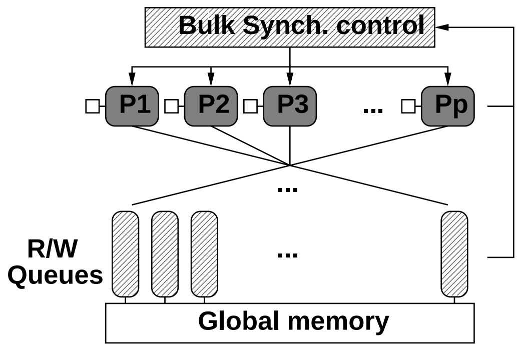


Figure 2.19: QSM computing model

Concurrent reads or writes in a given cell are possible during the same phase, but not both. The read/write restrictions allow the emulation of a QSM machine in a MIMD environment, pipelining the shared memory accesses to amortize the latency of remote accesses. When multiple writes are issued, any one of them finally succeeds.

A phase finalizes when the local computations finish in every processor and all the read/write operations pending in the R/W queues of shared-memory cells finish.

QSM proposes only two parameters to model the architecture features:

p: The number of available processors.

g: The latency parameter for read/write operations in shared-memory. It represents the *gap* between local instruction rate and communication rate due to limited bandwidth in the processor interface.

The cost model of a phase represents contention vs. computation vs. communication. Let the *maximum contention* k represent the maximum number of processors reading or writing to a given shared-memory cell during the phase. Let $m_{op} = \max_i\{c_i\}$, and $m_{rw} = \max\{r_i, r_w\}$ for the phase. The total cost of the phase is:

$$T = \max\{m_{op}, gm_{rw}, k\}$$

Synchronization architecture

The CS SA is bulk-synchronous. The model works in synchronized phases, no read/write are allowed in the same phase in the same shared-memory cell, and shared-read values are not obtained until the next phase begins. Thus, no condition dependences can be produced except from one phase to the next. The dynamic data-dependent structures are reduced to static ones due to the bulk-synchrony (see discussion in section 2.4.2). This model has no ME primitive, but it supports ME. Read/write operations invocated by the programmer can content in the queues of the shared-memory cells, allowing programmed ME.

Discussion

The model presents the facility of a shared memory space, to be used with the usual read/write operations. However, the semantics of the read operations is modified (values can not be used before next phase), in a way that is equivalent to the consistency statement of BSP model.

In fact there is a highly inherent conceptual equivalence between QSM and BSP model. The read/write accesses to the shared-memory have similar semantics to message passing, and they are done in two phases that can be pipelined by the processors. Each phase is charged with a similar latency parameter g. The h-relation is substituted by many concurrent writes, and many concurrent reads.

The main difference with BSP is that in QSM the contention in the arriving messages (read/write operations in this model), is accounted explicitly. Thus, the cost model can predict contention problems due to non-balanced communication patterns (bottlenecks that appear when many read/write concurrent operations are issued to the same shared-memory cell). An interesting remark is that the model do not charge any cost for the synchronization mechanism. This can favors programming with too small computation phases and many synchronization points. The cost model does not penalize this practice.

Efficient emulations of BSP are possible in QSM and viceversa [76]. The relationship of emulation possibilities between QSM, BSP and LogP models is presented in [160]. The main results indicate that these latency based models are quite similar in computational power and modeling solutions of real architectures. QSM has the advantage of a comfortable interface based on simple shared-memory operations, making the data-layout transparent for the programmer.

This model exploits the highly restricted bulk-synchronous static SA to allow the insertion in the cost model of a simple account of the contention scheme (that allows ME), assuming that the bulk-synchronization waits for contention problems to be solved. Again, a restricted static SA shows its analyzability benefits.

2.4.4 Class (NSP,ME,DS): LogP

Another important model based on messages and network latency modeling is LogP [49]. It tries to overcome PRAM and BSP models limitations by creating a more realistic and detailed model of real parallel computers. In fact, its SA is in the most flexible and expressive class. Its success is still compromised by the higher complexity of use that it introduces.

Description

In this model the idealized computer architecture is similar to the BSP concept of independent processors with local memory and a non topology-detailed network represented by few parameters (see Fig. 2.20). This is a representation of a generic distributed-memory multicomputer where processors communicate by point-to-point messages. Compare it with the block diagrams of message-passing machine models in Fig. 2.15.

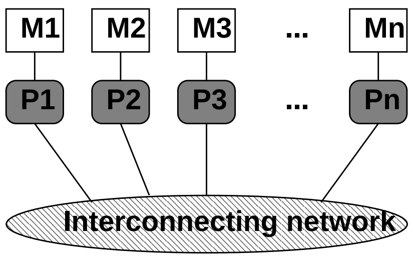


Figure 2.20: LogP computing model

The main new features are:

Asynchrony: No synchronization device is considered, as in BSP. In LogP, processors work at their own path and do not synchronize except by explicit message-passing instructions included in the program.

Limited bandwidth: In LogP does not exist the concept of h-relations in synchronized phases to limit the amount of traffic. Processors can communicate to their heart's content, limited only by the speed of their network ports (modeled with new parameters). Thus, the network capacity can be overcome, producing a stall state. The limited bandwidth of the network must be considered.

The cost model includes the following parameters:

L: *Latency* upper bound of the communication of a small message (in the order of a few words).

o: The *overhead* or time during which a processor cannot work when engaged in sending a message. It has been argued that new network interface technology has reduced this overhead to almost negligible times. Thus, some authors claim that this parameter may be effectively disregarded.

g: The *gap*. Time interval between consecutive messages reception or transmission in a given processor.

P: The number of computing elements (processor, memory and network interface modules).

The capacity of the network is limited by the parameters. No more than $\lceil L/g \rceil$ messages can be in transit through the network at the same time. Processors that try to transmit over the capacity of the network stall until the network is not saturated. Messages that produce stall states can take more that L time units in being sent.

The description of the model includes the following remark: "an algorithm must produce correct results under all interleaving of messages consistent with the upper bound of L on latency" [49]. For the general cost model all messages are assumed to incur in the worst case latency of L. Although some examples are provided in the literature, no general procedures to derive cost model formulae are proposed, as each algorithm can present a complete different behavior that must be analyzed on its own.

Synchronization architecture

The model assumes asynchrony in the processors work, and point-to-point communication without restrictions. Any static as well as dynamic synchronization structures are possible. The expressive power is big, being the SA in the NSP and DS classes. ME can be explicitly programmed due to the unknown interleaving of messages during network transit. No order rule exist in message arriving, allowing non-deterministic contention. Thus, the SA is in the (NSP,ME,DS) class.

Discussion

Due to its NSP condition synchronization, the LogP model does not offer a simple analytic cost calculus for performance prediction. For a generic application that can use unstructured programming constructions, it is usually not possible to reduce the cost expressions to simple formulae only dependent in given application and model parameters values. At the same time, the complexity and asynchrony of unstructured computations prevents simple debugging techniques based on global state checking.

Scheduling, data partition and mapping decisions are completely faced by the programmer. Even more, the stall states in the network must be detected and prevented by the programmer, as the contention is not represented in the cost model (see details about LogGPS below).

Performance predictions are computed in an unrestricted structure of communication produced by the implementation of the application, and results cannot be projected backwards through the implementation transformations path (is impossible to automatically determine what effects are produced by each transformation or implementation decision). The fast growing complexity of testing any possible mapping or transformation make the testing of a wide range of choices impossible. Thus, the model gives little help for software development in the generic case.

LogP has been proven to be useful for optimal design and performance prediction of low level applications [116, 50]. Some low-level implementations of message-passing could support the LogP model of computation. However, its simplicity of parameters and machine modeling is not enough to predict the optimized operations of a complex message-passing interface such as PVM or MPI [5]. Extensions to the LogP model include:

LogGP: Support for long-messages communication latencies [6].

LogGPS: Variable overheads to simulate implicit synchronization of processors before long-message transmission in message-passing interfaces [109]. LogGPS is indeed a complex architecture-oriented model, which includes hidden features of optimized messages-passing interfaces like MPI.

LoGPC: Contention in network traffic [139].

The first two extensions model the underlying architecture with many low-level details, obtaining improved accuracy for specialized cases. However, the SA is not changed, and the difficulties of applying the model are still coming from the unstructured NSP synchronizations.

The LoGPC model presents the same problem as long as the SA is also not changed. However, the contention costs are considered and added to the cost model. Thus, it eliminates an important problem of the LogP cost model, where applications were not encourage to be designed with communication patterns that do not cause stall conditions due to contention. Low level trade-offs between contention, communication and computation can be modeled.

Conclusion

Although it is similar to BSP as a latency oriented model, and substantially equivalent as a computation model in asymptotic analysis [16], LogP presents worse software development features (e.g. easy of programming, correctness checking and debugging). In this model, the programmer does not only face data-layout but many other mapping problems like explicit scheduling. Any set of mapping decisions lead to a new algorithm that must be analyzed in detail with the cost metrics. The LogP model has an NSP SA that prevents easy and methodical generic algorithm design, driven by a cost model.

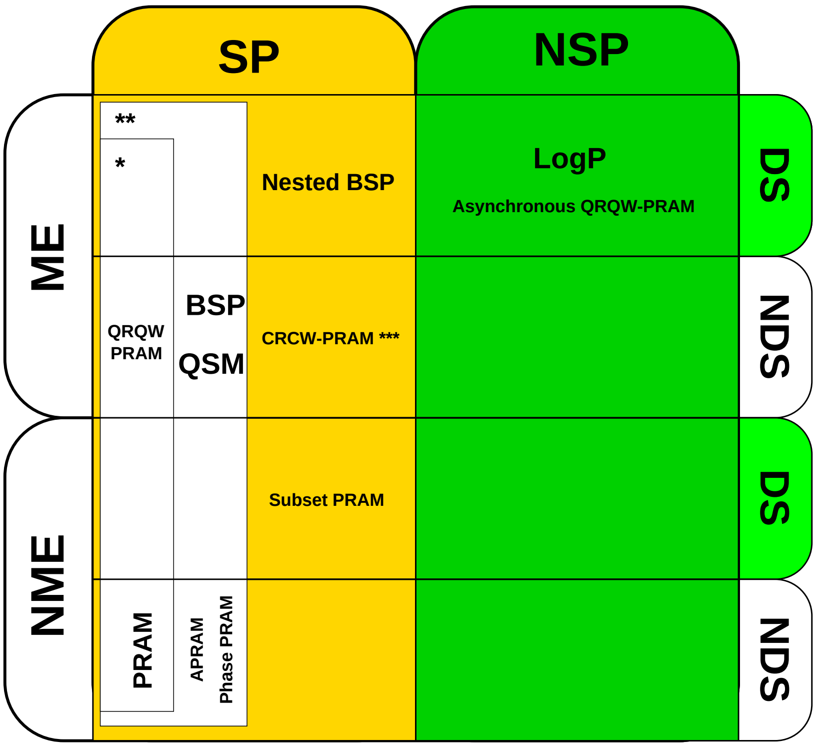
Its extensions can better represent the behavior of the underlying architecture, and predict it with better accuracy than the basic model. Thus, they are more suitable for low-level analysis of optimized routing, scheduling and communication schemes and tools. Portable low level layers or last phases of parallel applications implementation can be designed and studied with these extended models.

2.4.5 Conclusions about PCMs SA

The graphical classification of the discussed PCMs SA is shown in Fig. 2.21. In this section we present some important conclusions about it.

After this review of parallel computing models the main conclusion is that SA is a key component of a PCM for its expressive power and analyzability features. Specifically, the CS axis becomes the most related to the complexity of the associated cost model. SP and NSP classes show important differences. The analysis complexity of the NSP structures becomes too hard for anything but toy problems. Restricting the CS structures seems necessary for achieving the PCMs/PPMs requirements proposed in section 2.1.5. SP models appear to be good candidates for their simplicity of programming and analyzability. However, we must determine the expressive power of these models, which types of applications may or may be not inherently SP, and check if it exists a systematic form to map more unstructured parallel computations into SP forms. More restricted CS subclasses of SP, as lockstep or bulk synchrony, provide only better analyzability if important expressiveness restrictions are assumed (as PRAM), where programmer finds even more troubles to map NSP applications.

We have classified PCMs in NDS or DS structure from the point of view of the synchronization structures created at programming level. Highly restricted models (lockstep and bulk-synchronous CS) appear to be highly static and data-independent. However, the implementation of the restriction mechanisms (lockstep or barrier) is the responsible of hiding the dynamics of the communication included by the programmer, to keep the structure static. In this sense, restricted PCMs provide only static synchronization structures, but they anyway allow the programming of dynamic or data-dependent applications. It would be a risky restriction not to support data-dependent communication structures, as many applications need them (see section 2.6). All PCMs, except PRAM model, consider a fixed number of processors. Applications that dynamically generate threads may need extra programming to pre-schedule the threads into the fixed number of processes. This shows that PCMs are oriented to the mapping level, where cost models become important. Models in full SP class include a dynamics level not which does not appear in bulk-synchronous and lockstep SP subclasses.



Notes: * Lock step subclass of SP

** Bulk synchronous subclass of SP

*** Only for non-deterministic contention policies

Figure 2.21: SA classification of PCMs

The origin is the possibility of data-dependent control of the parallelism nesting. This dynamic makes the static cost analysis much more difficult and not always possible. Programming techniques that do not allow dynamic control of nesting in SP models, would be more desirable with respect to cost analysis.

Only more restricted PRAM models do not allow ME, because it is inherently avoided by the lockstep system and contention solving policies. However, this situation restricts some of the expressive power in the model. Some applications that need mutual exclusion (see section 2.6) can not be directly programmed in these restricted PRAM models. The PCMs studied that include ME mechanisms have something in common: Instead of using primitives with implicit ME semantics (as locks), the ME is programmable by queuing up memory accesses or messages, assuming non-deterministic orderings. Some restrictions to the queue lengths may directly or indirectly help in cost modeling (for example limited bandwidth, limited number of messages among processors, or limited number of messages in the same computation step). The reason is that contention queues model real machine effects produced in the lowest level.

It seems that exists a convergence in the restrictions that PCMs assume to provide an accurate cost analysis for generic applications. Nowadays, bulk-synchrony is a typical feature which allows the transformation of dynamic and complex synchronization structures in static and analyzable ones. ME is supported by programmable contention through (somehow limited) queue systems of non-deterministic reception. However, the completely unrestricted message-passing oriented models, as LogP, are popular because they model the current trends of high-performance programming, where complex synchronization structures are generated due to manual low-level optimizations.

2.5 Parallel programming languages and models

Our trip is taking us now to the mountain shoulders, where PPLs provide the programmer with higher abstraction levels. In the shadowy depth of narrow valleys, near the quick waters, we will find classical approaches that lead directly to the PCMs low lands. Trekking up-hill through the more dangerous rocky paths, we will make to the more abstract PPLs. In hidden glacier valleys new models with different conceptual proposals feed the rainbowed waterfalls, which impressive view we enjoyed from the valleys.

We review many popular and conceptually interesting parallel programming languages. They have been designed from the higher abstraction levels, but they also implicitly impose a PPM. Most of the time, languages that have been developed with other design principles in mind than to be good parallel computation models present different approaches and solutions to the analyzability vs. expressiveness problem. We will study some of them in terms of their SA and other characteristics related to the decisions taken during the model design. The expressiveness vs. analyzability trade-off is considered in each case. A more detailed study of parallel programming languages and a comprehensive classification can be found e.g. in [174].

2.5.1 Class (SP,NME,DS): Pure nested parallelism

Some languages include only pure nested parallelism structures of synchronization. A well-known example is Cilk [19, 42] (see other examples commented in [187]). This language proposes a multi-threaded model, where spawning and joining of threads is only possible hierarchically. The only possible synchronization between threads is through the spawning/joining process. Thus, the possible synchronization structures are always in SP class and no ME exists in the model. However, spawning of new threads can be data-dependent, with no restriction for the number of threads that are spawn at any time (the programmer does not concern about the number of real processors). Thus, the SA is in class (SP,NME,DS).

The good point of Cilk is that it uses the analyzability advantages of the SP structure to implement a run-time work-stealing scheduling algorithm. It achieves good performance even with highly dynamic structures. Many applications with typical non-structured solutions have been programmed in SP structured Cilk, experimentally showing minimal loss of performance [42]. The SP structure can be further exploited with other simple scheduling policies to be efficiently adapted to wide-area and hierarchical networks [187].

2.5.2 Class (SP,ME,NDS): Nested parallelism with ME

The nested parallel SP programming languages that support ME include specific primitives with ME semantics. We find in this class an important programming set of primitives oriented to shared-memory architectures (OpenMP), as well as more abstract proposals (as SPC). Both are oriented to static and non-data-dependent synchronization structures. Nevertheless, both examples may create less restrictive SA structures when mechanisms not promoted but supported in the models are used.

OpenMP

The OpenMP [149] programming tool has become a major trend for programming in shared-memory machines (and possibly distributed-memory in the future, as several proposals for mixed message-passing and shared-memory support are appearing [36]). The main advantage of OpenMP is that it provides the programmer with a portable and easy to understand interface of *pragma* directives to parallelize sequential code (for reusability purposes), getting profit of the shared memory capabilities of the underlying implementation. OpenMP is the result of a common effort of several vendors and corporations, thus, it is well supported and is widely being used.

Shared memory accesses should be controlled to avoid race conditions. The typical way is to include a dynamic non-deterministic accessing mechanism to create ME. OpenMP provides two types of ME directives to create *critical sections*: (1) For code pieces, or (2) for atomic access to a given variable for a single operation. At the same time it allows a parallel section of code to declare their own private variables for programming flexibility (which do not introduce new properties in the synchronization mechanisms). The main parallel control directives provide only *nested parallelism* for code sections, or for loops in a data-parallelism fashion. However, current implementations may support only one level of parallelism, running sub-threads sequentially in the main thread that creates them. Thus, the main programming model is nested parallelism (SP class), while most implementations relay on a more restricted BSP like model. Global reduction operations[3] and a barrier mechanism is also supported. Although not popular, designers of OpenMP included compulsory support of an external library for lock-variable based synchronization. It has been added to let the programmer to create any kind of complex CS structures. Thus, the full implementation moves to NSP and DS classes.

Thus, the spirit of the OpenMP model is in the (SP,ME,NDS) class, or even the (SP,ME,DS) class if data-dependent control of the (mostly unsupported) nested parallelism is allowed. But the use of the external lock variable mechanism allows all kind of unrestricted structures: (NSP,ME,DS) class.

OpenMP does not propose a specific cost model or software engineering methodology. However, while using only the nested parallelism (SP class) condition synchronization scheme, the restrictions included in the design allows program compilers to include interesting mapping and optimization features. However, the semantics of OpenMP nested directives are complex and poorly defined [44].

Because OpenMP is designed to operate in shared-memory environments, processes have direct access to the full memory space. Thus, in NUMA machines any variable usage may imply a bounded but unpredictable cost for the memory access or communication. Shared memory accesses, not marked by a dynamic synchronization mechanism, could produce inherent communications and synchronizations that change the apparent structure or produce non correct results. These perturbations can only be detected by the compiler using data-dependence analysis of the sequential code and internal data deployment information.

The OpenMP standard does not include data distribution directives. Although interesting for the unification with a distributed-memory environment [17], recent studies claim that for state-of-the-art ccNUMA shared memory computers *"reasonable balanced page placement schemes incur modest performance losses, and the OpenMP runtime environment can use page migration for implementing implicit data distribution and redistribution schemes without programmer intervention"* [144]. Thus, the programmer can work in a proper abstraction level to achieve portability.

SPC programming model

The SPC (Series-Parallel & Contention) model [71] proposes a restricted SP synchronization architecture that allows improved analysis techniques to be used during the implementation path. SPC is a nested parallelism model plus non-deterministic coordination expressed as mutual exclusion restrictions. An an-

[3]Reduction can be forced to be non-synchronized. But in this case, the values of the reduction variable are undefined until an explicit synchronized directive is issued.

alytic cost estimation model is associated with SPC programs [71, 170]. The accuracy depends on the level of detail of the target machine model used. SPC is designed to obtain benefits from explicit and structured synchronization. It is a programming paradigm with respect to the coordination of the program parallelism, based on a process-algebraic specification model. The model is presented as a coordination language. Thus, its constructs can be used to express parallelism and coordination, using any sequential programming language for computation.

An SPC program consist in a collection of processes equations, mutual exclusion declarations and computation parts associated with processes. Computations are functional units declared in any sequential language, thus their syntax is not specified in SPC. The set of process equations constitutes one parallel process expression through substitution. (By convention, the expression tree is rooted by a special process called `main`). Processes can be composed with serial (;) or parallel (||) operators. For correct binding of compound process expressions, delimiters are allowed ($\{,\}$). Parallel composition works with *cobegin/coend* semantics [9], thus, it implies a full synchronization after tasks completion. No hidden condition synchronization is allowed. The programmer must avoid data-dependences between different processes for program correctness.

Conditional and iterative execution of tasks are supported, although they can introduce a kind of probabilistic (data-dependent) effect that produces dynamic synchronization structures, affecting the performance analysis (see discussion below). Resources are *computation providers* that introduce limitations on the parallelism exploited. A resource can be logical (e.g. a critical section, a server) or physical (e.g. a processor). In SPC, they are modeled with a global name. The programmer specifies which resources are needed to proceed with each task. Mutual exclusion is associated with task to resources assignment ($task \rightarrow r_1, r_2, ..., r_n$). Tasks contending for a resource will be serialized in the scheduling phase.

The SPC model restricts CS structures to those which the associated task graph is *Series-Parallel* [184]. The non-deterministic contention for global named resources has implicit ME semantics. Thus, the SA is in (SP,ME,NDS) class. If conditional and iteration statements are allowed in the process equations, then dynamic structures are possible: (SP,ME,DS) class.

The cost estimation in SPC is based on several performance modeling techniques [70]: When the model allows only series-parallel static synchronization structures a simple analytic cost calculus can be introduced, based on critical path analysis of the generated graph. (See the equivalent calculus for nested BSP in [172]). The mutual exclusion effect in performance can be only approximated. Algorithmic techniques that keep lower/upper bounds are provided in the cost model [70]. Although syntactically not yet provided, the use of resources with several units is allowed in the associated cost modeling language Pamela [69]. In the case of iterative or conditional constructs that are data dependent, with no possible probability derivation, only classical simulation techniques are available to get performance approximations. However, the static and dynamic part of the application performance model can be substituted by the analytic and approximation expressions obtained by the previous techniques, highly improving the simulation performance.

Using only explicit structured synchronization (SP + ME), interesting analysis techniques are possible to help decisions during the implementation trajectory. Full cost-driven mapping to any architecture is possible except for irregular data-dependent applications with no load-balancing or scalability properties.

2.5.3 Class(NSP,NME,NDS): Mapping oriented models

In this section we discuss features of models oriented to express synchronization structures generated by typical applications (like neighbor synchronization, static access patterns and specific data mappings). We study two important examples: HPF as example of the long-ago introduced *data-parallelism* programming model, and some new proposals evolved from the *skeletons* world.

HPF and data-parallelism

Languages based on the data-parallelism paradigm are originated on the SIMD (Single instruction, multiple data) model. In this model, the operands of a given parallel instruction are a set of data pieces with the same type, and all processors execute the same operation on a different subset of them.

In the 1980s there was a significant research in parallelizing compilers. However, sequential languages obscure or eliminate the parallelism inherent to an application with sequential constructs as loops or recursion, that are difficult to analyze for parallelism detection. Writing a parallel program in a sequential language is not a natural approach. In the early 1990s, there appeared extensions of sequential languages that could express the parallelism associated with executing the same operations on different pieces of a data structure partition (e.g. Vienna Fortran [40], Fortran D [104]). Compilers and environments for data-parallelism were widely studied [1]. The most famous language derived from these efforts was HPF (High Performance Fortran) [27, 108].

Data-parallel languages typically include parallel constructs such as parallel array operations, *forall* and *where* statements, and intrinsic functions.

ME typically cannot be exploited in these languages. The data-parallel model allows the programmer to create repetitive static CS structures. The tasks associated with the data operations are synchronized with next tasks through a fixed pattern, as the model simply replicates the same operation, with the same dependences, in each piece of data. Thus, the generated synchronization structure presents a repeated synchronization pattern between each consecutive pair of task layers. In some situations this property can alleviate the analysis problem derived from the NSP structure, but typically any new pattern must be studied and analyzed.

Compilers take advantage of this structure regularity to optimize the codes. Static mapping and scheduling is typically easy. One problem with the model is that the exact synchronization pattern must be extracted analyzing the code inside parallel constructs. Many times the programmer must help the compiler with data-distribution or alignment information. The technique is sensible to changes with the target machine architecture and communication system. The second problem is the restrictions of the model. Only data-parallelism (fine-grain parallelism) can be efficiently expressed. Many applications (coarse-grain, less regular, dynamic, fault prone ...) present *task-parallelism* that cannot be efficiently expressed in this model [32, 33]. Many efforts to combine data-parallelism with or within more generic task-parallelism languages exist [66, 94, 41, 15, 11, 150].

Nevertheless, data-parallelism is an interesting and productive model [30, 145, 110]. Many computing intensive applications or parts of bigger applications (mainly lattice and matrix computations) can be efficiently exploited by data-parallelism methods.

From skeletons to structured languages

Algorithmic templates or *skeletons* try to identify and exploit the structure of a family of algorithms. Parallel structures that have common properties can be used as a skeleton or a programming paradigm. The programmer must identify the skeleton that fits with her/his application, and fill in the exact computation details. Specific compiler transformations and techniques can then be fully exploited. Skeletons are usually implemented in high-order functional languages, where a skeleton function that encapsulates the parallel behavior can receive as parameters other functions that are internally used as the computation part of the generated tasks.

Several libraries or sets of program skeletons have been proposed and studied [43, 53, 26]. Identifying parallel structures present in applications are a key for constructing such sets [31, 152]. More information about skeletons can be found in [45]. A further refinement of the skeletons idea, known as *archetypes* [132], combines broadly-defined computational patterns with data-flow considerations for systematic development of parallel programs.

Skeletons are fixed-structure templates. Thus, the ME and CS structures allowed are the ones defined in each set or library. Each skeleton encapsulates the abstract description of a very concise synchronization structure. Many parallel skeletons proposed are static well-known synchronization structures, such as pipeline or neighbor synchronization. They are specific examples of high regular

NSP structures that have been individually studied and analyzed. There are skeletons that support ME. It is at least typically supported in a given skeleton called *farm* (see section 2.2.2 for the close relation between ME and the farm paradigm). This specific skeleton is in SP and ME SA classes. Data-dependent and data-independent versions are possible.

What skeleton libraries propose is a set of given synchronization architecture schemes for which interesting applications can be derived, and for which efficient specific mapping, scheduling and optimizing methods are well-known. In this sense, the skeletons model is the most restrictive one, since only a given set of fixed structures can be programmed. However, many parallel applications fits in these skeleton structures. The skeletons key is that they capture the common parallel structure of many applications, and can produce efficient and reusable components (see e.g. [54]).

Structured languages

A further step in parallel structure analysis leads to the structured languages approach. In these languages several parallel constructs, based on typical structures found in applications, can be composed to form a more complex application structure (see e.g. P3L [152]). The key of the applicability of this languages is a cost model which is able to compose the predictions based on the basic structures. At the same time, composition of basic structures leads to software development based on well-defined decomposition techniques of the problem.

There are still applications that do not properly fit in the basic structures proposed. They must be modified and mapped by the programmer. The cost calculus is also not so simple and the different techniques of composition increase the complexity of the analysis.

Skeletons in the nested parallelism framework

A new approach to skeletons idea is introduced in the Frame language [44]. In this language a nested parallelism skeleton is implemented as a set of primitives that can be composed generating high-level SP structures. This scheme provides clear semantics and a familiar syntactic framework for programming (SP advantages). In a further step, the programmer has the option of using inside the high level nested parallel structure other low level unstructured computations. This can be done with other skeletal elementary units, or by allowing the programmer to access the underlying communication or parallel software layer in a controlled form. Thus, the programmer has access to the advantages of both, SP programming semantics and specialized and optimized non-SP parallelism. We see this option as a promising research direction. Currently, Frame does not support ME in the high level structure.

2.5.4 Class (NSP,ME,DS): Message passing

The message passing model is based on *communication models* [107, 34, 188]. Standard interfaces and implementations of this model like MPI [48, 140] or PVM [178, 155] are widespread used, and it is nowadays one of the most common models for general parallel programming environments.

Its success is derived from the generic approach it uses, giving only the mechanisms to communicate and explicitly synchronize isolated processes through abstract channels. Thus, it allows the programmer to create and exploit any kind of parallelism that fits a problem solution. At the same time it is a low-level model, for which efficient and highly optimized implementations in real hardware are possible. In fact it is highly related to the implementation level and the message-passing machine model (see section 2.3.1).

In the message-passing model a process is an independent active element. It executes a sequential code and it uses a local memory space. Processes can be created and destroyed dynamically, either by other processes or externally by the system (typically in the context of distributed computing). Abstract named channels can be established between processes for communication. The sequential code can *send* data through a named channel, or try to *receive* data through a named channel. Sent data is kept in the channel until the target process is in a state in which it tries to read it. Synchronization is produced when a blocking receive operation waits for the arrival of a message. Processes can scan several channels at the same time for data, reacting in different ways depending on which channel data is received first.

This kind of point-to-point communication is enough to express any computation and communication scheme. Nevertheless, extended primitives for collective communications (reduction, broadcast, scan, barrier synchronization ...) are included in interfaces and implementations. For these collective operations, using restricted processes groups is typically possible, in order to create virtual communication topologies. To hide many communications in one primitive is a more high level abstraction. Thus, it simplifies programming and allows better optimized implementations of the collective operations. Furthermore, programming with collective operations can lead to even more high-level transformations for performance improvement and software development techniques [90, 88].

The model allows any condition synchronization scheme. Thus, it is in the NSP class. There are no ME primitives, but non-deterministic contention exist, because a process can be waiting for data from different sources at the same time, reacting in different ways depending on the order in which messages are arriving. This feature can be exploited to produce programmed ME. The sending and receiving of messages can be data-dependent. Thus, dynamic synchronization structures are allowed. In many interfaces even the number of active processes may change. Thus, the SA is in (NSP,ME,DS) class.

The counterpart of the model advantages is that being a so low-level model, the programmer faces problems about parallelization grain, data-partition, map and scheduling of any new application considered. As the static or dynamic structures allowed are completely unrestricted, no special heuristics or techniques can be exploited by the compiler or run-time environment for a generic application. The compiler can not match the *send* and *receive* primitives for syntactic or semantic validation or reasoning.

The theoretical models on which message-passing is based, provide a calculus to derive possible or forbidden states of the system. Nevertheless, the fast growing complexity of the search space makes such tests intractable for anything but toy problems. Extensions of LogP also try to model the internal features of message-passing interfaces (see section 2.4.4), but they offer no help in software design. They can predict the communication behavior of a given communication pattern, but do not provide a systematic procedure to analyze a full subset of the possible solutions or design search space, due to the NSP SA.

However, the message-passing interfaces hide the communication details, and can be used as an efficient abstract communication layer when accurate measures of given communication patterns are affordable. More high-level programming techniques can be applied or integrated in an environment that, underneath, uses message-passing for communication [87, 182, 191].

Other authors complain about the non-deterministic behavior of message-passing interfaces. It leads to non-reproducible and more difficult to debug developments, that is antithetical to scientific methods. An interesting approach to eliminate the non-determinism in a message passing model is FortranM [67]. It is based on extensions to sequential languages (in this case, Fortran) with semantic and syntactic restrictions in the creation and manipulation of communication channels. Nevertheless, FortranM provides non-deterministic constructs for applications where it is needed. Thus, the programmer can restrict the use of non-determinism and she/he has more control on the type of SA used (NME or ME). Its modular or object-oriented approach make it easy to couple with data-parallel modules (see section 2.5.3).

2.5.5 Class (NSP,ME,DS): Maximum abstraction

In this section we discuss two more abstract example models that fit in the SA class with maximum expressive power: Concurrent object-oriented programming and tuple spaces. They present a PPL/PPM with powerful semantics. Many PPL solutions include both of them. The counterpart is the problems of cost analysis and efficient implementation.

Concurrent object-oriented programming

In a pure concurrent object-oriented model, a computation is a collection of processes that access and use shared objects with a contention mechanism to avoid race conditions. Thus, it can be viewed as a model without CS that relies only in the contention mechanism to control the parallelism. The contention is controlled by monitors associated with objects. A monitor also implements a mechanism to wait for or notify the success of a guarded condition [106]. Thus, condition synchronization is affordable if a complete monitor implementation is provided in the language.

For example, the JAVA synchronization model is based on inherent monitors associated with the objects. Not only methods, but also code pieces can be made mutually exclusive using the monitor associated to a given object. The primitives wait, notify, and notifyAll, associated with the Thread object, can be used inside synchronized methods, along with specific condition fields, to create and control condition synchronization.

The underlying model for concurrent object-oriented programming is also a message-passing model when non-shared-memory architectures are used. Remote method invocations create communication channels for the data interface when accessing objects information across processes. Three main differences (advantages) can be observed with respect to pure message-passing:

- The remote method invocation is done across a shared name space of objects.
- ME can be directly used as it is implicit in method invocations controlled by monitors.
- Data are associated with objects and methods. Although data partitioning decisions are still faced by the programmer, they can be helped by this arrangement.

From the previous discussion it follows that a basic concurrent object-oriented model has no CS and uses only ME to control parallelism. The implicit contention mechanisms (calls to monitor protected methods) have ME semantics. The SA is always in class ME. However, monitors allow the creation of condition synchronization and certain implementations make use of remote method invocation to create other CS mechanisms. Both lead to NSP structures. Condition synchronization structure is unrestricted and dynamic. New objects are created and unpredictably used during execution of the system. Compilers and run-time systems do not get much help to decide where to locate objects, or how to schedule processes to processors from the unknown and non-SP structure. The main synchronization control in this model relies on monitors and mutual exclusion, then, in an implicit dynamic synchronization system. However, analyzing dynamic mutual exclusion is not as easy or accurate as condition synchronization analysis [70]. SA in NSP class inhibits analyzability also in the dynamic synchronization space.

The monitor system and the global name/address space makes this model a middle point between pure message-passing and the more complete global tuple space model described below.

Coordination languages *a la* Linda. Global tuple spaces.

The *tuple-spaces* are a coordination and communication system, independent of the computation language [37, 38, 68, 148]. Tuple-spaces provide a PPM with a high-level abstract virtual machine, separated from the computational issues. The PCM is considered to provide a global shared space of data pieces called *tuples*. A tuple is a named collection of data fields of any nature. Processes work asynchronously and exchange data by writing, reading, inserting and extracting tuples in the tuple space. The language also provides primitives for checking the presence of tuples and information in the global space.

The condition synchronization is done through checking, writing, and reading tuples. There is no restriction about which processes synchronize and when they do it. Multiple processes can check the same tuple at the same time. Thus, the language is rich in expressive power and full of possibilities for condition synchronization structures. The counterpart is that it leads to NSP SA class. Operations of checking and reading/writing/modifying tuples can be atomic. Thus, the languages provide primitives with ME semantics. The SA is in the ME class. There are no restrictions to the use of the synchronization mechanism or even to the manipulation of threads. The system is fully dynamic and data-dependent (thus, the model is in the class DS).

Due to the NSP condition synchronization scheme, the cost model presents the problems associated to any NSP model. Efficient implementations on real architectures are not so simple, as the communication problems that arise to maintain the shared tuples are complex. However, the tuple syntactics are clean, and the compiler can do some semantic checking and verification. They provide a good abstraction for a maximum expressive power PPL/PPM.

2.5.6 Conclusions about PPLs/PPMs SA

In Fig. 2.22 we show the location in the SA space of the more relevant models reviewed in this section. The arrows represent a possible change in the SA class when some extensions are added to the basic model. The main conclusions obtained previously for PCMs and bridging models (recall section 2.4.5), are confirmed and extended in this more abstract level. SA is an important feature of the PPLs/PPMs for its expressiveness and analyzability features.

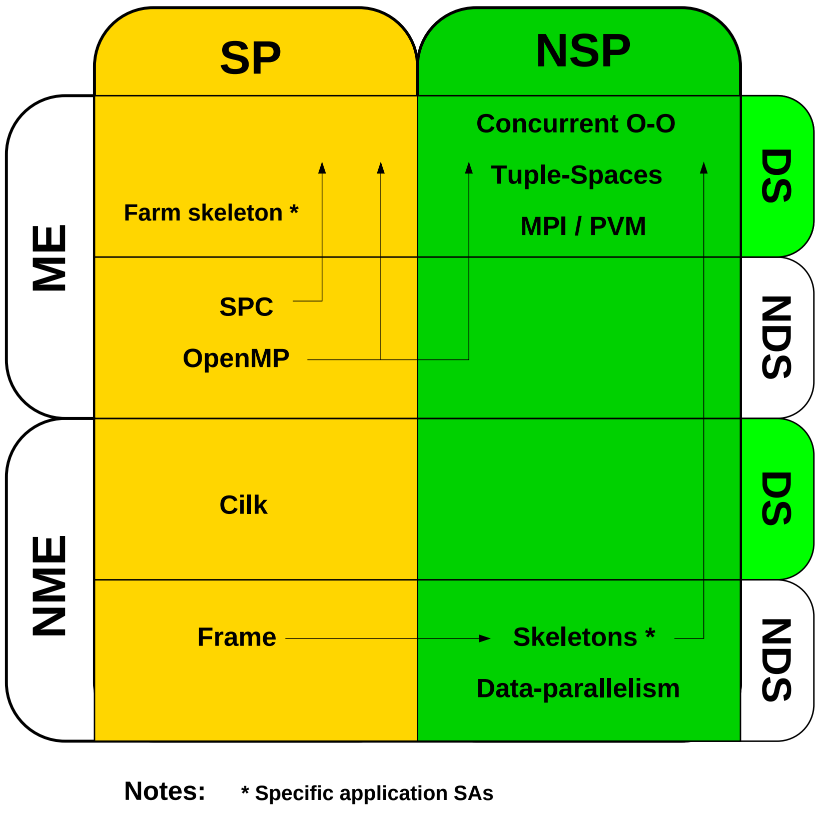


Figure 2.22: Classification of PPLs/PPMs reviewed

Comparing with PCMs and bridging model, a first observation is that the more abstract models usually include ME primitives instead of relaying in a low-level programmable ME mechanism. The lower mapping level is oriented to easier implementations in a full range of real architectures, promoting programmable mutual exclusion. However, in the more abstract level of programming, the models are more oriented to simplify the programming task. Primitives with implicit ME have clear semantics and are easy to use. Except for the case of OpenMP (clearly oriented to shared-memory architectures), the implementation of ME in other architectures is not so direct, although it is clearly promoted from the higher abstraction levels.

At this level we can also see that models which allow NSP CS structures are in the lower or higher extremes of restrictiveness. Models in the NSP classes that offer mapping solutions or reliable cost analysis are based on analyzing and using only a small set of well-know structures and solutions for parallel problems. They are oriented to specific applications structures, like data-parallelism or skeleton based models, that are all in the (NSP,NME,NDS) class. On the other hand, full NSP models are oriented to maximum expressive power, like concurrent object-oriented programming, tuple spaces or message-passing, all in (NSP,ME,DS) class. In these models, the programmer gets little help to understand or predict the system behavior. Mapping and optimization decisions must be taken manually and must be based mainly on the programmer's experience.

At the opposite side of the CS axis, we find SP models. In this case, there is not such an extreme diversification. In fact, the most popular SP based models, either do not support ME (pure nested parallel languages such as Cilk), or are oriented to static synchronization structures (as OpenMP or SPC). The reason is that restricting the SA to the SP class is always introduced in a model to increase the analyzability characteristics of the model. In the case of Cilk, the dynamic scheduling algorithm works with the CS information available. Thus, no ME mechanism exists but data-dependent synchronization is available. In the case of OpenMP or SPC, ME mechanisms are considered, but no data-dependent structures are promoted to still get mapping benefits derived from the static SP structure. However, is important to notice that both SPC and OpenMP allow also dynamic constructions to let the programmer implement any kind of application. In the case of using data-dependent structures, the programmer is responsible for explicitly programming some kind of scheduling and mapping tasks. Therefore, the benefits of using SP structures regarding automatic mapping are prevented. OpenMP goes even further, allowing the programmer to create NSP structures with the lock-managing external library. For a modern and commercial oriented language it would be a real shortcoming if the so many unstructured-mind oriented programmers could not implement their ideas without restrictions. Manual mapping and optimization is still current practice in parallel programming.

2.6 Synchronization architecture of applications

Finally, we are to climb the highest peaks of abstraction, where applications lie surrounded by the clouds of parallel algorithmics. For this upper perspective, we will contemplate all the lands we have previously traveled along. PPLs and PPMs are interfaces to express the parallelism of an application. Thus, we study the SAs present on typical parallel applications, kernels, and parallel problems solutions. We also discuss how do they map to restricted SA classes.

This classification of the SA of applications is intended to help the reader to understand the real purposes, benefits and disadvantages of the different restricted and unrestricted PPMs. At the same time it will point us to case-study applications for the mapping problem (systematically transformation of synchronization structures across different SA classes). In the following classification we are not trying to be exhaustive, but we are only presenting some well-known examples of parallel solutions and applications which are representative of each SA class. The graphical representation of this classification is shown in Fig. 2.26.

2.6.1 Class (SP,ME,NDS/DS)

There are two typical programming paradigms or problem solutions that are based on the use of ME: Farms and non-ordered macro-pipelines.

Farms: Many irregular and dynamic applications are directly programmed using a pure ME scheme through a workers-farm or work-stealing paradigm (centralized or decentralized load balancing scheduling algorithms). Thus, many highly irregular applications, derived e.g. from graph exploration or combinatorial search [156, 189], are transformed to this structure. The scheduling module is then reported about the possibility of non-deterministic synchronization between computation pieces. Applications of this type are typically dynamic (tasks generate new data pieces to process), but the number of tasks may also be statically determined by the problem nature.

ME-Macropipeline: Macro-pipeline is a wide-accepted name for a synchronization structure that represents a generic solution for many problems. Consider macro-pipelines representing problems based on the parallel execution of n processes composed by m tasks or *stages*, such that the stage i of a process needs ME with the i stages of all the other processes (typically due to the use of a shared resource). An example code of such a macropipeline programmed with semaphores is presented in Fig. 2.23. These macro-pipelines can be programmed as a collection of task series with no CS between different series and ME among the i-depth tasks. Other macro-pipelines not based on resource restrictions are not in this class and will be discussed below. The number of stages is known in almost all situations. If the number of processes is also known the structure will be static, else it will be dynamic.

```
MEmacropipeline() {
    Semaphore s[m];
    createSemaphores(s);
    initializeSemaphores(s,1);
    spawnThreads(n);
        process(...);
    syncThreads(n);
    }
```

```
process(...) {
    int stage;
    for(stage=0; stage<m; stage++) {
        P(s[stage]);
        do(stage);
        V(s[stage]);
        }
    }
```

Figure 2.23: Example of a ME-Macropipeline

If the programming model selected for implementation directly supports ME primitives, the application structure is formed by spawning n tasks that synchronize only by ME to obtain more data pieces in the farm, or to avoid concurrent access to the same resource in ME-Macropipelines. If ME is not supported, a false order originally not present in the problem definition should be introduced with CS (see section 2.2.6). The solution probably will incur in high losses if SP synchronization is forced.

It is arguable whether this kind of solutions must be explicitly programmed with ME or they can be even automatically detected and scheduled by a compiler.

2.6.2 Class (NSP,ME,NDS/DS)

No typical parallel applications are found in these classes. In problems where ME is used to provide a solution, it is frequent that no CS is needed except to create sequences of processes that use only ME to avoid interactions, or full barriers to synchronize between iterations. Thus, they can be programmed in a nested-parallel restricted model (SP,ME,NDS/DS). We are not taking into account here implementations that use ME only to simplify communication phases when using a shared-data space. In this case the original application does not really need ME and they can also be programmed in their relative (NSP,NME,NDS/DS) classes.

2.6.3 Class (SP,NME,NDS)

In this class we found applications that directly map to CS structures in the SP class. The problem or solution is hierarchical or highly synchronous. Thus, it can be programmed with hierarchical self-synchronized processes groups. The structure is also static, dependent only in the input data-size or number of processors, and possibly fixed in compilation phase.

Types of applications to be found in this class are trivial parallel computations, static structures derivated from divide & conquer or branch & bound paradigm (sometimes as a data-partition scheme), and synchronized loops.

Trivial parallel computations: Applications that are easily and directly parallelized by a wise data-partitioning avoiding communication between tasks during normal computation phases. The only synchronization needed is to distribute data and collect results. They do not need a powerful NSP language or model to be programmed. Some examples are found in image processing algorithms: Geometrical transformations of a set of different objects in n-dimensional spaces, ray-tracing and other rendering algorithms. Other examples are searching and optimization methods like simple *Monte Carlo* or *hill climbing* methods, specifically when parallel random number generators are used [31, 189].

Reduction trees: Parallel prefix sums, maximum or leader identification, etc [73, 156].

Some sorting algorithms: Merge-sort and radix or bucket sort [189, 120].

Parallel multigrid methods: The overall structure of V-cycle and multigrid simulation programs based in cellular automates (not the cellular automata itself) is hierarchical. Grid local operations to solve partial differential equations, or SOR methods in general, present a divide & conquer SP structure that can be implemented with only one synchronized communication phase per iteration. Many typical solutions to simulation programs in grids use synchronized phases (see e.g. [111]).

Some numerical algorithms: Numerical integration [189]. The overall structure of Strassen matrix multiplication [156] (although local dependences can be exploited in a complicate NSP form).

Synchronized parallel loops: Many applications are programmed with parallel loops or similar structures. After a computation phase, processes interchange boundary information with neighbors, or communicate in an unpredictable pattern with other processes. If the problem semantics need a full barrier synchronization after the communication phase, they can be directly programmed in an SP form.

Because of the easy of programming and understanding of such synchronized structures have, they are used in most situations, even when the synchronization is not in the original problem semantics. For example, the OpenMP [149] model assumes this kind of behavior for its main primitives for parallel loops and sections, although variable-locks can be use to produce NSP patterns at programmer discretion. For well-balanced applications the delay introduced by processes waiting for other processes to synchronize is negligible.

2.6.4 Class (SP, NME, DS)

Divide & conquer may be used as a load balancing technique. In this case data-partitions should be dynamically constructed. Many applications also present an adaptable hierarchical structure that is further or recursively spawned in a data-dependent form. For example, solutions that are recursive over selected pieces of data (like quicksort algorithm) force dynamic structure. However, applications that split data into equal size chunks generate a static structure if the data size is known from the beginning (like mergesort). Some examples of dynamic SP applications are:

Unbalanced sorting: Quick-sort [189].

Some geometric problems: Convex hull or Voronoi diagrams [56].

N-body simulations: Barness-Hutt, Fast Multipole Methods and other non-adaptative hierarchical algorithms for N-body simulation are based on a hierarchical divide & conquer paradigm. (See e.g. [136, 189]). They are intuitively programmed in SP, as they basically construct and evaluate dynamic trees.

2.6.5 Class (NSP,NME,NDS)

In this class we discuss applications which their problem natures imply static non-hierarchical CS structures. The exact synchronization pattern is quite different for different applications. For example, many high regular and scalable applications are generated by replication of a local communication pattern. Most of them are well-known data-parallel solutions, where processes receive a piece of a data-structure partition and proceed in two phases: Computation and communication of boundaries of the data structure with neighbor processors (in a virtual topology defined by the problem, the data partition, and the mapping). They are widely used in simulation and engineering fields and they are specifically studied to obtain specific high-performance optimized solutions. Iterations of a neighbor synchronization pattern defines an NSP CS structure. Many of them present a well-known repetitive synchronization structure that scales-up easily.

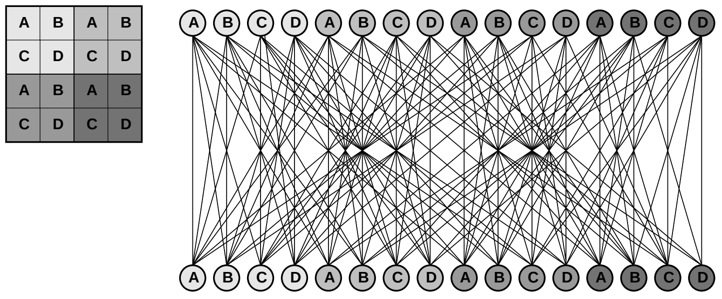

A B A B
C D C D
A B A B
C D C D
A B C D A B C D A B C D A B C D
A B C D A B C D A B C D A B C D

Figure 2.24: D&C block matrix multiplication.

Irregular divide & conquer: Not all divide & conquer techniques lead to SP structures. When the *conquer* phase merges partial solutions generated by other processes, an NSP communication pattern may be natural. In Fig. 2.24 we show the NSP pattern generated by a typical divide & conquer block matrix multiplication, where each processor uses only 7 of the 16 pieces computed in the previous phase.

Static dependent pipelines: Pipelines produced by static code dependences leads to a typical NSP structures. For example, the data-parallel loop presented in Fig. 2.25 creates a macro-pipeline structure that cannot be expressed by ME.

```
(1) FOR i=1,n-1 parallel=8
(2)       $V[i+1] = V[i+1] + f(V[i])$
(3) ENDFOR
```

Figure 2.25: Example of static dependent pipeline

Simple neighbor synchronization: Cellular-automata and other grid/lattice simulation programs based on *stencils* or local synchronization patterns [162]. Example applications include many physics and chemistry simulations or image processing programs.

Problem solving networks: Many applications based in a specific topology exchange network as FFT [153], odd-even reduction or sorting networks [156].

Matrix scientific computing: Most dense matrix scientific computing algorithms like Gaussian elimination, matrix multiplication, QR and LU reductions [79, 78] can be programmed in an NSP form to exploit all possible parallelism. The synchronization structures generated for these applications is not so symmetric as in previous examples. However, they are regular and easily scalable.

For most of these problems, the computation phase is executing the same piece of code on a approximately equal sized piece of data for every process. Synchronized iterations (see section 2.6.3) are very popular for these very regular and high-balanced computations. The performance degradation effect of programming them in an SP PPM is very small [86].

Moreover, specifying these regular computations in a hierarchical synchronization structure, with fine grain parallelism, may allow automatic mapping techniques that perform a good data-partition and load balance, minimizing potential performance degradation.

Another solution is to encapsulate an efficiently programmed solution based on the NSP communication structure into a skeleton [45] or a given language construction [152]. Thus, it can be used compositionally as a language primitive and inside a hierarchical nested-parallel scheme [44].

2.6.6 Class (NSP,NME,DS)

In this class we find applications that generate non-repetitive specific NSP communication patterns depending on the input data and partial computation results. Applications in this class include:

Sparse linear-algebra algorithms: Although most sparse linear solvers try to reduce their behavior to regular vector operations [99, 186], in many specific techniques the synchronization structure is dependent on the matrix density structure (e.g. [124]). In these applications all the structure may be predicted if the matrix structure is known. Sparse linear solvers are an important category of algorithms for many different domain applications, and direct solving methods for sparse linear systems is an important research field (see e.g. [96]).

Simulations in graphs: Many structural engineering applications and similar problems based on iterative PDEs solvers. A graph partitioning algorithm is applied to the input graph to distribute data among processors, minimizing the communication needed due to interactions between points assigned to different partitions [154].

Adaptative grids: PDEs solvers where an adaptative grid is dynamically refined [147]. These problems need dynamic evolution of the data partition, that can lead to dynamic modification of communication patterns.

Dynamic simulations: Adaptative N-body simulations [136] and chemistry or physics simulations, where particles or points are in motion, changing the data elements with which they interact to [115]. In some solutions, the data partition must evolve dynamically.

When the irregular synchronization structure is predictable, once the data structure (e.g. an sparse matrix structure) is known, sophisticated algorithms can be used to transform the structures to SP form trying to minimize the losses [85]. These algorithms may be used even as a pre-scheduling phase. Multilevel graph partitioning may also be used to create nested dissection orderings for solving sparse linear systems of equations [154].

The highly dynamic solutions to simulation problems where communication patterns evolve along iterations are still a big challenge on themselves. In most cases these solutions are heuristic hard-wired load-balancing techniques highly dependent on the problem. Most of the time complex knowledge about the application behavior and decomposition is needed. Good results may be obtained by the hierarchical application of different scheduling polices for processes that show different synchronization roles instead of only one plane policy [115]. However, the identification of such processes classes is not direct and it is not clear how a hierarchical specification of the original problem could help.

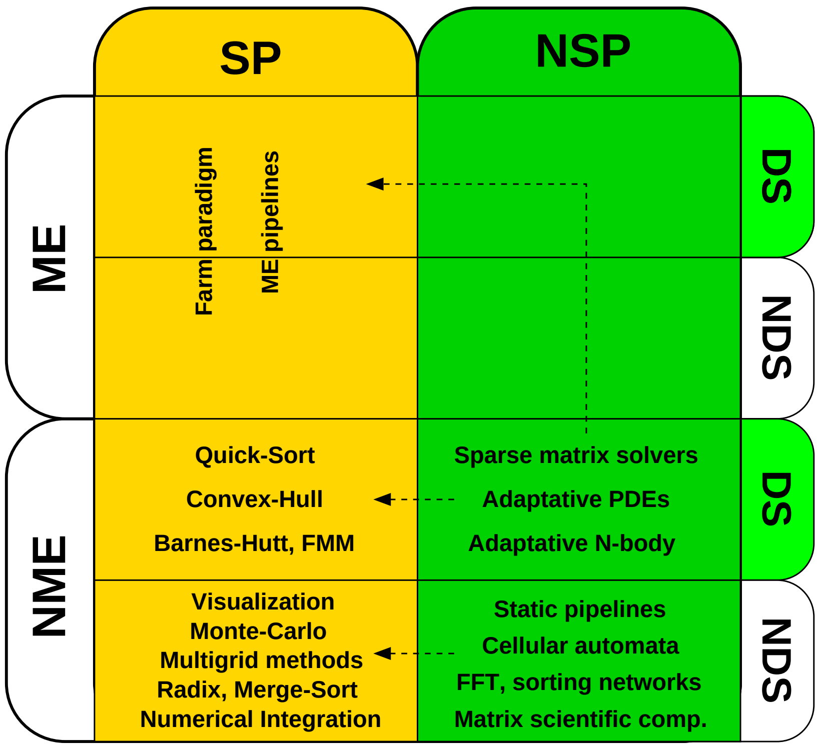


Figure 2.26: Classification of example applications

2.6.7 Conclusions about applications SA

In Fig. 2.26 we show a diagram that summarizes the classification of some example application types described in this section. Dashed lines indicate typical transitions between classes to map applications structure into restricted SP PPMs.

An important observation is that ME is used only to program applications mainly based on two SP paradigms that implicate a specific load-balancing scheduling solution, useful for many dynamic applications. In fact, some dynamic NSP applications may be transformed to fit into the *farm* paradigm, and consequently into SP class. It is important for a PPM to support ME to easily program this kind of dynamic solutions.

Most applications do not need ME. We have found many of them suitable for SP PPMs. For the applications that do really have NSP CS structure we have identified representative examples for any SA class. Simple possible mapping solutions to convey their synchronization structure into SP SA classes have been discussed.

2.7 Summary

In this chapter we have presented the synchronization architecture concept and its relevant classes accordingly to three important criteria: CS, ME and data-dependence. Then, we have explored the different programming abstraction levels to detect the SA classes of PCMs, PPMs, PPLs and applications.

At the lowest abstraction level, execution models provide maximum expressive power and synchronization opportunities. However, as we travel up to the higher abstractions proposed by parallel programming models, we notice that low-level based implementation models (as e.g. message-passing interfaces) are being substituted by higher level models with two main trends:

1. High abstractions with maximum expressiveness power (as e.g. tuple spaces)

2. Restricted models with efficient mapping and software development initiatives (as e.g. BSP).

A parallel computation is a much more complex object than a sequential computation. More and more parallel programmers are accepting that a higher level of abstraction is needed to introduce software development and debugging techniques in parallel programming [89]. However, implementation and mapping problems plague the highly abstract but unrestricted programming models. Nowadays, the programming models that look more promising are those which analyzability capabilities are improved by introduced expressiveness restrictions. In our study we have found that the most relevant frontier in this analizability vs. expressive power trade-off is the SP vs. NSP choice in the condition synchronization axis. Programmers who take the decission of crossing this frontier and force the CS structures to SP form (nested-parallelism), achieve an important increase in their analizability capabilities, opening a full new world of compiling and run-time techniques for verification, performance prediction, mapping, scheduling, portability and software development in general.

Although many typical parallel applications are perfectly suitable for these SP restricted models, some important ones still present a challenge for being efficiently transformed to nested-parallel form. Intuition indicates that in many cases the impact of such a transformation in the application performance is limited. However, the potential performance loss produced by the SP restriction introduced at the programming level, before the application is coded, has not been yet fully studied. The rest of this dissertation addresses this important problem. In chapter 2 we use graph theory to characterize both NSP and SP structures and we study systematic transformations from NSP to SP forms. We also investigate the potential performance impact of such transformations. An experimental framework to verify the propositions introduced in our study, that can also be extended for quantitative evaluation of PPMs in general, is presented in chapter 3.

Chapter 3

Theoretical approach

"This is how I will do it: if there is a whelp of the same breed to be had in Ireland, I will rear him and train him until he is as good a hound as the one killed; and until that time, Culain," he said, "I myself will be your watchdog, to guard your goods and your cattle and your house." "You have made a fair offer," said Conchubar. "I could have given no better award myself," said Cathbad the Druid. "And from this out," he said, "your name will be Cuchulain, the Hound of Culain."

Cuchulain of Muirthemne, 1902
LADY GREGORY

In the previous chapter we have classified SAs and identified the SP (nested-parallelism) restriction as the most important frontier between expressiveness and analyzability. We have also determined that many applications directly map to models in the SP SA classes, but others do not. Although strategies for this mapping are proposed, two important questions arise:

- How much potential parallelism loss is introduced by a transformation which map NSP application structures to SP form, and is it possible to predict it?
- Is it possible to derive automatic transformation techniques to map NSP structures to SP form?

The latter question is motivated by the fact that (1) tool support is an important enabling factor in the use of SP models for NSP problems, and (2) such tools can be used to partially automate the experiments needed to address the first question.

A formal approximation to the mapping of NSP structures to SP form may be developed with the help of graph theory. The synchronization structures of applications have been for a long time represented with graphs. More precisely, *DAGs (Directed Acyclic Graphs)* have been used to represent the *Posets (Partial Order Sets)* or dependences that CS introduce between tasks. These graphs do not directly support specification of ME dependences or alternative structures of data-dependent programs. They may be used to represent only one possible structure created during the execution of a given program in a given PPM (when ME and data-dependences are transformed to CS). Nevertheless, we are interested mostly in the CS structures, as long as we have previously show that CS and ME are orthogonal, and we have determined the impact of ME in the expressiveness vs. analyzability trade-off. To represent the structures created by data-dependent programs we can use several graph representations of the possible structures generated by the program. A complete study of how to extract task graphs from applications is presented in sections 4.2.2 and 4.2.3.

Hence, we will study graph transformations to approximate NSP structures to SP form. The devised transformations will try to minimize the potential parallelism loss introduced by added dependences, that may be responsible of the performance degradation. We study not only the topology impact of a transformation, but the potential impact in the performance through *critical path* analysis. For such an study, the workload distribution of the graph nodes is critical. At the highly abstract level of programming, no exact (or even no) workload information is typically available. In our study, several synthetic workload models are considered. In an experimental study with real applications, presented in section 4.2, we validate and refine these workload models to consider real execution workloads.

In this chapter we use graph theory to formally present definitions and properties of NSP and SP graphs. We also study and compare basic techniques and full algorithms to transform NSP synchronization structures to SP form, minimizing the potential parallelism loss. The impact of such transformations is theoretically analyzed and discussed.

3.1 Graph preliminaries

We present here a collection of mathematical notations used throughout the rest of this dissertation. They are organized in sections about specific subjects: Basic graph concepts, transitive closure and reduction, simple topological parameters, and task graphs.

3.1.1 Basic graph concepts and notations

Since graph-theoretical definitions differ somewhat in the literature, we define here the basic concepts. Definitions are mainly adapted from references [12, 28, 93, 184]. A reader who is familiar with graph theory may skip this section and refer to these definitions later if it is needed.

In this dissertation we denote sets with upper case alphabetic characters $(A, B, C, ...)$, and elements of a set with lower case $(a, b, c, ...)$. Calligraphic upper case alphabetic characters denote set partitions:

Definition 3.1.1 *The symbol $\mathcal{P}$ denotes a partition of a set in non-overlapping subsets:*

$$\mathcal{P}S = \{S_1, S_2, ..., S_n\} : S_i \subseteq S, \bigcap_i S_i = \emptyset, \bigcup_i S_i = S$$

□

Definition 3.1.2 *A* directed graph *G is a pair (V, E), where V is a finite set of* nodes *or* vertices *and $E \subseteq V \times V$ is a set of ordered pairs called* edges*. The number of nodes in a graph is denoted by $n = |V|$, and the number of edges by $m = |E|$.*

There can be multiple edges between the same nodes. Graphs with multiple directed edges are called multidigraphs*.* Self-cycles *(nodes in the form (v, v)) will not be used in our study.* □

Definition 3.1.3 *Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are* isomorphic *$(G_1 \sim G_2)$ if there exists a bijective function f from V_1 to V_2 such that $(v, v') \in E_1 \iff (f(v), f(v')) \in E_2$.*

For the following definitions let $G = (V, E)$ be a directed graph.

Definition 3.1.4 *For each edge $(v, v') \in E$, v is the* source *of the edge and v' is the* target *of the edge.* □

Definition 3.1.5 *For each node $v \in V$, $indeg(v)$ is the* indegree *or number of edges for which v is the target and $outdeg(v)$ is the* outdegree *or number of edges for which v is the source:*

$$indeg(v) = |\{e \in E : e = (v', v)\}|$$

$$outdeg(v) = |\{e \in E : e = (v, v')\}|$$

□

Definition 3.1.6 *A* root *or* source *of a graph is a node v with $indeg(v) = 0$. $R(G)$ is the set of all roots in G: A* leaf *or* sink *of a graph is a node v with $outdeg(v) = 0$. $L(G)$ is the set of all leaves in G.*

$$R(G) = \{v \in V : indeg(v) = 0\}$$

$$L(G) = \{v \in V : outdeg(v) = 0\}$$

□

Definition 3.1.7 *The* successors set *of a node v is the set of target nodes of edges for which v is the source. The* predecessors set *of a node v is the set of source nodes for which v is the target:*

$$Succ(v) = \{v' : (v, v') \in E\}$$

$$Pred(v) = \{v' : (v', v) \in E\}$$

□

Definition 3.1.8 *A* subgraph *of G is another graph $S = (V_S, E_S)$ in which $V_S \subseteq V$ and $E_S \subseteq E$.* □

Definition 3.1.9 *A* Path *from a given node to another $p(v, v')$ is non-empty a sequence of nodes connected by edges that defines a possible way from v to v':*

$$p(v, v') = v, v_1, v_2, ..., v_p, v';$$
$$(v, v_1), (v_1, v_2), ..., (v_p, v') \in E$$

The length *of the path is the number of edges p in the path:*

$$length(p(v, v')) = |p(v, v')| - 1$$

A non-direct path *is a path with length more than 1:*

$$p_{nd}(v, v') = p(v, v') : length(p(v, v')) > 1$$

A Full path *is a path $p(v, v')$ where v is a root and v' is a leaf. $P_f(G)$ is the set of all possible full paths in G:*

$$P_f(G) = \{p(v, v') : v \in R(G), v' \in L(G)\}$$

A Cycle *is a path from/to the same node: $p(v, v)$.* □

Definition 3.1.10 *A node v' is said to be* reachable *in the graph G from another node v iff exists $p(v,v')$ or $v = v'$:*

$$v \preceq_G v' \iff \exists p(v,v') \vee v = v'$$

Where it is obvious by the context in which graph is this relation defined, we omit the name of the graph G and we use the symbol $\preceq$ alone. □

Definition 3.1.11 *A node v' is said to be* strictly reachable *in the graph G from another node v iff exists $p(v,v')$ and v, v' are different:*

$$v \prec_G v' \iff \exists p(v,v') \wedge v \neq v'$$

Where it is obvious by the context in which graph is this relation defined, we omit the name of the graph G and we use the symbol $\prec$ alone. □

Definition 3.1.12 *Two nodes v, v' are* connected *in the graph G iff one of them is reachable from the other:*

$$v \prec\!\succ_G v' \iff v \preceq_G v' \vee v' \preceq_G v$$

$$v \not\prec\!\not\succ_G v' \iff v \not\preceq_G v' \wedge v' \not\preceq_G v$$

Where it is obvious by the context in which graph is this relation defined, we omit the name of the graph G and we use the symbol $\prec\!\succ$ alone. □

Definition 3.1.13 *For any node $v \in V$, the* depth level *or $d(v)$ is the length of the longest path from a root to that node:*

$$d(v) = max(length(p(r,v)) : r \in R(G))$$

□

Definition 3.1.14 *A* directed acyclic graph *(DAG) is a directed graph $G = (V, E)$ with no cycle. For any node v there is no $p(v,v)$:*

$$G \in DAG \iff \forall v \in V : \nexists p(v,v)$$

□

In this dissertation we only study directed acyclic graphs. From here on, the word "graph" always refers to a DAG.

Definition 3.1.15 *A* two-terminal directed acyclic graph, *also called* standard two-terminal *or* STDAG *is a DAG such that there is only one root and only one leaf in the graph:*

$$G \in STDAG \iff G \in DAG, |R(G)| = 1, |L(G)| = 1$$

□

Proposition 3.1.16 *Properties of STDAGs:*

1. *Any node in an STDAG is reachable from the root.*
2. *The leaf of an STDAG is reachable from any node in the graph.*
3. *Any STDAG is a connected graph.*
4. *For any node $v \in V$ exists at least one full path that contains v.*

Proof: *A node $v \in V$, is the root or it has at least one predecessor. If it is not the root, take any predecessor of v and proceed by induction. Use the same rationale for successors and the leaf. The rest is trivial using the definitions.* □

Definition 3.1.17 *The* normalized STDAG $\underline{G}$ *of a DAG G is a two-terminal directed acyclic graph, constructed from G, adding at most two nodes and $O(n)$ edges to resynchronize the possible multiple roots and possible multiple leaves of G, as follows:*

Let $G = (V,E)$ be a DAG, $\underline{G} = (V',E')$:

$$\begin{array}{ll} V' = V \cup \{v_r\} & \textit{if } |R(G)| > 1 \\ V' = V \cup \{v_l\} & \textit{if } |L(G)| > 1 \\ E' = E \cup \{(v_r, v) : v \in R(G)\} & \textit{if } |R(G)| > 1 \\ E' = E \cup \{(v', v_l) : v' \in L(G)\} & \textit{if } |L(G)| > 1 \end{array}$$

□

Proposition 3.1.18 *The normalized STDAG $\underline{G}$ of any DAG G can be constructed in $O(n)$ time complexity.*

Proof: *Detecting the $R(G)$ and $L(G)$ sets implies checking only the in-degree and out-degree of every node in V. Each node appears at most once on each set. Thus, each set has $O(n)$ nodes. When the two sets are known, at most two new nodes are added, and exactly one edge per node in each set.* □

3.1.2 Transitivities

The reachability relation established by edges in the graph is transitive. Thus, we define the following concepts as in [137]:

Definition 3.1.19 *An edge in a graph $e = (v,v') \in E$ is a* transitive edge *iff there is a non-direct path between the nodes $p_{nd}(v,v')$.* □

Definition 3.1.20 *The* transitive closure *of a graph $G = (V, E)$ is another graph $G^+ = (V, E^+)$ such that E^+ contains an edge (v, v') iff exists a path $p(v, v')$ in G.* □

Definition 3.1.21 *The* transitive reduction *of a graph $G = (V, E)$ is a subgraph $G^- = (V, E^-)$, minimal under inclusion, whose transitive closure coincides with that of G.* □

Definition 3.1.22 *A* topological order *of a graph $G = (V, E)$ is any total order $\prec^t$ of V such that if $(v, v') \in E$ then $v \prec^t v'$. Each DAG has at least one topological order.* □

3.1.3 Topological graph parameters

We define the following basic graph topology parameters that we will use to characterize the graphs.

Definition 3.1.23 *We define* Maximum Degree of Parallelism *as the maximum number of nodes in a graph that are not dependent on each other:*

$$mP(G) = \max |L \in V / \not\prec\not\succ|$$

This number can be approximated by the cardinality of the biggest layer (subset of nodes with the same depth level) in the graph. We call it simply Degree of Parallelism*:*

$$P(G) = \max_i |\{v : d(v) = i\}|$$

□

Definition 3.1.24 *The* Depth *of a graph is the maximum depth level of any node in it:*

$$D(G) = \max_{v \in V} \; d(v)$$

□

Definition 3.1.25 Synchronization Density *of a graph G is the amount of edges relative to the number of nodes:*

$$S(G) = |E|/|V|$$

□

In a graph G, the S parameter (number of edges related to the number of nodes) may provide information not only about dependences, but about the overall shape of the graph. For very high sizes of $|E|$, the graph will have so many dependences that most nodes will be serialized. For very low number of edges, most nodes will be disconnected and the degree of parallelism will be higher. We may define a more topology-independent parameter to represent the overall number of edges in a graph.

Definition 3.1.26 *We define* Relative Synchronization Density *as the synchronization density relative to the number of nodes:*

$$R_s(G) = S(G)/|V|$$

Or in other words, the amount of edges relative to the square of the number of nodes. It represents the amount of edges relative to the maximum number of possible edges in a DAG with $|V|$ nodes:

$$R_s(G) = |E|/|V|^2$$

□

3.1.4 Task graphs

In this thesis we use *activity on nodes* (AoN) graphs. The nodes represent an activity and the edges a precedence order for the execution of the activities. More specifically we introduce the following definitions:

Definition 3.1.27 *For a given system, a* task *is an atomic activity which modifies the global state of the system and can be executed independently of the local state of other activities (tasks), provided a collection of preconditions. After the execution of the activity a task may produce a collection of postconditions (depending on the system state), in order to allow activation of other tasks.* □

Definition 3.1.28 *A* task graph $T = (V, E)$ *is a DAG in which a node $v \in V$ represents a task and an edge $e = (v, v') \in E$ represents the precedence relation established between two tasks when a postcondition of v is a precondition of v'.* □

Definition 3.1.29 *In the context of task graphs, the reachability property is also called* dependence. *A node v' is dependent on another node v iff $v \prec_G v'$.* □

A task graph represents a possible evolution of a system given an initial state. In the case of a parallel program, a task graph represents the dependences of the tasks generated by the program when executed with specific input data. The task graph generated by a parallel program for a given initial state (input data) is unique only if the program has no race conditions, and the evolution of the system state is independent of the scheduling of the tasks.

A task graph is some times transformed to an STDAG adding a root and a leaf that represent the starting and ending points of the whole system activity. Then, properties of STDAGs can be exploited.

Definition 3.1.30 *The* load *of a node is a positive number that represents the* cost *or* span *of executing the task in a given parameter axis. The* load distribution *of a graph is the function that maps nodes to their load values:*

$$\tau : v \in V \rightarrow \mathbb{R}^+$$

□

A typical parameter for which load is defined is time, where load represents the execution time of the activity. The total cost of a graph (the summation of all its node's load) is associated with the cost of the computation represented by the graph. The notions of path cost, and *critical path* are also defined.

Definition 3.1.31 *The* cost *or* load *of a graph G, is the sum of the loads of all its nodes:*

$$\tau(G) = \sum_{v_i \in V} \tau(v_i)$$

□

Definition 3.1.32 *The* cost *or* load *of a path, is the sum of the loads of all its nodes:*

$$\tau(p(v, v')) = \sum_{v_i \in p(v,v')} \tau(v_i)$$

□

Let us consider some usual concepts in distributed computing. In complete asynchronous communication models, the complexity of an application is related to the largest chain of messages [122]. Modifying the synchronization structures, the chains of messages are altered, and probably, also the length of the largest chain. The computation times should also be included if they are significant [122].

Application and program synchronization structures are modeled with task graphs. In our case we use AoN graphs, with nodes representing tasks or communications. Thus, the accumulated load value of the nodes in a full path represents the estimated performance time of executing this chain of nodes, with the precedence restrictions expressed by the whole graph. The maximum load of any full path, or *critical path value* (*cpv*) of the graph, represents the largest chain of communications or dependences, with computation times considered. Consequently, the *cpv* of a graph may be used as an indicator of the modeled application performance.

Definition 3.1.33 *For a given graph $G = (V, E)$ and a given load distribution, the* Critical paths *of the graph $P_c(G)$ are the full paths with maximum load. The* Critical path value *$cpv(G)$ is the load of any critical path.*

$$P_c(G) = \{p \in P_f(G) : \tau(p) = max(\tau(P_f(G))\}$$

$$cpv(G) = \tau(p) : p \in P_c(G)$$

□

3.2 Series-parallel graphs

3.2.1 Definitions

Series-parallel DAGs, their construction and their relation with general DAGs are the main focus of this chapter. We present here formal definitions and properties of this kind of graphs. The following definitions are adapted mainly from [14, 184].

SP-graphs preliminaries

The class of *edge series-parallel directed graphs* is defined recursively as follows:

Definition 3.2.1 Edge series-parallel multidigraphs (ESP)*:*

1. *A DAG with a single edge joining two nodes is ESP.*
2. *If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are ESP multidigraphs, so are the DAGs constructed by each of the following operation:*
   - Two-terminal parallel composition*: Identify the root of G_1 with the root of G_2, and the leaf of G_1 with the leaf of G_2.*
   - Two-terminal series composition*: Identify the leaf of G_1 with the root of G_2.*

□

Definition 3.2.2 Series-parallel graphs (SP-graphs)*:*
A DAG is SP iff its normalized STDAG is ESP:

$$G \in SP \Longleftrightarrow \underline{G} \in ESP$$

□

Definition 3.2.3 Non-series-parallel graphs (NSP-graphs)*:*
A DAG is NSP iff it is not in the class of SP-graphs. □

The class of SP-graphs can be characterized by not exhibiting a *forbidden subgraph*. This subgraph represents the basic topological characteristic associated with an NSP structure. We use the term *homeomorphic* to refer to graphs with similar topological features, or in other words, graphs that contains nodes with the same partial order relation. We first introduce a formal definition of the homeomorphic term to help us to characterize the relation of a graph with the forbidden subgraph.

Definition 3.2.4 *An* induced subgraph $G' = (V', E')$ *of another graph* $G = (V, E)$*, is a subgraph obtained by eliminating some nodes from* V *and eliminating from* E *the edges incident to those eliminated nodes:*

$$G \supseteq G' \Longleftrightarrow V' \subseteq V, E' = \{(u, v) \in E : u, v \in V'\}$$

$\square$

Definition 3.2.5 *A graph* $G = (V, E)$ *is* homeomorphic *to another graph* G' *iff its transitive closure does contain* G' *as an induced subgraph:*

$$G \sqsupseteq G' \Longleftrightarrow G^+ \supseteq G'$$

$\square$

Theorem 3.2.6 *A DAG is an SP-graph iff it is not homeomorphic to the W graph of Fig. 3.1; or using an equivalent characterization, iff its transitive closure does not contain the W graph of Fig. 3.1 as an induced subgraph. (See proof in [59]).* $\square$

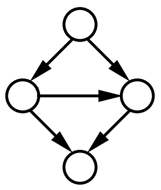


Figure 3.1: The forbidden subgraph for SP-graphs

SP graphs are a subclass of planar graphs, and also a subclass of $k-$terminal graphs (see e.g. [28]). SP graphs are equivalent to partial 2-trees, a subclass of bounded tree-width graphs (see e.g. [21, 28]). Based in the properties of these graph classes, linear time complexity algorithms to recognize SP-graphs are possible.

Proposition 3.2.7 *The recognition of a series-parallel digraph can be done in linear time. (See proof in [184, 168]).* □

Efficient parallel recognition algorithms also exist for SP-graphs and derivated classes (see [98, 61, 22, 105, 23]).

An interesting property of SP graphs, that justifies the tight complexity bounds of many algorithms for these graph class, is the bounded number of edges:

Lemma 3.2.8 *Let G=(V,E) be an SP-graph with no multiple edges. The number of edges is bounded by (see e.g. [168]):*

$$|E| \leq 2|V| - 3$$

This lemma is easily proven by induction on the SP-graphs definition. □

Lemma 3.2.9 *Let G=(V,E) be an SP-graph with no multiple edges and no transitive edges ($G = G^-$). The number of edges is bounded by*

$$|E| \leq 2(|V| - 2)$$

A proof may be found in [84]. □

SP reduction

Two operators which reduce the series or parallel structures in a graph to a single edge have been proposed [14]. The result of the use of these operators in simple graphs is shown in Fig. 3.2.

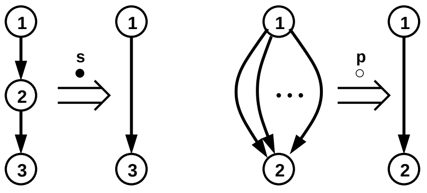


Figure 3.2: Reduction operators

Definition 3.2.10 *The* series reduction operator *or* $\overset{s}{\bullet}$*, is a mapping,* $\overset{s}{\bullet} : STDAG \times V \to STDAG$*, according to:*

$$\begin{aligned} &G\overset{s}{\bullet}v = (V', E'); \\ &\quad indeg(v) = outdeg(v) = 1, \\ &\quad E' = E \setminus \{(s,v),(v,t)\} \cup (s,t), \\ &\quad V' = V \setminus v \end{aligned}$$

□

Definition 3.2.11 *The* parallel reduction operator *or* $\overset{p}{\circ}$*, is a mapping,* $\overset{p}{\circ} : STDAG \times E \rightarrow STDAG$*, according to:*

$$\begin{aligned}
&G\overset{p}{\circ}(v, v') = (V', E');\\
&\quad |\{(v, v') \in E\}| > 1,\\
&\quad E' = E \setminus \{(v, v') \in E\} \cup (v, v')
\end{aligned}$$

□

Definition 3.2.12 *A* trivial graph *is a graph with only two nodes and one edge:*

$$G_t = (V, E);\ V = \{v, v'\}, E = \{(v, v')\}$$

□

Definition 3.2.13 *The symbol* $\vdash$ *denotes a sequence of one or more reduction operations in a graph:*

$$\begin{aligned}
&\vdash \equiv \{\overset{s}{\bullet}, \overset{p}{\circ}\}^+\\
&\vdash^{\overset{s}{\bullet}} \equiv \{\overset{s}{\bullet}\}^+\\
&\vdash^{\overset{p}{\circ}} \equiv \{\overset{p}{\circ}\}^+
\end{aligned}$$

□

Definition 3.2.14 *A* series graph *is a graph which can be reduced to a trivial graph using only series reduction operations:*

$$G \in SG \Longleftrightarrow G \vdash^{\overset{s}{\bullet}} G' \sim G_t$$

□

Definition 3.2.15 *The* minimal SP reduction graph *of G, is another graph* $[G]$ *obtained by using all possible series and parallel reduction operations in G:*

$$G \vdash [G]; \nexists G', [G] \vdash G'$$

□

Proposition 3.2.16 *A graph G is an SP-graph iff its normalized STDAG, can be reduced to a trivial graph by series and parallel reduction operations.*

$$G \in SP \Longleftrightarrow [\underline{G}] \sim G_t$$

This result is easily proven by induction on the ESP and reduction operation definitions. □

3.2.2 Distance from NSP to SP graphs

In this section we present formal methods to define and measure the *distance* from an NSP to an SP approximation graph. These definitions motivate some transformation techniques and a distance concept to be used later to try to measure the impact of NSP to SP transformations.

We can measure the distance from an NSP to an SP form by the number of induced forbidden subgraphs it has. This distance has shown to be a very important parameter of a graph. Many graph analysis problems that show to be feasible when the graph is bounded to an SP form, are NP-hard to solve in a generic NSP graph. Nevertheless, it is possible to derive algorithms that are exponential in the distance from the graph to an SP form, instead of in the number of nodes [14].

Node reduction and complexity

The number of forbidden subgraphs in a graph G can be algorithmically measured by *reductions* or *path expressions* [14, 143]. The reduction system uses series and parallel reductions to eliminate the parts of the graph that are already SP. After that, only nodes and edges associated with forbidden subgraphs remain. To eliminate one node and its associated forbidden subgraph, a new operator called *node reduction operator* is introduced. It operates on a node that is connecting *one to many* or *many to one* nodes. In the first situation, it substitutes a node with only one predecessor for a collection of edges between its only one predecessor and its successors. In the second situation it substitutes a node with only one successor for a collection of edges between its predecessors and its only one successor. The effect of a node reduction in both cases ($indeg(v) = 1$ and $outdeg(v) = 1$), is shown in Fig. 3.3.

Definition 3.2.17 *The* node reduction operator *or* $\overset{n}{\star}$*, is a mapping,* $\overset{n}{\star} : STDAG \times V \to STDAG$*, according to:*

$$G\overset{n}{\star}v = (V', E'); indeg(v) = 1 \vee outdeg(v) = 1,$$

$$\begin{aligned}
&\text{If} \quad indeg(v) = 1,\\
&\qquad E' = E \setminus \{(s,v),(v,t_i) : t_i \in Succ(v)\} \cup \{(s,t_i) : t_i \in Succ(v)\}\\
&\qquad V' = V \setminus \{v\}\\
&\\
&\text{If} \quad outdeg(v) = 1,\\
&\qquad E' = E \setminus \{(v,t),(s_i,v) : s_i \in Pred(v)\} \cup \{(s_i,t) : s_i \in Pred(v)\}\\
&\qquad V' = V \setminus \{v\}
\end{aligned}$$

□

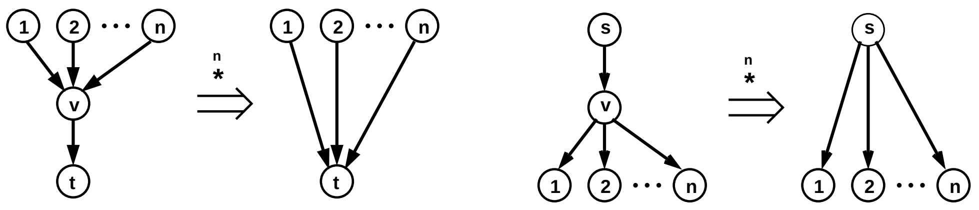


Figure 3.3: Node reduction operator

After all possible series-parallel reductions are applied all nodes except the root and the leaf are associated with a forbidden subgraph. Any one can be chosen for elimination. At least the children of the root have $indeg(v) = 1$, and at least the parents of the leaf have $outdeg(v) = 1$. Thus, there are always nodes that can be *node reduced.* After applying a node reduction, new series and parallel reductions are usually possible. They should be applied before new node reductions.

Duplication of nodes

Although previous works which present the node reduction do not rationale it, this operator is intrinsically related to an NSP to SP transformation based on the duplication of nodes, also discussed in section 3.3.1. See Fig. 3.4. The node reduction operation intrinsically creates multiple instances of the node that is being reduced. A different path from/to the unique parent/child is constructed through any of the multiple copies. The duplicated nodes are inherently reduced by serial reduction. Thus, the node reduction does not add new dependences to the graph, and the non-SP conflict (the forbidden subgraph) disappears.

We may define a *distance* from any graph G to an SP form based on the *reduction complexity* of G:

Definition 3.2.18 *The* reduction complexity *of a graph G, denoted by $\mu(G)$, is the minimal number of node reductions sufficient to reduce G to a trivial graph.*

$$\mu(G) = min(c); [...[[[G]\overset{n}{\star}v_1]\overset{n}{\star}v_2]...\overset{n}{\star}v_c] \sim G_t$$

□

Definition 3.2.19 *The sequence of $\mu(G)$ nodes $(v_1, v_2, ..., v_c)$ that reduce the graph G to a trivial graph is called the* reduction sequence*.* □

As was shown by Bein, Kamburowsky and Stallman in [14], it is possible to compute $\mu(G)$ and the reduction sequence in polynomial time complexity.

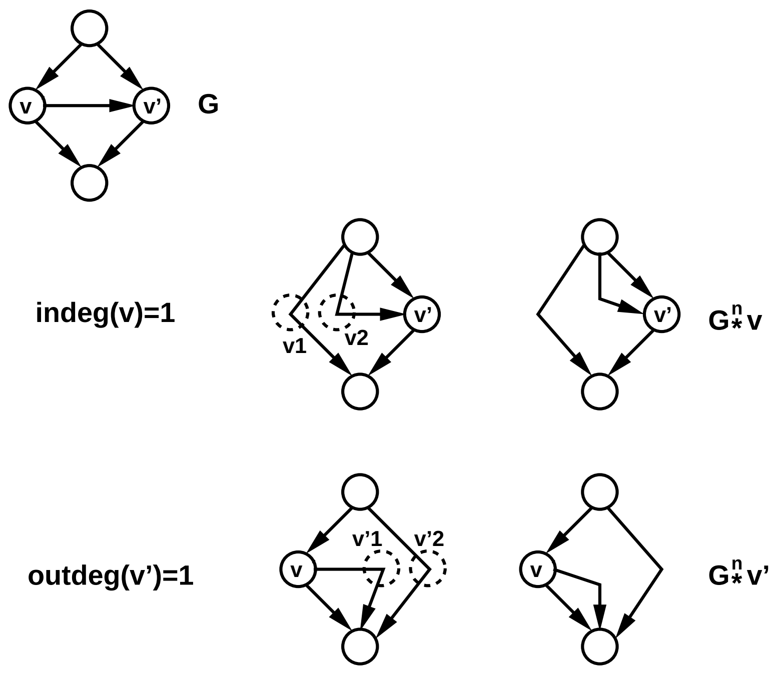


Figure 3.4: Intrinsic operations in node reduction

At the same time the maximum distance of a graph to an SP form (reduction complexity) is limited by the number of nodes:

$$\mu(G) \leq n - 3$$

3.3 Transformation problem (NSP to SP)

In this section we investigate the foundation of full transformation methods to approximate the structural differences between NSP and SP graphs. The usual asymptotic notation is used for complexity bounds throughout the following sections. We use O, and Θ notation as defined in [13].

3.3.1 SP-ization

We are interested in methods to approximate NSP graphs to an SP form that both: (1) keeps the dependences information of the original graph; and (2) minimizes the potential parallelism loss. Different approaches are possible:

Duplication of nodes: As shown in section 3.2.2, a method to transform an NSP graph into an SP form by the duplication of nodes is possible. The main interest of this transformation is that it does not add dependences to the task graph, and it produces no potential parallelism loss. However, the duplication of tasks increases the total cost of a computation (it is not a work-preserving transformation). In specific circumstances it can trade communication costs for computation and memory costs. Duplication of tasks (in other processors) increments the total computation and resources cost, but it may lead to a higher locality degree, reducing the number of synchronization or communication operations among processors that execute the duplicated tasks. Task duplication is known to have a favorable effect on minimizing the total execution time in distributed systems scheduling (see e.g. [158]).

The cost increase is determined by the number of node duplications. Taking into account that every node reduction duplicates the node a number of times equal to the number of incident edges minus one, the number of node duplications can be $O(m)$! In cases of small degree of parallelism, very low $\mu(G)$ and specific topologies where the nodes to reduce have a very small *indeg*, *outdeg*, the benefits obtained may compensate the global cost increase. Let $\tau(G)$ be the total cost of a computation represented by G. Let G' be the SP version of G produced by duplicating nodes with any reduction sequence. Then, if $\max_{v \in V}(indeg(v), outdeg(v)) = k$, the following result can be derived:

$$\tau(G') \leq k\tau(G)$$

Also, in the case where all nodes in G have the same load, $\forall v \in V, \tau(v) = c$, we can exert the result:

$$\tau(G') \leq ck\mu(G) + \tau(G)$$

Although a linear time algorithm for detecting the shorter reduction sequence exists [14], it does not assure that the nodes with less incident edges are the ones selected. The problem of selecting a reduction sequence which minimizes the edges affected (node duplications) is, as far as we know, not studied.

Another problem with this approach is that we are only considering the CS problem. However, if the PPM supports mutual exclusion mechanisms, the nodes to duplicate may need to contend with others for execution privilege. The duplication of a node involved in a mutual exclusive operation can increase the critical path, as the copies of a duplicated node cannot be executed in parallel, leading to more contention and more synchronization costs. Indeed, most of the time, duplication only minimizes execution time if additional (CPU) resources are available. Moreover, a task that uses a previously non-shared resource cannot be safely duplicated without modification; duplicated operations on the resource may lead to a correctness fault.

For general parallel computing, especially in massive parallel computing or applications with many inter-task dependences the total cost increase can easily be unaffordable. The applicability scope of this technique is narrow.

Adding dependences: The alternative mechanism to transform an NSP graph to SP form without duplicating nodes and without increasing the total computation cost is adding new dependences. These work-preserving techniques are not directly based on reduction sequences, and the number of topology modifications may be not related to $\mu(G)$. Indeed, graphs with higher $\mu(G)$ may need less added dependences to be transformed to SP. We study in section 3.6.2 an algorithmic metric of the impact, in a given graph, of a given technique based on adding dependences. The main drawbacks of these techniques are that: (1) they serialize previous potentially parallel tasks, and (2) the selection of dependences to add is guided by heuristics which should make assumptions about the task workloads, in order to minimize the potential impact of the task serialization.

Mixed techniques: Mixed techniques that mainly add dependences but strategically select a small subset of nodes to duplicate could be interesting. However, no convenient one has yet been proposed. A good starting point to devise such techniques will be: (1) the methods based on adding dependences studied in this thesis, and (2) the works about reducing expensive communication costs by computation redundancy, or scheduling with redundancy in UTC (Unit Time Cost) graphs [24, 60, 141].

In this work we study new methods and heuristics to transform NSP to SP graphs by adding dependences, trying to minimize the potential loss of parallelism introduced by them. We denote such transformation methods as *SP-izations*.

Definition 3.3.1 *An* SP-ization *is a graph transformation technique T which transforms any generic STDAG into an SP form, keeping the same nodes and dependences as in the original graph, and possibly adding new zero loaded nodes (resynchronization points) and edges (dependences).*

$$\begin{aligned} &T : STDAG \longrightarrow SP;\\ &\quad T(G) = (V', E'),\\ &\quad V \subseteq V', V' \setminus V = \{w; \tau(w) = 0\}\\ &\quad \forall u, v \in V, u \preceq_G v \Longrightarrow u \preceq_{G'} v \end{aligned}$$

□

3.3.2 Local resynchronization

Several SP-ization techniques may be proposed. We will focus first in the approximation of graphs containing in its transitive closure only one basic NSP problem, or in other words, only one instance of the forbidden subgraph presented in theorem 3.2.6. Then, more elaborated techniques for complex NSP problems (combinations of several instances of the forbidden subgraph) will be studied.

In the following examples and figures we will not present full NSP graphs, but only the induced subgraph which contains nodes related to the NSP problem we want to illustrate. Thus, every edge in the example graphs may represent a full SP-reducible subgraph of the original graph, and the propagated dependence is not eliminable by a transitive reduction. We name these edges as *SP branches.*

Definition 3.3.2 *The* SP branches *of a graph G are the subgraphs $S \subseteq G$ that are themselves SP graphs, $S \in SP$.* □

Consider for example the graphs in Fig. 3.5. The graphs on the right side represent the forbidden induced subgraphs found in the transitive closure of the left side graphs. The light-grey edges represent SP branches of the original graph. Thus, the original left-side graphs are homeomorphic to the forbidden subgraph, and the transformation solutions presented below can be applied to both of them.

We present three different methods to resynchronize the forbidden subgraph. The first two methods can be applied in two different forms. The final five transformations are illustrated in Fig. 3.6. Any of them can be used to eliminate an isolated NSP problem. The four nodes related to the forbidden subgraph are named s, v, v', t, accordingly to their role to simplify the references in the text.

Up synchronization: An SP branch is resynchronized changing the leaf of the branch for an ancestor of the original leaf. This transformation can be applied to two different SP branches related to the forbidden subgraph (v, t) or (s, v').

- G', resynchronizing (v, t): New dependences are created from the nodes in the SP branch represented by (v, t) to v' and, thus, to nodes in the SP branch represented by (v', t). New dependences added are defined by:
$$\{w : v \prec w \prec t\} \prec \{w' : v' \preceq w' \prec t\}$$
- G'', resynchronizing (s, v'): New dependences are created from the nodes in the SP branch represented by (s, v') to v and, thus, to nodes

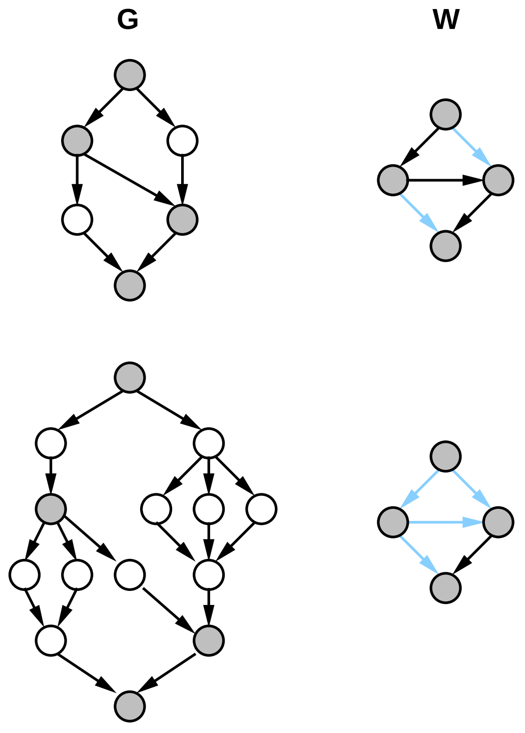


Figure 3.5: Example graphs homeomorphic to the W forbidden subgraph

in the SP branches represented by (v,t) and (v,v'). New dependences added are defined by:

$$\{w : s \prec w \prec v'\} \prec \left\{ \begin{array}{l} \{w' : v \preceq w' \prec v'\} \\ \{w' : v \preceq w' \prec t\} \end{array} \right.$$

Down synchronization: An SP branch is resynchronized changing the root of the branch for a descendant of the original root. This transformation can be also applied to two different SP branches related to the forbidden subgraph (v,t) or (s,v').

- G', resynchronizing (s,v'): New dependences are created from the nodes in the SP branch represented by (s,v) to nodes in the SP branch represented by (s,v'). New dependences added are defined by:

$$\{w : s \prec w \preceq v\} \prec \{w' : s \prec w' \prec v'\}$$

- G'', resynchronizing (v,t): New dependences are created from the nodes in the SP branches represented by (s,v') and (v,v') to nodes

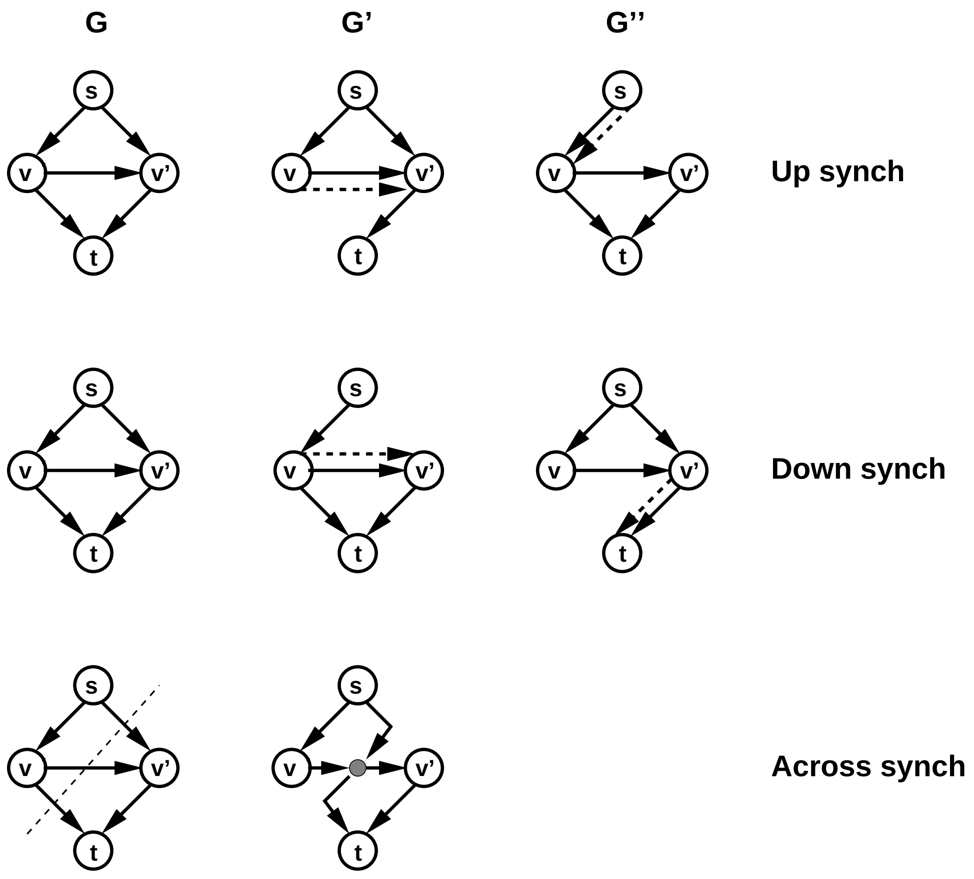


Figure 3.6: Methods for resynchronization of graphs homeomorphic to W

in the SP branch represented by (v, t). New dependences added are defined by:

$$\left.\begin{array}{l}\{w : s \prec w \preceq v'\} \\ \{w : v \prec w \preceq v'\}\end{array}\right\} \prec \{w' : v \prec w' \prec t\}$$

Across synchronization: In this third more general transformation, the three SP branches $(s, v'), (v, v'), (v, t)$ are splited in two parts. The first part of the three branches is resynchronized over a new zero loaded node. Edges from this node to the second parts of the three branches are added to connect the graph. Let S_1, S_2, S_3 be the subgraphs corresponding to the three SP branches:

$$\begin{array}{l}S_1 \subset G : S_1 \in SP; R(S_1) = \{s\}; L(S_1) = \{v'\} \\ S_2 \subset G : S_2 \in SP; R(S_2) = \{v\}; L(S_2) = \{v'\} \\ S_3 \subset G : S_3 \in SP; R(S_3) = \{v\}; L(S_3) = \{t\}\end{array}$$

Let A_1, A_2, A_3 be the sets of nodes in the first part of the three branches, and B_1, B_2, B_3 be the sets of nodes in the second part of the three branches. The first part will contain at least the root of the branch, and the second part will contain at least the leaf of the branch:

$$A_i, B_i \subset S_i : A_i \cup B_i = S_i; A_i \cap B_i = \emptyset; \forall w' \in B_i, w' \not\preceq w \in A_i$$

The transformation works adding the node and dependences defined by:

$$V = V \cup \{c\}$$

$$w \in A_i \prec c \prec w' \in B_i$$

It is possible to eliminate any NSP problem (or combinations of them) by applying several up/down synchronizations in order to eliminate local problems. With no information about the workload of the implicated nodes it is not possible to decide when up or down synchronization may incur in a higher penalty in the critical path. On general graphs, the up/down synchronization may serialize big subgraphs with high probabilities of many added dependences.

The across synchronization can be applied in only one way in the context of the basic NSP problem or forbidden subgraph elimination. However, when the edges represent non-empty SP branches, we must propose a rule or strategy to decide which nodes will be in the first and second parts of the branch. In Fig. 3.7 we show an example of two different strategies for cutting subgraphs during an across synchronization (dotted edges represent original graph edges which degenerate in transitivities, and can be eliminated). The decision relies again in the information we have about the workload of the nodes in these subgraphs. If properly applied, across synchronization may derive in lesser amount of added dependences compared with up/down synchronization, especially when applied to combined NSP problems, as the ones presented in next section.

3.3.3 Combinations of NSP problems

When a graph presents several NSP problems, the induced forbidden subgraphs may be composed. (In [14] three composed forbidden subgraphs are studied to decide which nodes must be chosen to minimize the reduction sequence. Some of those graphs are somehow related or inspiration for our resynchronization solutions).

We present here different compositions of the basic NSP problem suitable to be resynchronized with the three previous methods. Further combinations of these compositions may reproduce any NSP graph topology.

To simplify the mathematical notation of precedences, for the following descriptions we assume there exists a source and a target node $s, t \in V$ that are

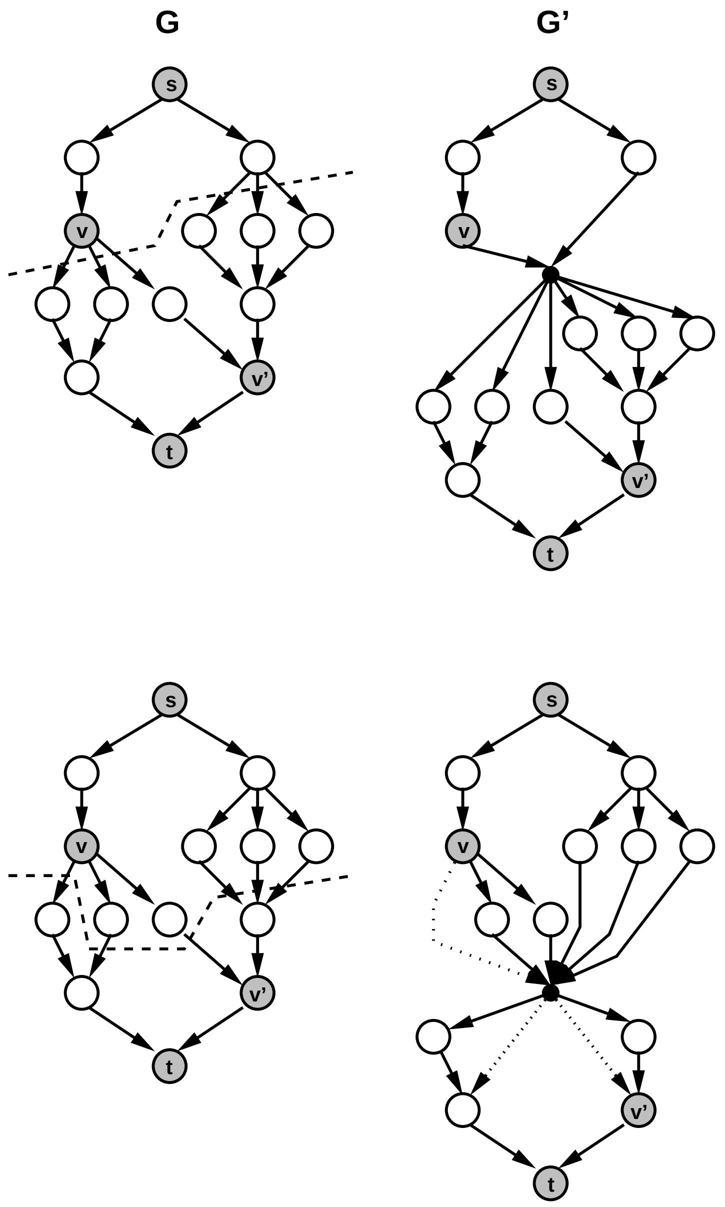


Figure 3.7: Example of different strategies for across synchronization

respectively before and after all nodes related to the NSP problems in the combination (denoted by $W \subset V$):

$$s \prec \{w : w \in W\}$$

$$\{w : w \in W\} \prec t$$

Series NSP composition: There exists two similar problems, characterized by a series composition of several v or v' nodes.

$$\begin{aligned} \text{Problem1}: \quad & W = \{v, v'_1, v'_2\}; \quad v \prec v'_1 \prec v'_2; v \prec_{G\setminus\{v'_1\}} v'_2 \\ \text{Problem2}: \quad & W = \{v_1, v_2, v'\}; \quad v_1 \prec v_2 \prec v'; v_1 \prec_{G\setminus\{v_2\}} v' \end{aligned}$$

This combination can be eliminated by several up/down synchronizations. Both problems can also be eliminated by a combined *across synchronization*. See in Fig. 3.8 an example of each type of transformation where G' and G'' represent solutions with up/down synchronizations respectively, and G''' the across synchronization solution.

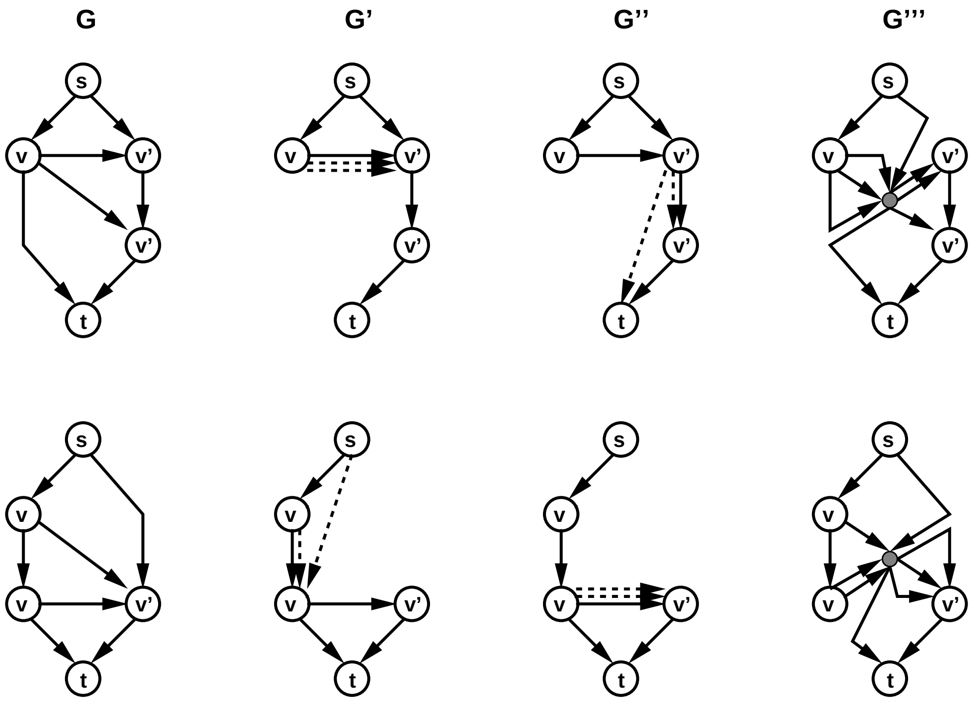


Figure 3.8: Resynchronization of graphs homeomorphic to Series-NSP

Parallel NSP composition: There exists two similar problems, characterized by a parallel composition of several non-dependent v or v' nodes.

$$\text{Problem1}: \quad W = \{v, v_1', v_2'\}; \quad v \prec v_1'; v \prec v_2'; v_1' \not\prec\not\succ v_2'$$

$$\text{Problem2}: \quad W = \{v_1, v_2, v'\}; \quad v_1 \prec v'; v_2 \prec v'; v_1 \not\prec\not\succ v_2$$

The composition can be eliminated by several up/down synchronizations. Both problems can also be eliminated by a combined *across synchronization*. See in Fig. 3.9 an example of each transformation where G' and G'' represent solutions with up/down synchronizations respectively, and G''' the across synchronization solution.

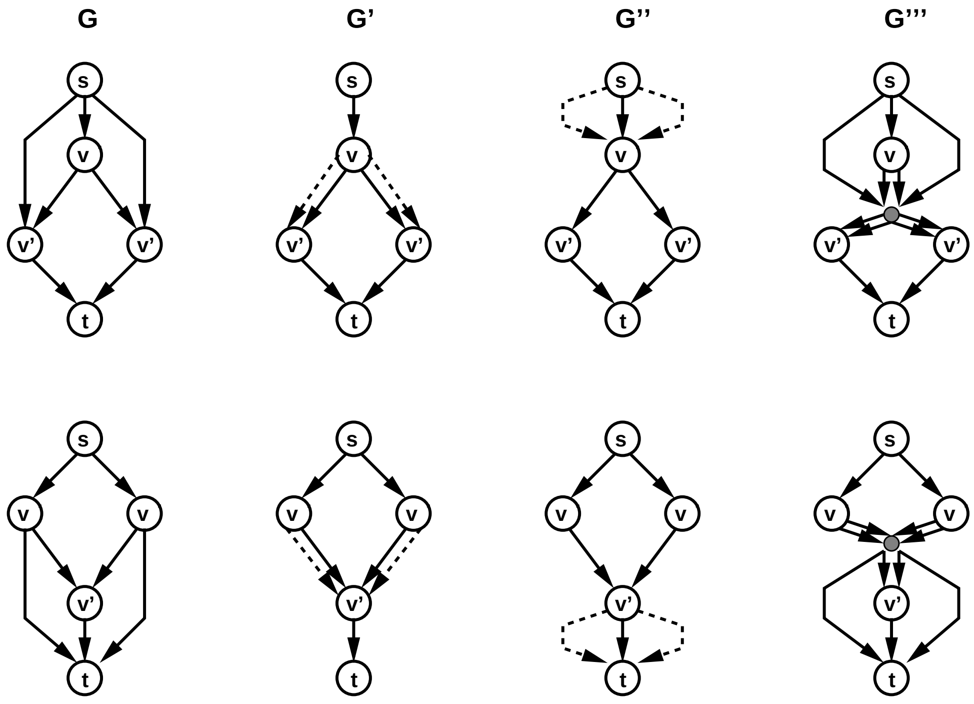


Figure 3.9: Resynchronization of graphs homeomorphic to Parallel-NSP

Chain NSP composition: An NSP problem is chained with another NSP problem when the v' node of the first of them is inserted between the v and t nodes of the second.

$$W = \{v_1, v_2, v_1', v_2'\}; \; v_1 \prec v_1'; v_2 \prec v_1'; v_2 \prec v_2'; v_1 \not\prec\not\succ v_2; v_1' \not\prec\not\succ v_2'$$

Several problems may be consecutively chained. Such a chain of NSP problems can be eliminated by a full *across synchronization*. See an example of two chained NSP problems and their resynchronization with across synchronization in Fig. 3.10.

Up/down synchronizations may be also used to eliminate a chain of NSP problems. However, it is a complicate operation that must be done in several phases, each of them with several choices for up/down synchronization. For example, a chain of two NSP problems has three NSP problems. The two original ones and the problem originated by the chain composition. We must eliminate first the local problems (each of them with up or down synchronization), before the problem originated by the chain is exposed and can be eliminated itself (with two up or two down synchronization possibilities). Apart from the amount of choices, other problem associated with this up/down synchronizations is that the chained problems, will be completely serialized, probably loosing a big amount of the original parallelism.

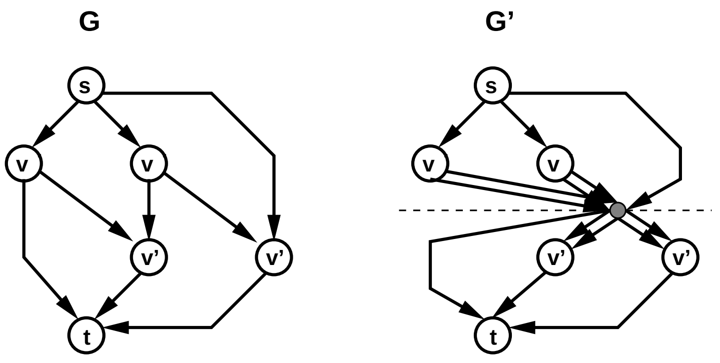


Figure 3.10: Resynchronization of graphs homeomorphic to 2 Chained-NSP

Crossed NSP composition: Two NSP problems are *crossed* when both v' nodes are inserted between the v and t nodes of the other problem. Multiple NSP problems may be crossed with one or several of the others to form multiple crossing NPS compositions.

$$W = \{v_1, v_2, v_1', v_2'\};\ v_1 \prec v_1', v_2'; v_2 \prec v_1', v_2'; v_1 \not\prec\not\succ v_2; v_1' \not\prec\not\succ v_2'$$

A collection of crossed NSP problems can be solve with *across synchronization.* See an example of this resynchronization on a crossed composition of two NSP problems in Fig. 3.11.

As in the chain problem, many choices for up/down synchronizations exist, but finally they serialize all the v nodes, and all the v' nodes implicated in

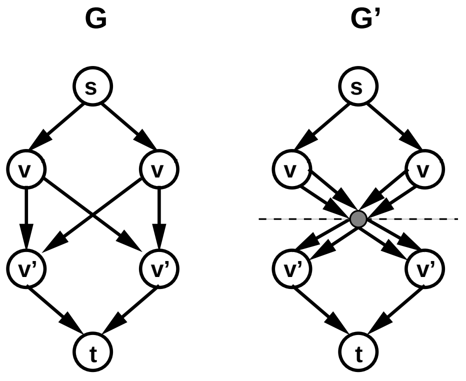


Figure 3.11: Resynchronization of graphs homeomorphic to 2 Crossed-NSP

the crossing, leading to a high loss of the original parallelism due to added dependences.

For simple combinations of NSP problems (series or parallel NSP problems) any technique may be appropriate, and no clear clues about which one to choose can be given without workload information or full topology inspection. For complex combinations (chain and crossed NSP problems) we detect that if up/down synchronizations are used, the serialization of nodes increases with the number of local NSP problems implicated. For these problems it seems more appropriate to try the across synchronization method. However, the cutting strategy must be carefully selected, as it could have an important impact on the results.

3.3.4 Simple SP-ization techniques

We present here two simple graph transformations that correspond to SP-ization techniques. They introduce the idea of SP-ization and motivate the presentation of our complex algorithms in the following sections.

Technique 1: Serialization

This first technique is a trivial example of what an SP-ization can be, but useless for practical purposes. It consists in a full serialization of the graph nodes, transforming the partial order defined by the graph in any total order that honors the original partial order. The result is a series graph, that is also SP.

Definition 3.3.3 *Let $\prec^t$ be any topological order of $G = (V, E)$. Then, an SP-ization T_s can be defined by:*

$$T_s(G) = (V, E');$$
$$E' = \{(v, w) : v \prec^t w; \nexists c, v \prec^t c \prec^t w\}$$

□

An example of the application of this technique is shown in Fig. 3.12. With this transformation most of the information provided by the original dependences is lost, and so many new dependences are added that all parallelism expressed in the original graph disappears.

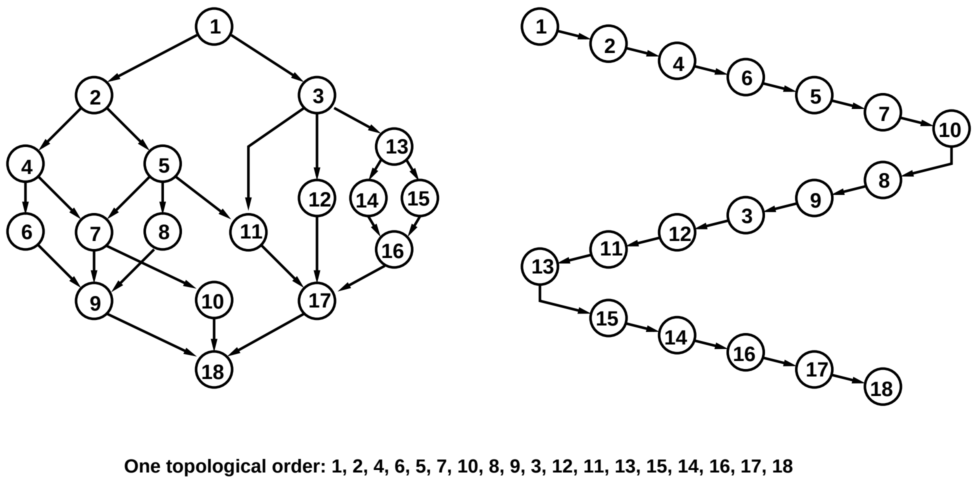

One topological order: 1, 2, 4, 6, 5, 7, 10, 8, 9, 3, 12, 11, 13, 15, 14, 16, 17, 18

Figure 3.12: Technique 1 - Serialization based on topological order

Technique 2: WSSynch (Simple layering synchronization)

The second technique consists in a full barrier synchronization of node *layers*. For this technique the information provided by the graph dependences is used only to determine the layers with a *wide or breadth first search* of the graph. Thus, the name *WSSynch (Wide first Search Synchronization)*. The wide first search of a graph visits the nodes in *level* or *depth* order. Each layer contains the nodes with the same depth.

Definition 3.3.4 *Let $G = (V, E)$ be a graph, with a maximum depth level $k = D(G)$. The* Wide first search layering *$\mathcal{L}_{WS}(G)$ is a partition of the graph nodes according to:*

$$\mathcal{L}_{WS}(G) = \mathcal{P}G = \{l_1, l_2, ..., l_k\}; \quad l_i = \{v : d(v) = i\}, i = 1...k$$

□

Based in the previous layering definition, we define the *WSSynch* or *layering* transformation as follows. An example of the application of this technique is shown in Fig. 3.13.

Definition 3.3.5 *Let $G = (V, E)$ be a graph, then an SP-ization T_p can be defined by:*

$$\begin{aligned} &T_p(G) = (V', E'); \\ &\quad \mathcal{L}_{WS}(G) = \{l_1, l_2, ...l_k\} \\ &\quad V' = V \cup \{b_1, b_2, ..., b_{k-1}\} \\ &\quad E' = \{(v, b_i) : v \in l_i\} \cup \{(b_i, v') : v' \in l_{i+1}\} \end{aligned}$$

□

The technique does not exploit the possible short distance of the graph to an SP form. The number of dependences added can be really high for graphs with low reduction complexity $\mu(G)$. Specifically it destroys the SP subgraphs of G that could be preserved.

The advantage of this technique is that there exist fast algorithms with low time complexity bounds $O(max(n, m))$ to compute the level of the nodes and the layering (see e.g. [29]). Moreover, the result is the only possible SP-ization for many specific regular structures related to common applications (see 4.2). In fact this technique has been previously exploited with modeling techniques for scalability and performance analysis of common parallel structures [130]. At the same time it can be used to trivially map such structures to the BSP model of computation.

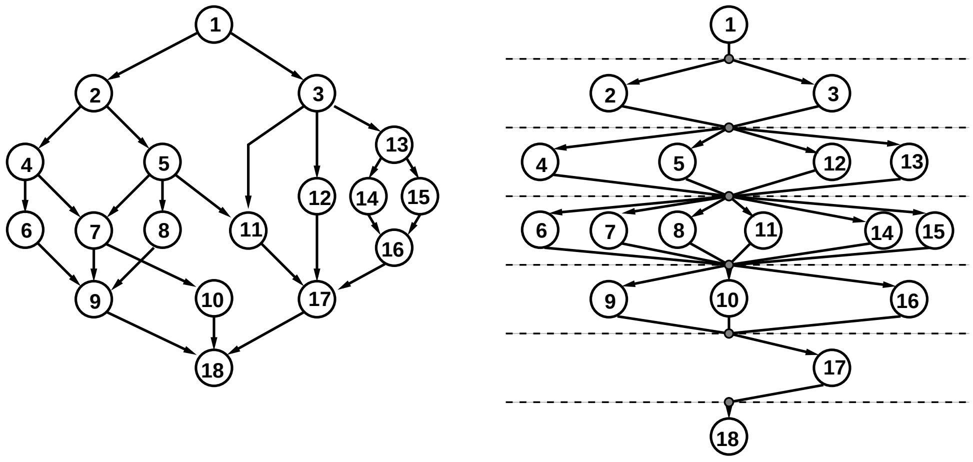


Figure 3.13: Technique 2 - Full barrier synchronization based on layering

Other layering techniques not based in wide first search (or node level) are possible, and for some graphs the increase of the number of dependences can be lesser. However they are specific for the graph topology.

3.4 Algorithm 1: Local exploration

We present here our first full SP-ization algorithm, introduced in [83] and fully explained in [82]. It was the result of our introductory exercises to the NSP to SP transformations, and it was quickly superseded by our second algorithm presented in section 3.5. Thus, a formal proof of correctness was never devised. Instead, an implementation was heuristically tested with about twenty thousand random graphs with up to hundreds of nodes. Its main interest is the local strategy used. The algorithm searches for the less complex or more local NSP problem combinations, to solve them before continue in an inside-outside style. The technique uses a mixed approach of *up* and *across* synchronizations. The core of the algorithm is the search technique that identifies the nodes related to a local NSP problem. Depending on the input order (node labeling) the solutions can be different. In this section we will present some new notations used in the algorithm, a formal definition of the transformation, and a full explanation of the algorithm strategy with an example.

3.4.1 Notations

Definition 3.4.1 *We classify nodes in three broad categories in terms of their synchronization role in the graph:*

$$Fnodes(G) = \{v \in V : |Succ(v)| > 1\}$$

$$Jnodes(G) = \{v \in V : |Pred(v)| > 1\}$$

$$JFnodes(G) = Fnodes \cap Jnodes$$

□

The algorithm gathers information of NSP problems and their composition from a minimal SP reduction ($[G]$) of the original graph. During the computation of $[G]$, several series or parallel reduction operations are applied to reduce SP branches to a single edge (see section 3.2.1). We use an annotation system to keep track of the ending edges of an SP branch, to modify them in the resynchronization phase.

Definition 3.4.2 *We associate a* Set of edges *Z to any edge in the graph:*

$$Z : E \mapsto 2^E$$

After a series reduction operation, let e be the new edge introduced by $G\overset{s}{\bullet}v$, and $(s,v),(v,t)$ the edges that disappear, then

$$Z(e) = Z(e'); e' = (v,t)$$

After a parallel reduction operation, let e be the new edge introduced by $G\overset{p}{\circ}(v,v')$ then

$$Z(e) = \cup\{Z(e') : e' = (v,v')\}$$

□

3.4.2 SP-ization technique

Here, we define formally the transformation applied by the algorithm. It is based on the application of a resynchronization operation on a collection of nodes related to a combination of NSP problems. The strategy we propose now works properly when the input is the transitive reduction G^- of the graph to be transformed. Edges which represent transitive dependences confuses the algorithm and makes it serialize unnecessary nodes. Thus, a previous phase must compute the transitive reduction of G.

We define the *handles* (F,J) of an NSP problem, as a pair of node sets with the properties to be defined below. The F and J sets will contain the v and v' nodes related to an NSP problem combination, which is suitable to be solve by one across synchronization. We present first the properties of these sets, and then the search strategy to find them.

Definition 3.4.3 *Let $G = (V,E)$ be an STDAG, and $[G] = (V_R, E_R)$ its minimal SP reduction. (F,J) is a pair of node sets ($F \subset V_R, J \subset V_R$), called* handles *with the properties: (a) All nodes in F are connected with at least one node in J, and all nodes in J are connected with at least one node in F; (b) all nodes in F have all their successors in J, except successors that are descendents of other nodes in J, and successors that are also in F; (c) all nodes in J have all their predecessors in F, except predecessors with an ancestor in J.*

Let us denote the nodes in (F,J) sets with $f \in F; j,j' \in J$. Then,

$$\begin{aligned}
&J \subseteq \mathit{Jnodes}([G]), F \subseteq \mathit{Fnodes}([G]);\\
&\quad \forall f, \exists j : (f,j) \in E_R,\\
&\quad \forall j, \exists f : (f,j) \in E_R,\\
&\quad \forall f, \forall t \in \mathit{Succ}(f); t \in J \vee \exists j', j' \preceq t\\
&\quad \forall j, \forall s \in \mathit{Pred}(j); s \in F \vee \exists j', j' \preceq s
\end{aligned}$$

□

The search strategy to find a pair of (F, J) sets can be describe as follows (all the graph operations are related to the minimal reduction graph $[G]$):

1. Select an initial F node related to an NSP problem. The only condition is that it must have a successor that is a J node.

$$f_0 \in \mathit{Fnodes}([G]) : \exists j \in \mathit{Succ}(f), j \in \mathit{Jnodes}([G])$$

2. Create empty set pairs. One for exploration (F', J') and one for final nodes (F, J). Put f_0 in the initial F' set:

$$F' = \{f_0\}, J' = \{\}, F = \{\}, J = \{\}$$

3. **DO UNTIL** $F' = \{\}$

   (a) Locate successors of F' nodes. The new J' set has those successors which are not in J, and are not dependent on other J nodes:

$$J' = \{j' \in \mathit{Succ}(f' \in F') : \nexists j \in J, j \prec j'\}$$

   (b) Eliminate J and J' nodes dependent on other new J' nodes:

$$J = J \setminus \{j : \exists j' \in J', j' \prec j\}$$

$$J' = J' \setminus \{j' : \exists j'' \in J', j'' \prec j'\}$$

   (c) Move explored F' nodes to F:

$$F = F \cup F'$$

   (d) Eliminate F nodes which has no more successors in J and J' due to elimination:

$$F = F \setminus \{f : (J \cup J') \cap \mathit{Succ}(f) = \emptyset\}$$

   (e) Locate predecessors of J' nodes. The new F' set has those predecessors that are not in F and are not dependent on any J node:

$$F' = \{f' \in \mathit{Pred}(j' \in J') : f' \notin F; \nexists j \in J, j \preceq f'\}$$

   (f) Move explored J' nodes to J:

$$J = J \cup J'$$

At the end of this procedure, the (F, J) sets have the properties defined previously. We define now a resynchronization operator that modifies G such that the collection of $[G]$ edges with its source node in F and its target node in J are substituted for: (1) a new synchronization node, and (2) a collection of edges from the nodes in F to the new node, and from the new node to the nodes in J.

Definition 3.4.4 *Let $G = (V, E)$ be an STDAG, and $[G] = (V_R, E_R)$ its minimal SP reduction. For a given pair of node sets (F, J), let $A = \{(f, j) \in E_R : f \in F, j \in J\}$ be the set of edges with the source in F and the target in J. We define the resynchronization operator $\triangleright$ as follows:*

$$
\begin{aligned}
G \triangleright (F, J) &= (V', E'); \\
&V' = V \cup \{r\} \\
&E' = E \setminus \{Z(e) : e \in A\} \\
&\cup \{(s, r) : (s, t) \in Z(e), e \in A\} \\
&\cup \{(r, t) : t \in J\}
\end{aligned}
$$

□

To improve the SP compositional looking of the result, we may synchronize the branches of any node in the F set with its own dummy synchronization point, and then, synchronize all dummy nodes over the general resynchronization point. This similar, although more complex, resynchronization operator may be defined as follows.

Definition 3.4.5 *Let $G = (V, E)$ be an STDAG, and $[G] = (V_R, E_R)$ its minimal SP reduction. For a given pair of node sets $(F = \{f_1, f_2, ..., f_n\}, J)$, let $A = \{(f, j) \in E_R : f \in F, j \in J\}$ be the set of edges with the source in F and the target in J. We define the resynchronization operator $\triangleright$ as follows:*

$$
\begin{aligned}
G \triangleright (F, J) &= (V', E'); \\
&V' = V \cup \{r_i : i = 0, ..., n\} \\
&E' = E \setminus \{Z(e) : e \in A\} \\
&\cup \{(r_i, r_0) : i = 1, 2, ..., n\} \\
&\cup \{(s, r_i) : (s, t) \in Z((f_i, j) \in A)\} \\
&\cup \{(r_0, j) : j \in J\}
\end{aligned}
$$

□

Definition 3.4.6 *Let $G = (V, E)$ be a graph, then an SP-ization T_{Alg1} can be defined by the recursive application of a resynchronization operator $\triangleright$ for any (F, J) sets until the result is an SP graph:*

$$T_{Alg1}(G) = (\ (((G \triangleright (F, J)) \triangleright (F', J')) \ ... \ \triangleright (F^n, J^n)\ ) \in SP$$

□

This strategy leads to some troubles in special situations that must be considered. They are discussed in the following sections.

3.4.3 JF combinations

A JF combination is a topological feature of a graph characterized for dependence relations described as follows (See an example in Fig. 3.14):

$$\begin{gathered} JFcombination = (f, j); f \in Fnodes(G), j \in Jnodes(G) : \\ \exists f', j'; f' \in Fnodes(G), j' \in Jnodes(G); \\ f' \prec j \preceq f \prec j' \end{gathered}$$

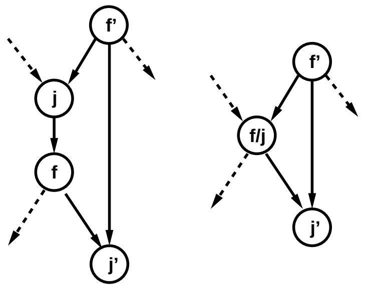


Figure 3.14: Example of JF combinations

In a JF combination, the relation $j \preceq f$ implies that all j' nodes such that $f \prec j'$, will be erased from J set because of transitive relation with other nodes in the J set ($j \preceq f \prec j'$). Consequently, f will have no successors in the J set and it will be also erased from the F set. The only nodes in handles sets will be finally f' and j'.

However, detecting a JF combination as soon as possible could avoid some f or j recursive exploration from nodes that we know they are going to disappear from the set. Or even we can mix two different NSP problems as we explain in the following section.

3.4.4 Mixing problems through JF combinations

Sometimes, the topology of the graph presents a JF combination in which the f node has two j nodes associated to different NSP problems. See Fig. 3.15. Depending on the order in which the nodes are explored and introduced in the

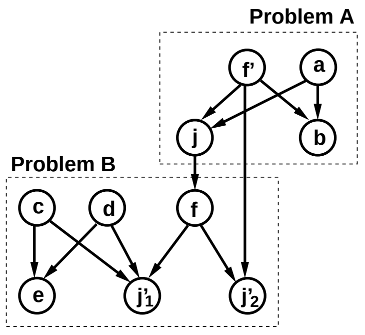


Figure 3.15: Example of mixed problems through a JF combination

sets, we can add J nodes and F nodes from the different problems in the exploring sets, before detecting and eliminating the JF combination. In the Fig. 3.15 example, if node f is added to F set and is explored, j'_1, j'_2 are added to J set. In the next phase, new nodes in J set are explored to find their F handles. The exploration of j'_1 will add nodes c, d to F, and j'_2 exploration will add f' to F set. In next phase the transitivity relation from f' to j, f, j' will be discovered and f, j will be eliminated from their sets. But at this point, we have in F set the non-dependent nodes c, d, f', that are related to two different local NSP problem combinations named problem A, and problem B in the figure. If the exploration begins with f' or c, d instead of f, this situation does not happen. In this case, the way the resynchronization is done is not incorrect but non-optimal. In Fig. 3.16*(a)* is shown what is the result after resynchronizing problem A first, and then problem B in a natural way. Fig. 3.16*(b)* shows how the algorithm resynchronizes the branches when it mixes the problems.

Detecting and eliminating JF combinations as soon as possible minimizes the probability of mixing the problems. Each time we add a new f or j node to the sets, we can check for the transitivity relation from j nodes to f nodes. Thus, the JF combinations are detected and the wrong f node eliminated. Although this technique minimizes the probabilities of mixing the problems it may still happen. The local search for J,F nodes, in which the algorithm is based, can not avoid this problem.

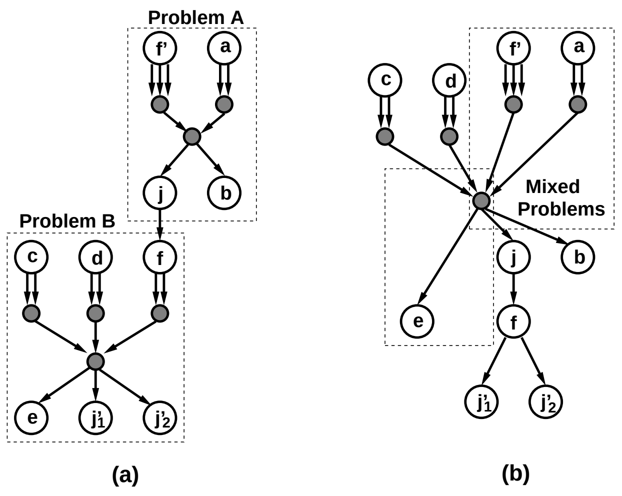


Figure 3.16: Solutions of a mixed problem through a JF combination

3.4.5 Example

We demonstrate the way our algorithm works with an example graph shown in Fig. 3.17*(a)*. Its minimal SP reduction graph, shown in Fig. 3.17*(b)*, has the edge annotations presented in Table 3.1.

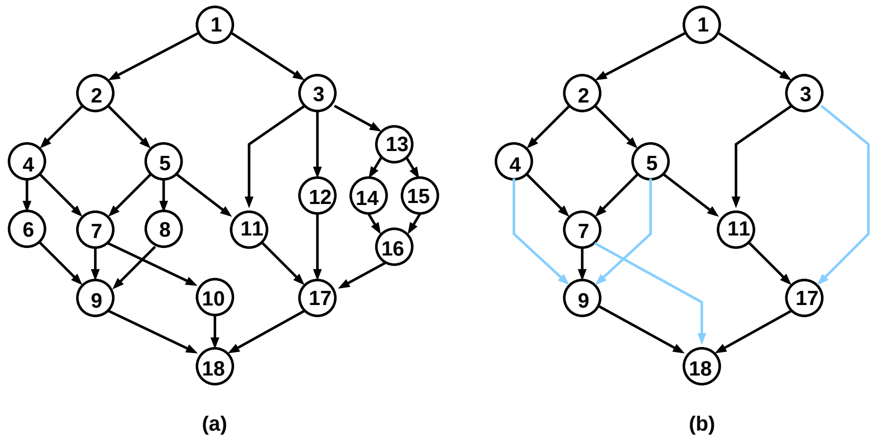


Figure 3.17: Example NSP graph and its minimal SP reduction

The evolution of the J and F sets during the problem handle detection phase can be seen in Table 3.2. The first column of this table (checkpoint number N) describes the event sequence. The algorithm would then proceed as follows:

We can choose as initial F node either nodes 3, 4, 5 or 7. All of them have

e	Z(e)
(4,9)	(6,9)
(5,9)	(8,9)
(7,18)	(10,18)
(3,17)	(12,17) (16,17)

Table 3.1: Edge annotations in the minimal SP reduction graph

SP branches to at least one J node. If we suppose that node 7 is the initial NSP problem, we would add it to the F set and explore it to locate its related J nodes 9 and 18, which should be included in the J set (checkpoints 1 $\rightarrow$ 2). In checkpoint 3, we explore the J nodes just added in the previous step and their related F nodes 4, 5 and 17 —which is taken as an F handle because it is also the origin of an SP branch. In checkpoint 4, we explore the next unexplored node in the F set, e.g. 4, and a new J node is obtained for the J set, namely node 7. In checkpoint 5, we test transitivities in J set, which implies the elimination of nodes 9 and 18, since node 7 represents their transitive closure and is the only one kept in the J set. In checkpoint 6, we detect how node 7 is also present in the F set, which represents a J–F combination to be ruled out from the F set. In checkpoint 7, the F handle 17 is taken out from the F set because there is no J node related to it in the J set. In checkpoint 8, we explore the next F node (5) and introduce a new J node in the J set (11). As a consequence, a new F node has to be added to the F set (3) after the exploration of this last J node; when this new F node (3) is explored, a new J node is added to the J set (17) which is then ruled out because of the transitivity relation with node 11. When we reach this point, we are at checkpoint 11 and there are no more J or F nodes unexplored, which concludes the search of the handles.

N	F set	J set
1	7	-
2	7	9,18
3	7,4,5,17	9,18
4	7,4,5,17	9,18,7
5	7,4,5,17	7
6	4,5,17	7
7	4,5	7
8	4,5	7,11
9	4,5,3	7,11
10	4,5,3	7,11,17
11	4,5,3	7,11

Table 3.2: Detecting the problem handles

After the resynchronization phase, both graphs, original and the minimal SP reduction with the same transformation, looks like Fig. 3.18*(a)* and *(b)* respectively. Computing the new minimal SP reduction graph, we obtain a trivial graph, because it is already SP. No more algorithm iterations are needed.

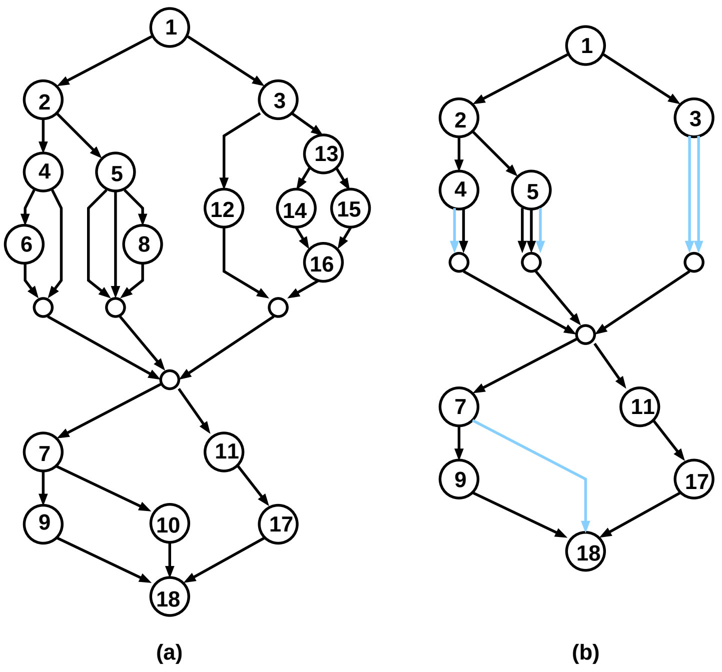


Figure 3.18: Solving the NSP problem in the original and SP reduced graph

3.4.6 Complexity

Space complexity

We use no more than two graphs at the same time; the original one and its minimal SP reduction. Any graph needs space for the nodes and for the edges. The nodes space is $O(n)$ and the edges $O(m)$. During the algorithm work we add a fixed amount of extra information in the nodes $O(n)$, and we also add annotations in the edges which represent SP branches. Annotations are non-overlapping subsets of the edges from the original graph. Thus, the whole amount of edges information is bounded by $O(m)$.

On the synchronizations we are adding more nodes. For each F node that is synchronized we add a new node. And one more node for each resynchronization. The number of F nodes can be $(n - 1)$. (All the nodes except the leaf). The number of resynchronizations depend on the number of independent NSP problems, bounded by the number of F nodes, at most $(n-1)$. So the number of solves we need is in the worst case $(n-1)$. (All the nodes except the leaf).

The number of final nodes on the SP graph is at most $n+2(n-1)$ that is $O(n)$. The final graph is SP, and it does not contain redundant edges and/or transitivities. Thus, the number of final edges is also $O(n)$ (see lemma 3.2.9). The final space complexity bound is:

$$O(n+m)$$

Time complexity

The complexity of the different operations that the algorithm does are:

- **Computing the transitive reduction of the graph:** Transitive edges mislead the algorithm to resynchronize non necessary parts of the graph. To improve solutions, transitive edges should be eliminated.

  Transitive closure and reduction is a well studied problem. The typical algorithm to compute transitive closure/reduction is Warshall's algorithm, based on Floyd's, with time complexity $O(n^3)$. However, faster algorithms, based on Strassen's matrix multiplication algorithm have been devised to obtain complexity $O(n^{2.81})$. See for example [29].

- **Compute the minimal SP reduction graph:** A node is series-reduced only once, eliminating two graph edges and introducing another (reductions for the whole graph are done in $O(n)$). When a node is series-reduced, it is possible to check if the edge already exists in the graph, avoiding including redundant edges and also parallel reduction operations. When a series reduction is performed, the source and target nodes can be checked to detect if the new change makes them available for series reduction recursively. Considering this strategy the total number of checks and reduction operations is done in $O(n)$, but the annotations update may need $O(m)$ time complexity.

- **Choose an NSP Problem:** Any F node can be checked. For any checked node we must traverse through any leaving edge looking for successors. In the worst case, all edges of the graph are evaluated to detect an F node with only J nodes as successors. Thus, the operation can be done in $O(m)$.

- **Identify problem handles:** This process is repeated until the problem is fully detected. We do not know how many nodes are related to the problem as F nodes or as J nodes, and some nodes can be explored in the two ways before the J/F problem is detected.

  To compute the upper bound we consider that any node could arrive at the F or J set, or both. Each time a node arrive at a J set we must check dependences in both directions. Dependencies for all the graph can be previously pre-computed, during the transitive reduction phase. Thus, this phase can be done in $O(n)$.

The check for old F nodes implies looking forward the successors of all F nodes in the set. As we consider that any number of nodes could be in the set, in the worst case we must traverse all the edges in the graph. When we look for predecessors of new J nodes, and successors of new F nodes we use the same considerations, so finally we can explore all the edges in the graph in both directions ($O(m)$).

Identify problem handles can be done in $O(n+m)$.

- **Solve problem:** The resynchronization moves the SP branches involved deleting their final edges (original edges from the graph) and adding one new edge to the synchronization point for every SP branch. In the worst case all the nodes minus one are in the F set, and all the leaving edges are SP branches, so all the edges in the graph are reallocated in $O(m)$.

  Then, the algorithm adds one edge from the synchronization point to any J node in the set. As no more than $n-1$ nodes can be J nodes, the operation can be done in $O(n)$.

All the operations described above are done once for any NSP problem. The algorithm does several operations with a maximum order $O(n+m)$. We do not know how many non-related NSP problems may be in a graph. We may assume a bad upper bound in which any F node is associated to a different NSP problem. The final number of detection and resynchronization iterations would be $n-1$. In each resynchronization we add one dummy node, so the number of nodes is growing in each iteration from n to $2n-1$. The number of nodes is always $O(n)$. The number of resynchronizations is $O(n)$. The number of operations for each resynchronization is $O(n+m)$. The time complexity of all problems resynchronization operations is: $O(n^2+n\times m)$. In a connected DAG, $O(m) \geq O(n)$. Hence, the time complexity is bounded by: (1) the transitive reduction computation (optional but strongly recommended), and (2) the graph resynchronizations $O(n\times m)$. Algorithm time complexity is:

$$O(n^{2.81}+n\times m)$$

Considering that transitive reduction is more necessary as the number of edges grows, and that $O(m) \leq O(n^2)$, when the product $n\times m$ is in $O(n^3)$ the problem solving dominates the transitive reduction. On the other hand, when $n\times m$ is in $O(n^{2.81})$, the transitive reduction could be skipped with minimum penalty for the algorithm solution. Thus, we conclude that the algorithm time complexity is dominated by:

$$O(n\times m)$$

3.5 Algorithm 2: Local layering technique

In this section we introduce a new SP-ization algorithm with the following interesting features [85]:

- A reduced time complexity: $O(m + n \log n)$.
- Local resynchronization of minimum number of nodes, guided by global topology information.
- It does not increase the critical path for UTC (Unit Time Cost) graphs, keeping the nodes layering structure of the original graph.
- The solution of the algorithm is the same for a given topology independently of the input order (node labeling).

The algorithm is based on a depth level search, solving local NSP problems while it traverses the graph. At any time, the already processed subgraph is SP. A tree representing the minimal series-parallel reduction graph of the processed subgraph is used to help in the search for handles, transitivity checks and operations that have lesser complexity bounds in a tree than in a generic DAG. Evaluation of edges that express dependences across several layers is delayed until the targeting layer is processed. A full implementation in JAVA language could be provided by the author upon request.

3.5.1 Notations

Let $G = (V_G, E_G)$ be the input graph:

Definition 3.5.1 *We define* d-edges *as the subset of edges which source and target have non-consecutive depth levels:*

$$(u, v) \in E_G : d(v) - d(u) > 1$$

□

Definition 3.5.2 *A* Layer *is the subset of graph nodes with the same depth level:*

$$L_i \subset V_G; L_i = \{v \in V_G : d(v) = i\}$$

□

3.5.2 Algorithm description

Initialization phase:

i. Transform the input DAG into an *STDAG* using the method presented in definition 3.1.17.

ii. Layering of the graph. Compute a partition of V_G, grouping nodes with the same depth level.

iii. Initialize an ancillary tree $T = (V_T, E_T)$ to L_0. This tree will represent the minimal series-parallel reduction of the step by step processed subgraphs.

Graph transformation:

For all layers (sorted) i from 0 to $D(G) - 1$:

a. Split layer in classes of *relatives*: Let us consider the subgraph $S \subset G$ formed by $L_i \cup L_{i+1}$ and all edges from G incident to two nodes in this subset. We construct the partition of this nodes into connected subgraphs. We define *relatives* classes as the subsets of nodes that belong to the same connected component of S and the same layer, as in Fig. 3.19.

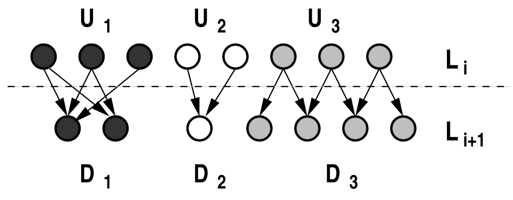


Figure 3.19: Example of relatives classes induced between two layers

$\mathcal{P}_U = \{U_1, U_2, ..., U_n\}$ will be the *up* classes (of nodes in L_i) and $\mathcal{P}_D = \{D_1, D_2, ..., D_n\}$ will be the *down* classes (of nodes in L_{i+1}). Each class $U \in \mathcal{P}_U$ induces a class $D \in \mathcal{P}_D$ that belongs to the same connected component ($U \to D$).

b. Tree exploration to detect *handles* for classes of relatives: We look in the tree for *handles*. For each U class, the *U-handle* ($h'(U)$) is the nearest common ancestor of all nodes in U:

$$H'(U) = \{v \in V_T : \forall w \in U, v \preceq_T w\}$$

$$h'(U) = h \in H'(U) : \forall h' \in H'(U) : d(h) \geq d(h')$$

We define $K_T(U)$ as the set of source nodes to the induced class D (it includes U and source nodes of d-edges targeting D): Sources of edges showing transitive dependence to D through the U-handle are to be discarded from $K_T(U)$:

$$K_T(U) = \{v \in V_T : (v,w) \in E_G, w \in D, v \not\preceq_T h'(U)\} \cup \{h'(U)\}$$

The *handle* node of class U, $h(U)$ is defined as:

$$H(U) = \{v \in V_T : \forall w \in K_T(D), v \preceq_T w\}$$

$$h(U) = h \in H(U) : \forall h' \in H(U) : d(h) \geq d(h')$$

We also define the forest of a class, as the set of complete sub-trees below $h(U)$ that include nodes in $K_T(U)$:

$$SubF(U) = \{u \in V_T : v \preceq_T u, (h(U),v) \in E_T, v \preceq_T w : w \in K_T(U)\}$$

In Fig. 3.20 we show a diagram of all concepts defined in this section.

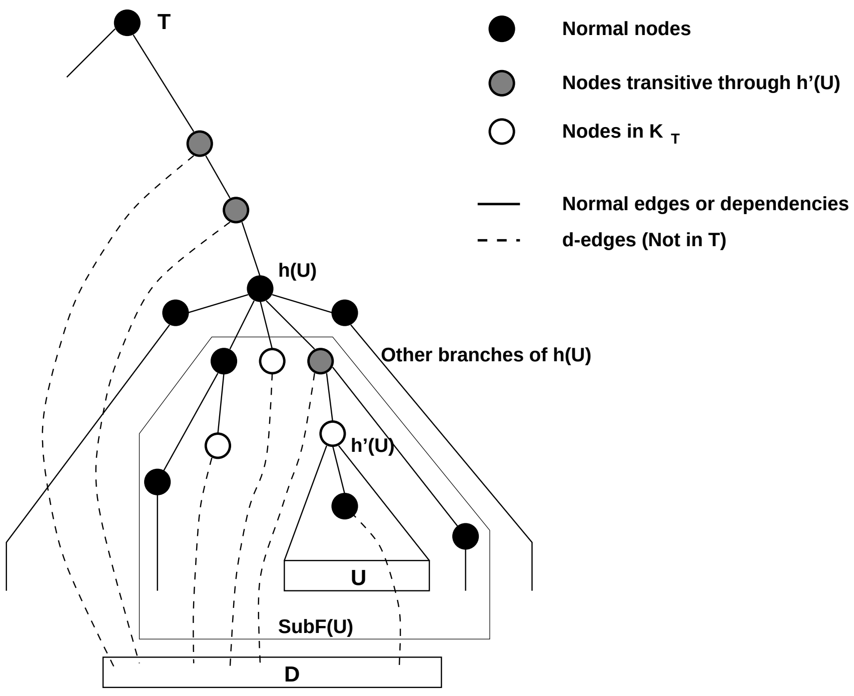


Figure 3.20: Example of handles and forest for an U class

c. Merge classes with overlapping forests: Classes with overlapping forests are merged in an unique U and D class. They will be synchronized with the same barrier.

$$\forall U, U' \in \mathcal{P}_U : SubF(U') \cap SubF(U) \neq \emptyset$$
$$U = U \cup U'; \mathcal{P}_U = \mathcal{P}_U \setminus U'$$
$$U \to D, U' \to D'; D = D \cup D'; \mathcal{P}_D = \mathcal{P}_D \setminus D'$$

d. Capture *orphan nodes*: We define *orphan nodes* as the leaves of the tree T that are not in any U class (they are nodes in layers previous to i with only d-edges to layers further than $i+1$). These nodes are included in the U class of the forest they belong to.

$$\forall v \in SubF(U), v \in L(T), v \notin U; U = U \cup \{v\}$$

e. Class barrier synchronization: For each final $U \to D$ classes:

- Create a new synchronization node b_U in the graph and the tree.

$$V_G = V_G \cup \{b_U\}$$
$$V_T = V_T \cup \{b_U\}$$

- In G, eliminate all edges targeting a node in D. Add edges from every node in U to b_U and from b_U to every node in D (barrier synchronization).

$$E_G = E_G \setminus \{(v, w) : w \in D\}$$
$$E_G = E_G \cup \{(v, b_U) : v \in U\}$$
$$E_G = E_G \cup \{(b_U, w) : w \in D\}$$

- Substitute every d-edge (v, w) with source $v \in SubF(U)$ and targeting a node $w \in L_k : k > i+1$ (a further layer) for an edge (b_U, w). This operation eliminate d-edges from the new synchronized SP subgraph, but avoiding the loss of dependences in the original graph.

$$dE(U) = \{(v, w) \in G : v \in SubF(U), w \in L_k, k > i+1\}$$
$$E_G = E_G \cup \{(b_U, w) : (v, w) \in dE(U)\}$$
$$E_G = E_G \setminus dE(U)$$

- Substitute the forest $SubF(U)$ in T for an edge $(h(U), b_U)$ representing the minimal series-parallel reduction of the new synchronized SP subgraph.

$$T = T \setminus SubF(U)$$
$$E_T = E_T \cup \{(h(U), b_U)\}$$

3.5.3 Example

An example of the algorithm applied to a given graph is shown, step by step, in Fig. 3.21,3.22,3.23. For each step, the first and second columns present the graph and tree respectively, as a result of the previous step. For step 1 we present the original graph with a layering diagram and the root initialized tree. The third column is a diagram of the exploration phase on the tree. U and D node classes are shown with different grey shades, showing the graph related edges (not in the tree) by dashed lines.

We also mark with names the orphan nodes, d-edges to further layers and the transitive/non-transitive property over the U-handle, of the d-edges arriving at D classes. U-handles are marked with $h'(U)$ and final handles with $h(U)$. Forests under each handle are surrounded by trapezoids. New added synchronization nodes are represented with smaller circles.

We comment now the remarkable algorithm features in the example. Step 1 presents a case with only one U class with one node in the U class (handle) and two nodes in the induced D class. A new node 19 is added to the graph to synchronize over the nodes in the D class. In step 2, there are two U class to synchronize, being the handles the nodes in U classes. A d-edge appears from a node in the second class, and it source node 3 is changed in the original graph to the new synchronization node 21. Exploring phase in step 3, detects node 20 as the U-handle of the first U class as the nearest common ancestor of all nodes in U class (4,5). However, a d-edge to a node in the induced D class (21,11), which source node 21 is not transitive through the U-handle node 20, forces to explore further. The handle for class 1 is not equal to the U-handle, but the nearest common ancestor of nodes 20 and 21, namely node 19. Moreover, forests under the handles of classes 1 and 2 overlaps in node 13, and they are merged and synchronized together. Notice how the orphan node 12 is included in the merged U class and synchronized over the new node 22. Step 4 presents a situation where two U classes have the same handle node 22, but non-overlapping forests. Thus, they are not merged, but synchronized with different nodes 23 and 24. In step 5 there is only one U class, because nodes 9 and 10 have only d-edges to further layers. The U-handle is the same node 16 in U class. Nevertheless, there are d-edges from previous layers. Edge (22,17) is discarded due to its transitive property through the U-handle 16. However, edge (23,17) is not transitive. Thus, the handle node is the nearest common ancestor of nodes 16 and 23, namely node 22. The forest include now orphan nodes 9 and 10. In last step 6, there is only one U class and two discarded transitive edges. The resulting graph is shown together with the final tree, that is always a series graph in which each edge represents the minimal series-parallel reduction of a full SP subgraph.

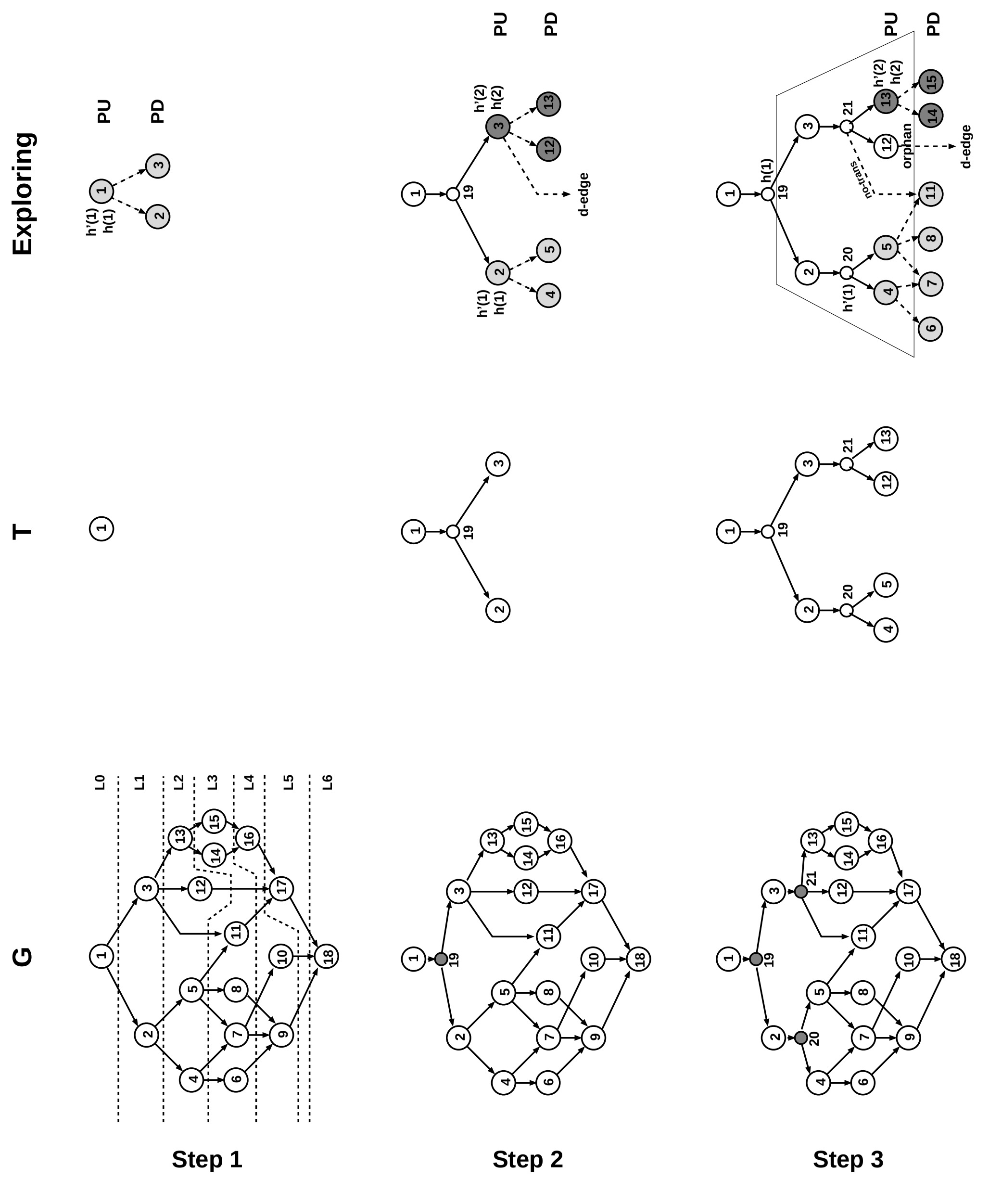


Figure 3.21: Example of algorithm 2: Steps 1,2,3

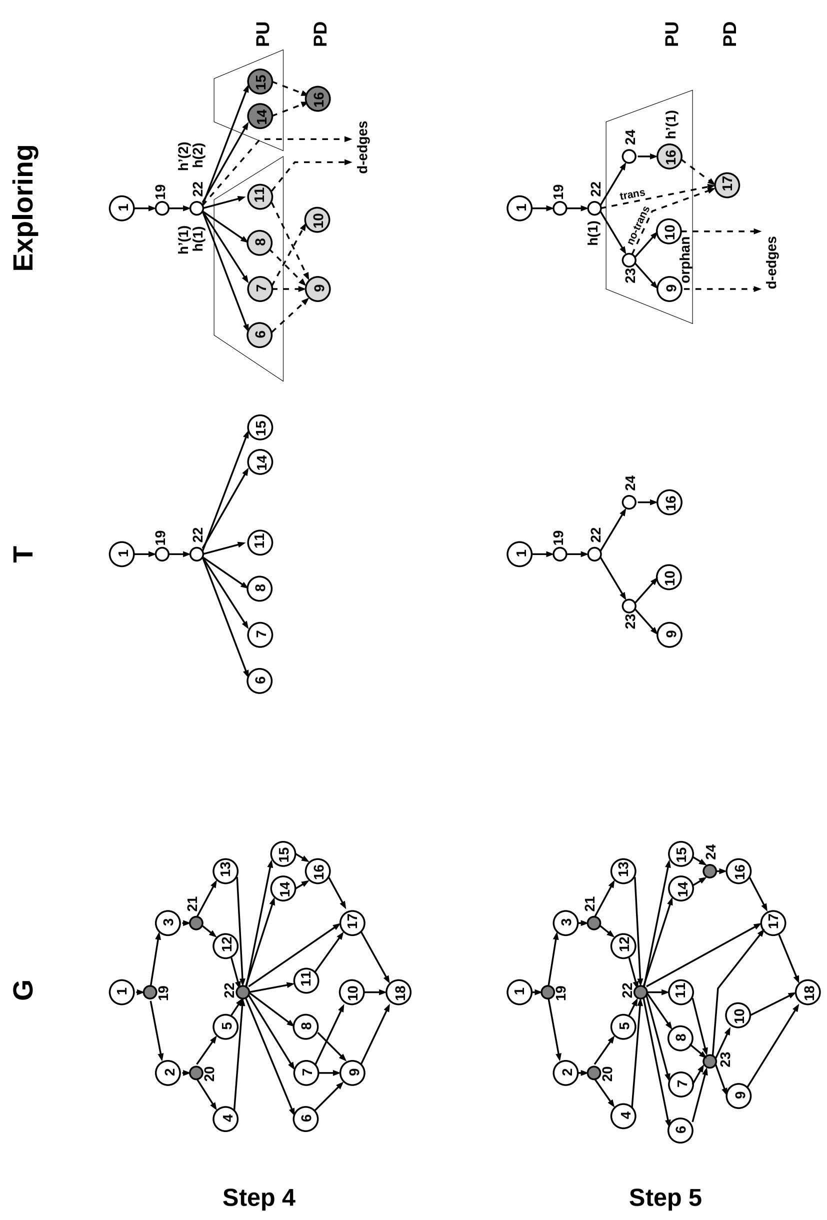


Figure 3.22: Example of algorithm 2: Steps 4,5

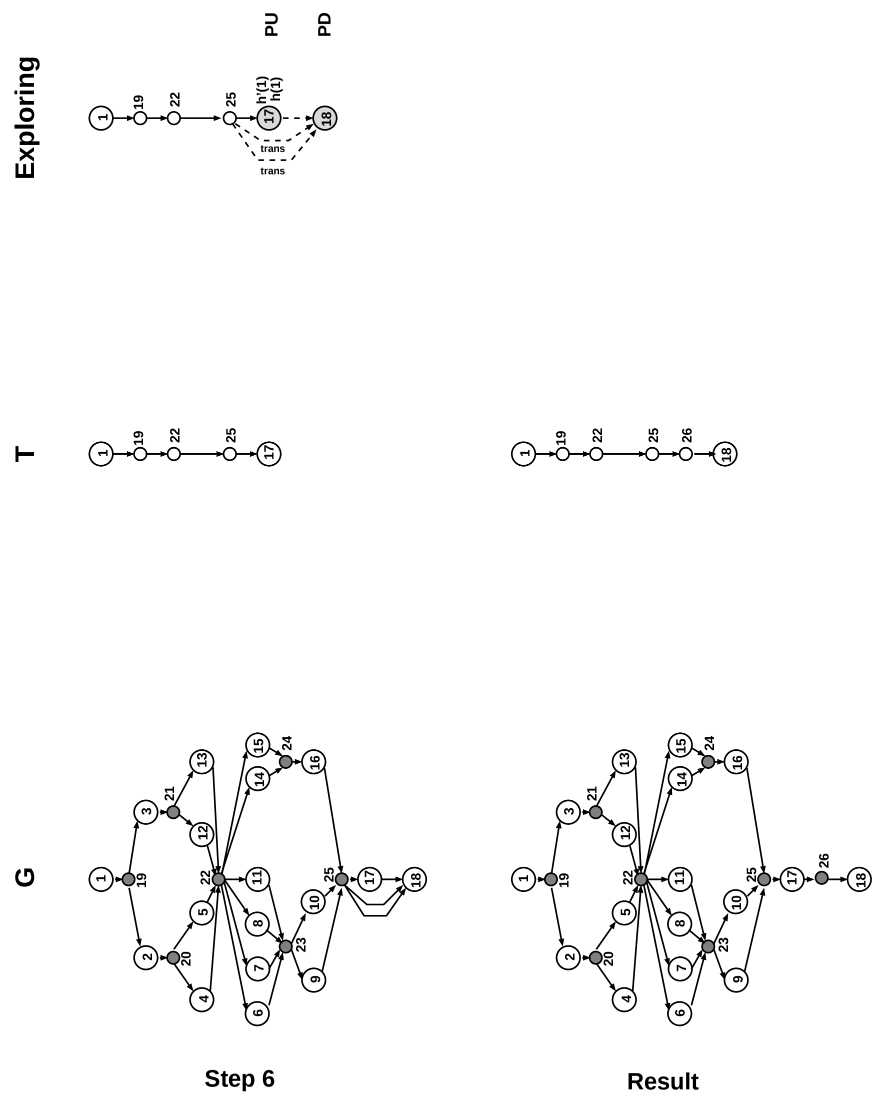


Figure 3.23: Example of algorithm 2: Step 6 & Result

3.5.4 Correctness

Since any tree can be easily transformed to a trivial SP $STDAG$, any graph which minimal series-parallel reduction is a tree, will be SP. As can be easily shown by induction on the depth of the $STDAG$, the minimal series-parallel reduction graph at each step is always a tree and, thus, has the SP property.

We define $TDAG$ as the subset of $DAGs$ that are one-rooted connected trees.

Proposition 3.5.3 *A tree is SP:*

$$T \in TDAG \Rightarrow T \in SP$$

Proof: *The $STDAG$ of T (called* closure of T*) is the original T with an added leaf b_U connected to all the leaves $L(T)$. Applying series reduction to all originally leaves of T and parallel reductions where there were several leaves with the same parent, the result is equal to the closure $STDAG$ of T', being T' the tree obtained eliminating $L(T)$ from T. Proceed recursively until only the root of T and the new leaf b_U remain and the reduction is the trivial graph.* □

Proposition 3.5.4 *A graph G which series-parallel reduction is a tree is SP.*

Proof: *Compute the series-parallel reduction of G until it is a tree. As proved previously the series-parallel reduction of the closure of a tree is the trivial graph. Thus, the $STDAG$ of the original graph can be series-parallel reduced to the trivial graph and is also SP.* □

Correctness proof:

1. **The result does not loose dependences:** No node is eliminated from the graph. During synchronization, all times an edge (v, w) is eliminated it is substituted by two edges (v, b_U) and $b_U, w)$. Thus, the original dependence is transitively keep through b_U. All times a d-edge $(v, w) : v \in SubF(U), v \in L_j, j \leq i \wedge w \in L_k, k > i + 1$ is moved down to the synchronization node, the original edge disappears and another edge (b_U, w) is added. After adding edges from U to b_U, $\forall u \in SubF(U), u \prec b_U$ and $v \prec b_U \prec w$.

   Thus, during the synchronization phase neither, the substitution of edges or moving down d-edges eliminate original dependences in G. No other edge alteration is done in G.

2. **The result is SP:** We call S_i the subgraph of G that includes all nodes in layers $L_0, L_1, ..., L_i$ and all G edges incident to both nodes in S_i.

When the algorithm begins (for $i = 0$) T is initialized with the root of G. S_0 is a one node tree. For $i = 1$ the closure of T and S_0 is computed and nodes in L_1 are hanged from the new synchronization node. T and S_1 are trees and, thus, they are SP.

In each subsequent iteration (for $i = i + 1$), we compute $\mathcal{P}_U$, $\mathcal{P}_D$ and their handles. Then we merge classes with overlapping forests. Each forest is composed by trees that represent the series-parallel reduction of a subgraph of S_i. Eliminating in G edges from U to D and d-edges from $SubF(U)$ for all classes, S_i gets disconnected from the rest of the graph, being a tree (or a graph that is a tree after series-parallel reductions). New synchronization nodes and edges are added to closure every tree in T and G included in a forest of an U class. Thus, after synchronization, S_{i+1} is a tree or a graph that is a tree after series-parallel reductions. S_{i+1} is SP. T represents the series-parallel reduction of S_{i+1}.

Proceed by induction until the last iteration. In last iteration (for $i = D(G) - 1$), L_{i+1} is formed by the only one leaf of G. There is only one U class and one D class. All resting sub-trees in T (and G) are closed together with only one synchronization node and only one node (the leaf of G) is added hanging from that new node. T, that represents the series-parallel reduction of G is a series of nodes, its series reduction is the trivial graph. Thus, G is SP.

3.5.5 Critical path property for UTC graphs

An interesting feature of the algorithm is that it does not increase the critical path value if the original graph has unit time cost per node. Transforming a graph to SP form, this property minimizes the possibilities for critical path increment when no knowledge of the task load distribution is available.

Proposition 3.5.5 *For an UTC (Unit Time Cost) input graph G, the result G' is not UTC (nodes added by the algorithm carry no load), but despite the added dependences, the critical path is not increased.*

Proof: *For UTC graphs, the critical path value of G is equal to the maximum number of nodes that can be traversed from a root to a leaf ($cpv(G) = 1 + D(G)$).*

The algorithm adds zero loaded synchronization nodes between layers. The only way of increasing the critical path is due to added dependences that make a node from a layer i dependent on a node from layer j, being $j > i$. However, the algorithm keeps the layers structure.

Moving d-edges sources to a node in a layer previous to the target node layer, does not change the depth level of any node. Substituting edges from nodes in U

classes to nodes in D classes to include b_U nodes keeps the depth level of U nodes and adds one to the depth level of every node in D classes.

In the resulting graph, all even layers are populated by zero loaded nodes and odd layers by nodes in the original layers. The longest path from the root to the leaf alternatively crosses nodes with unit and zero time cost. The number of unit time cost nodes in the longest path is at most $1 + D(G)$, and, thus, the critical path value in G' is the same as in G. □

3.5.6 Complexity

Space complexity

Let n be the number of nodes and m the number of edges in the original graph. The number of nodes in the graph increases with one more node for each U class. Every node appears just once in an U class over the full algorithm run. Thus, the total number of nodes is upper bounded by $2n$. The number of edges is upper bounded because the processed subgraph (after each iteration) is SP, and the number of edges in an SP graph is bounded by $m \leq 2(n-2)$ (see lemma 3.2.9). Other ancillary structures (as the tree) store graph nodes and/or edges. Thus, space complexity is:

$$O(m+n)$$

Time complexity

$STDAG$ construction can be done in $O(n)$ and getting layers information in $O(m)$ with a simple graph search.

Classes of relatives for two consecutive layers can be computed testing a constant number of times each edge. Thus, all the classes along the algorithm run are computed in $O(m)$.

Exploration of the tree for handles can be self-destructive: Nodes are eliminated during the search. While searching for the handle of a class, all the forest can be eliminated and orphan nodes and other classes to be merged detected (see section 3.5.7 for a description of such an implementation).

Check and eliminate a transitive edge can be done in $O(1)$ if appropriate data structures are used for the tree [21], but assuming tree modifications are done in $O(\log n)$. $O(n)$ nodes and edges are inserted and eliminated in the tree. Thus, all tree manipulation has a time complexity $O(n \log n)$.

The synchronization phase adds $O(n)$ nodes, eliminate $O(m)$ edges and add a bounded number of edges ($O(n)$ because it is an SP graph). The movement of d-edges can be traced in $O(n \log n)$ with a tree-like groups joining structure to avoid real edge manipulation (see 3.5.7 for details).

Thus, time complexity is:

$$O(m + n \log n)$$

3.5.7 Implementation

We propose an implementation for the tree exploring phase. This implementation is based on a self-destructive search of the tree that eliminates the already used forests from the tree and detect handles with only one check per node. This implementation is needed to bound the time complexity as explained in section 3.5.6.

Searching for handles: For any given U class, we create an exploration structure call *explorers* (E). This structure stores nodes in sets indexed by depth level.

$$E = (m, V_E); m \in \mathbb{N}, V_E = \{V_1, V_2, ..., V_m\}$$

We initialize it with the nodes in any chosen U class.

$$\forall v \in U : V_{d(v)} = V_{d(v)} \cup \{v\}$$

$$m = \max d(v) : v \in U$$

For all nodes in E with maximum depth, we eliminate them from the tree, and we add the parent of the eliminated node to the explorers structure (avoiding repetition by marking the parent node when first visited).

To eliminate a tree node, we check previously if it is a leaf. If it is not, we proceed to eliminate all sub-trees hanging from it. The leaves of these sub-trees will be orphan nodes (that we immediately add to U) or nodes in other U classes. In this last case, both classes are merged, adding the new U nodes to the explorers structure.

When the explorers structure has only one node, this node is the U-handle $h'(U)$. Then we check the transitive condition of all d-edges arriving at D in the tree with $h'(U)$ to compute $K'_T(U)$. Non-transitive d-edges sources are added to explorers and the search is continued until the structure has again only one node. This last node is the handle $h(U)$, and is marked in the tree (a node can be handle of several classes at the same time).

During exploration, a node that is processed to be eliminated can also be marked as handle of other previously explored class or classes. In this case these classes are also merged with the one being explored.

When this exploring operation is performed for all U classes, all handles have been detected and marked, related classes already merged, and forests $SubF(U)$ deleted from the tree.

Tracking of d-edges: During the elimination of tree nodes we keep track of d-edges from these nodes to further layers. Each class maintains a set of these source nodes. When classes are merged, these sets are also merged. When a class is synchronized, this set will provide information for d-edges to be moved to the new synchronization node.

To keep track of d-edges movements without performing modifications in the graph, we use a modified version of a disjoint-sets data structure with union by rank and path compression (see e.g. [47]). The structure will map any node label to the node label of the final source of the associated d-edges. A joining operation of a pair of node labels (i, j) will indicate that d-edges with source i are to be mapped to source node j. The structure has the property that for any sequence of joining operations $(i_1, j_1), (i_2, j_2), ..., (i_n, j_n)$ where $i_1 \neq i_2 \neq ... \neq i_n$ all joining operations take $O(n \log n)$ to be performed, and any mapping query takes $O(1)$.

Definition 3.5.6 *We define the* Joining structure $J = (\vec{I}, \vec{W}, \vec{S})$, *where* $\vec{I}, \vec{W}$ *are arrays of indexes and* $\vec{S}$ *is an array of sets of node labels (we define* N *as the set of all possible node labels). Let* $n = |V_G|$*:*

$$N = \{i : \mathbb{N}; i \in [1..2n]\}$$

$$\vec{I}, \vec{W} : N^{2n}$$

$$\vec{S} : S^{2n}, S_i \subseteq \{v : N\}$$

The J *structure is initialized as follows:*

$$I_i = i; W_i = i; S_i = \{i\}$$

It supports a joining operation indicating that i *must be mapped to* j *defined as:*

$$J \pitchfork (i, j) : J \to J'; J = (\vec{I}, \vec{W}, \vec{S}), J' = (\vec{I}', \vec{W}', \vec{S}');$$

$$I'_{(W_i)} = I_{(W_j)}$$

$$big = \begin{cases} W_i & \textit{if } |S_{W_i}| \geq |S_{W_j}| \\ W_j & \textit{otherwise} \end{cases}$$

$$small = \begin{cases} W_i & \textit{if } |S_{W_i}| < |S_{W_j}| \\ W_j & \textit{otherwise} \end{cases}$$

$$W'_i = W'_j = big$$

$$S'_{big} = S_{big} \cup S_{small}$$

$$\forall k \in S_{small} : W'_k = big$$

The query function is defined as:

$$J : V_G \to V_T; J(i) = W_i$$

□

3.5.8 Improvement: Unnecessary synchronization nodes

Some synchronization nodes may be eliminated. In situations where the U class, the induced D class, or both, have only one node, the new synchronization node is not necessary. The lonely node can play that role. This reduces the number of nodes and edges added, producing a completely equivalent graph result in terms of structure and dependences between nodes from the original graph.

We modify the algorithm synchronization phase along the following lines. For each final $U \rightarrow D$ classes:

- Detect/create synchronization node, and eliminate/add edges:
    1. If $U = \{u\}$, $b_U = u$:
       In G, eliminate all d-edges targeting a node in D.
       $$E_G = E_G \setminus \{(v,w) : w \in D, d(v) < i\}$$
    2. Else if $D = \{d\}$, $b_U = d$:
       In G, eliminate all d-edges targeting a node in D.
       $$E_G = E_G \setminus \{(v,w) : w \in D, d(v) < i\}$$
    3. Else (normal case where $|U| > 1, |D| > 1$): Proceed as in the original algorithm creating a new synchronization node b_U, eliminating in G all edges targeting a node in D, and adding edges from every node in U to b_U and from b_U to every node in D (barrier synchronization).
- Substitution of d-edges with source $v \in SubF(U)$, as in the original algorithm.
- Substitute the forest $SubF(U)$ in T for an edge $(h(U), b_U)$, as in the original algorithm.

In Fig. 3.24 we show the solutions obtained with the normal and the improved algorithms for the same graph example used previously. The dependences structure created on the original graph nodes is the same for both solutions, although the improved algorithm uses less nodes and edges.

3.6 Measuring the SP-ization impact

We discuss now methods to evaluate the SP-ization impact in terms of structural modification of the original topology and potential loss of performance after the transformation. We study different possible alternatives of the transformation impact. The objective is to propose a measure which allows us: (1) to evaluate how different SP-ization techniques perform on a given graph, in order to

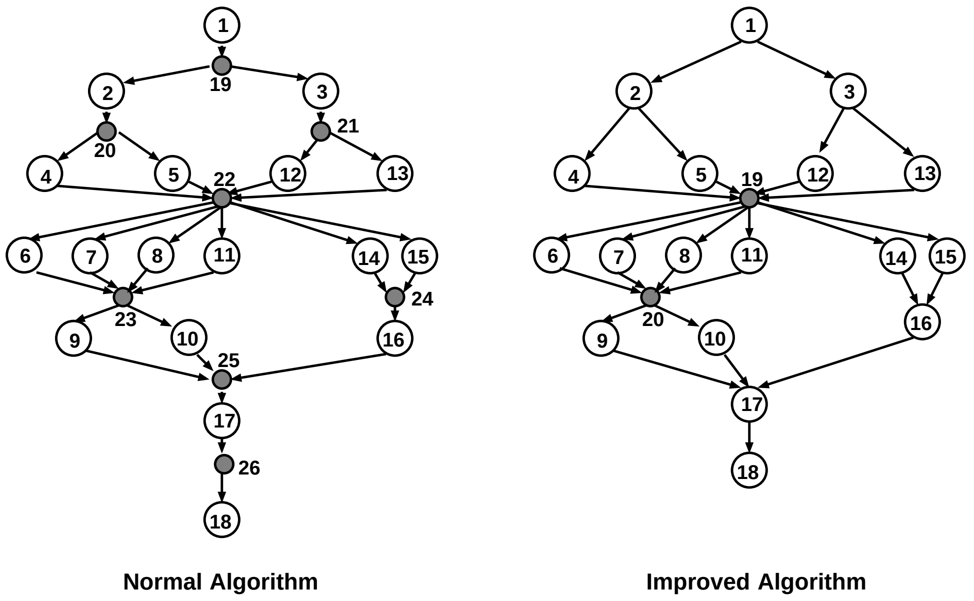


Figure 3.24: Solutions obtained by the normal and improved algorithms

compare the techniques themselves, and (2) to study, for an *ideal* transformation algorithm, which topological or workload parameters of the graph are related to the potential parallelism or performance loss induced by the added dependences. After proposing a measure, we try to relate the potential transformation impact to simple topological graph parameters as the depth level, the degree of parallelism or the synchronization density (see formal definitions on section 3.1.3). Analytical models and experimental measures are discussed.

3.6.1 Potential performance impact

In this section we focus into the analysis of the potential impact of an SP-ization in the final performance of the application through critical path value (cpv) analysis. We say *potential* because we are applying transformations at the programming level of abstraction. The program will suffer subsequent transformations in order to optimize and map it to a specific machine. The transformation path will be quite different in NSP and SP cases, leading to unexpected benefits or losses in the final performance. However, we are interested in the potential impact of the programming high-level transformations, as it will be an important part of the final performance effect.

We use the critical path value cpv to measure the performance of an application, modeled as a task graph, for a given workload distribution τ (see section 3.1.4). Thus, for the analysis of the performance degradation of an application when programmed in an SP PPM, we compare the *cpv* of the graphs that model: (1) the original synchronization structure of the application and (2) the structure produced by an SP-ization.

Definition 3.6.1 *Given two graphs G, G' modeling the same application, and a load distribution τ, we define the* Relative critical path difference *between the two graphs G, G' or $\gamma_\tau(G, G')$, as:*

$$\gamma_\tau(G, G') = \frac{cpv(G')}{cpv(G)}$$

The mean of the relative critical path difference between two given graphs G, G', for several workload distributions, is defined as:

$$\overline{\gamma}(G, G') = \sum_{i=1}^{n} \frac{\gamma_{\tau_i}(G, G')}{n}$$

The upper bounds of the performance loss correspond to very unlikely cases of highly unbalanced computations, where pathological workload distributions appear. However, parallel applications are designed with load-balance and regular work distribution in mind. Also for dynamic codes, where structure and task workloads are generated by processes taking random or data dependent choices, an average cost study is more appropriate [122].

The average cost will be studied as a function of the topology characteristics, workload model and SP-ization technique used for the transformation.

Definition 3.6.2 *Let T be an SP-ization technique, we define:*

$$\gamma_\tau^T(G) = \gamma_\tau(G, G') : T(G) = G'$$

$$\overline{\gamma}^T(G) = \overline{\gamma}(G, G') : T(G) = G'$$

From now on, we will use γ as $\overline{\gamma}^T$ when the transformation technique used is obvious from the context. As this measure is dependent on the transformation applied, it can be also used to evaluate and compare how different transformation techniques may affect performance (see section 3.6.3).

This indicator, γ, is defined for a given graph and transformation technique. Thus, it is an experimental measure. Several γ measures may be distinguished depending on the level of detail or abstraction level at which the graph model of a given application is derived (see Fig. 3.25). A program is an expression of an algorithm to solve a problem in an specific PPM. At this first level, the graph represents the synchronization structure that the program creates; or may create for a given input data in case of dynamic applications (see section 2.6). When a

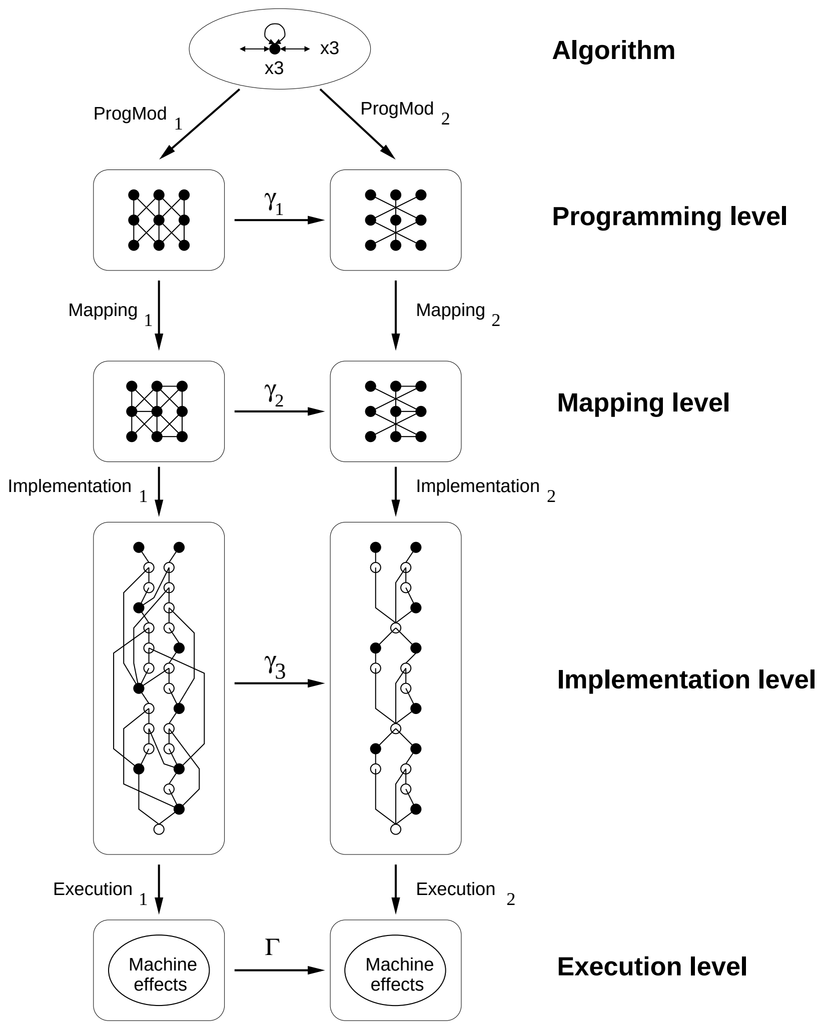


Figure 3.25: Implementation trajectory. Abstraction levels

program is mapped to a given machine, with a fixed number of resources (such as processors), the graph structure may change. We say that these modified graphs are modeling the application at mapping level. Graph models may be constructed even for lower levels of detail, including even specific communication and synchronization tasks. Then, they are modeling applications at implementation or machine level, where the underlying communication system is relevant. Thus, we distinguish several γ levels: γ_1 for programming level; γ_2 for mapping level; and γ_3 for implementation level.

Our study is mainly focused at γ_1. Transformations made to an application by compilers and implementation systems during the mapping and the implementation phases are difficult to predict and in general are favored by a restricted PPM as discussed previously. Thus, our interest is to determined (analytically or experimentally) the potential performance degradation at programming level. How, or how much, the underlying technology may improve γ_1 estimations is not part of this work. However, for experimental measurement of γ a sufficient level of detail should be considered in the graph model to assure enough accuracy. Thus, sometimes it will be necessary to measure γ_2 or even γ_3 values with mapping or implementation level graphs. Application modeling with graphs at different detail levels is discussed in sections 4.2.2 and 4.2.3.

To determine the accuracy and relevance of γ predictions, we must check our results against measurements of real performance when applications are implemented through different PPMs. We define Γ as a measure of the *real* performance degradation when the same algorithm or application is programmed, implemented and executed through different PPMs.

Definition 3.6.3 *Let imp_1, imp_2 be two different implementations of an application or kernel algorithm for a given machine; and $t(imp_1), t(imp_2)$ the execution times of these implementations as measured in the real machine. We define the* Relative real performance degradation Γ *as:*

$$\Gamma(imp_1, imp_2) = \frac{t(imp_2)}{t(imp_1)}$$

A full experimental framework, comparing Γ measurements with more abstract level γ predictions, is presented in chapter 4.

3.6.2 Structural impact

In this section we explore measures of the structural impact of an SP-ization in the graph topology and we will try to relate them to the critical path value increment represented by γ. A first approximation to a measure of the impact of an SP-ization in a graph may be the *distance* to SP form (as defined in section 3.2.2). Another could be the number of local barrier synchronizations added by the transformation. However, these indicators are not good measures. The loss of parallelism is produced by the added dependences that serialize potentially parallel tasks, and the number of dependences added by each technique for a local resynchronization can be completely different even if the number of resynchronizations is the same. The possible impact on the final performance is related to the probability of a critical path increase, induced by new dependences.

Generally, as long as we do not have information about the exact workload of the graph nodes, our first proposal for a measure to represent the probability of critical path increase is the number of added dependences itself. The number of node dependences in a DAG is the number of edges m in the transitive closure G^+. Hence, the number of added dependences is the difference in the number of edges between the transitive closure of the SP transformed graph and the transitive closure of the original NSP graph. The edges from/to new nodes eventually introduced by the transformation does not account for the number of added dependences.

Definition 3.6.4 *The* transformation distance $\theta(G,T)$ *produced by the SP-ization T in the graph G is the difference of the number of edges (only related to nodes in V) between the transitive closure of G' and G.*

$$G = (V,E), \quad G^+ = (V,E^+),$$
$$G' = T(G) = (V',E'), \quad G'^+ = (V',E'^+),$$

$$\theta(G,T) = |\{(v,w) \in E'^+ : v,w \in V\}| - |E^+|$$

This transformation distance can be used to compare how different SP-ization techniques perform for a given graph topology without knowledge of specific workloads. The Fig. 3.26 shows an example graph of low synchronization density transformed with four different techniques: Layering; both algorithmic techniques proposed in chapter 3 (Algorithm1,Algorithm2) and a manual solution obtained by applying down synchronizations guided by personal experience. The node labels show the number of dependences from other nodes. Dark nodes are added for synchronization and they are not considered in the dependences count. The transformation distances obtained, point to the manual solution as the transformation with the lower structural impact ($\theta = 1.5$). However, an important graph parameter, the maximum depth level (D), has been duplicated. Our transformation algorithms are the second option ($\theta = 1.64$), while layering technique has a great structural impact ($\theta = 1.93$). However, our second algorithm does not increase the maximum depth level of the original graph (always discarding new synchronization nodes), while the first algorithm technique does increase it. In fact, the maximum depth level value is an important parameter for critical path values in a graph, (see discussion about the transformation algorithms in section 3.6.3).

In Fig. 3.27 we show the results obtained in an experiment conducted to relate the θ indicator with the mean increase of the critical path value (cpv). We select random workloads with four different Gaussian random distributions (see section 4.1.1):

$$\tau(v \in V) \rightsquigarrow N(\mu,\sigma) : \mu = 1.0, \sigma \in \{0.1, 0.2, 0.5, 1.0\}$$

The different deviations represent different load balancing situations. From very well balanced ($\sigma = 0.1$) to highly unbalanced ($\sigma = 1.0$). For each example graph,

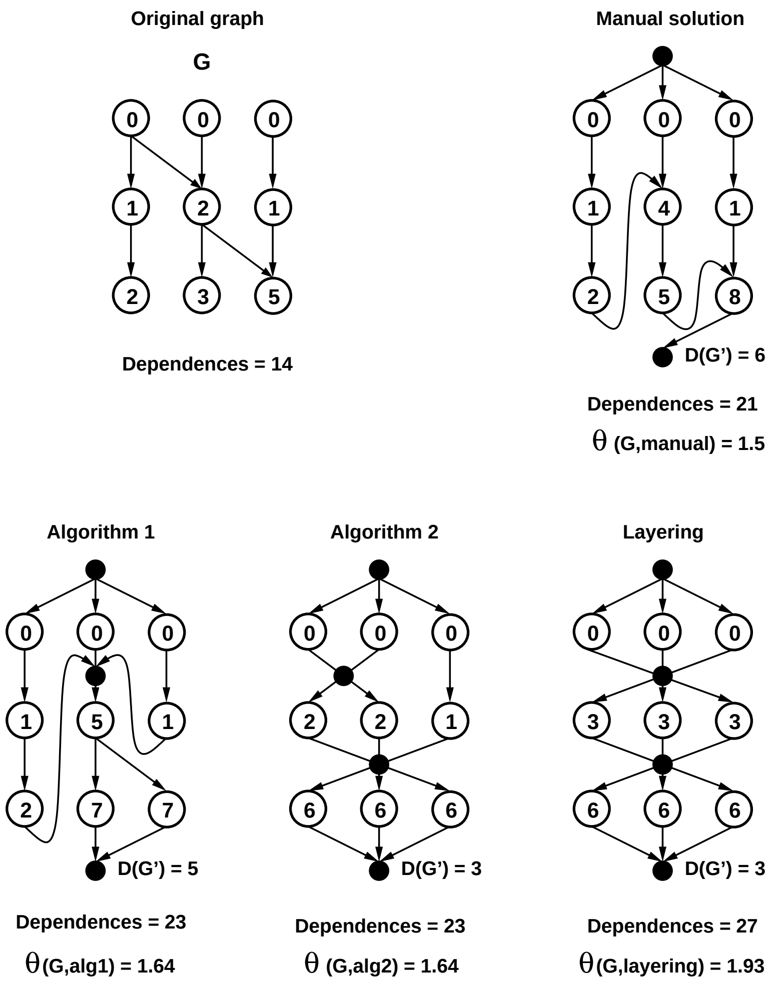


Figure 3.26: Measuring transformation distance (θ)

and each Gaussian model, we measure the mean *cpv* produced when drawing 1000 different random workloads. The results indicate that for this specific low S (synchronization density) topology, the techniques that do not increase D value have lower impact in the critical path value in balanced situations. In these cases all nodes have very similar load values, and the number of nodes in the critical path becomes a key factor. However, when the loads are highly unbalanced and random distributed, the techniques that minimizes θ may obtain better results. All these results, although typical, may not be extended to any other topology.

Thus, we conclude that the structural impact of a transformation technique alone, measured as the relative number of added dependences θ, is not a good

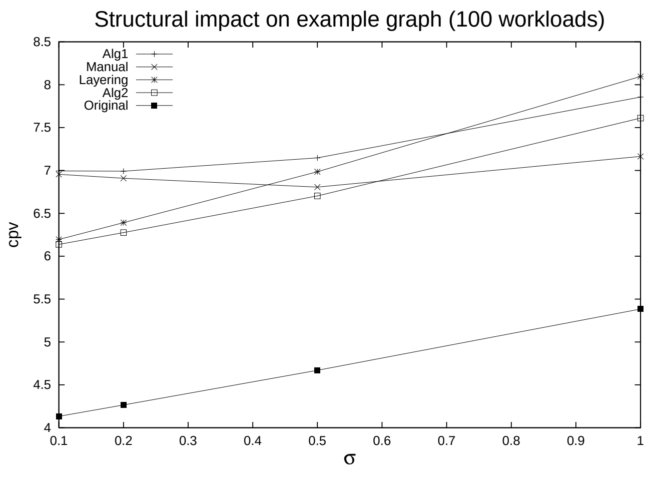


Figure 3.27: Experimental *cpv* measurements for the example graph

indicator of the potential performance impact. Other factors, as the maximum depth level D or the workload deviation are as important as θ to determine the potential increment of the critical path value of a transformed graph.

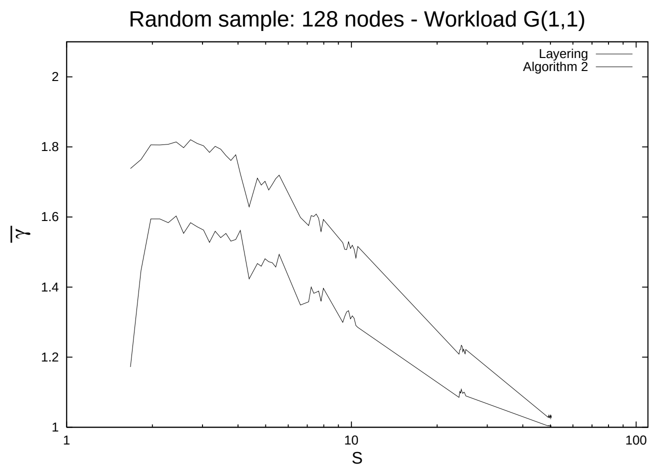
For low deviated workloads the D parameter is a key factor of critical path value increase. Thus, keeping the original *cpv* for UTC (Unit Time Cost) graphs is an interesting design principle for SP-ization techniques (see 3.5). For more unbalanced workloads, no relation between the increment of the critical path value and a combination of structural impact parameters have been yet found for any graph topology or size. It is still an open problem.

3.6.3 Algorithms comparison

In this section we compare the techniques and algorithms previously described. Complexity, suitability for any kind of graphs, and mean increment of the critical path value γ are to be considered to evaluate the applicability of these techniques.

The first technique presented in section 3.3.4, which serializes all nodes in topological search order, is not suitable for parallel computing purposes, as all the parallelism is lost after the transformation. For simplicity we will refer to the other three techniques as Layering, Algorithm1 and Algorithm2. Results are summarized in Table 3.3.

Experimentally transforming many structures from highly regular applications (see section 4.1.3), we have found that, for these highly regular structures, the three techniques obtain similar results. Nevertheless, the SP forms obtained differ for more irregular structures. While the simple layering technique (Layering), has the lower time complexity bounds it does not offer good



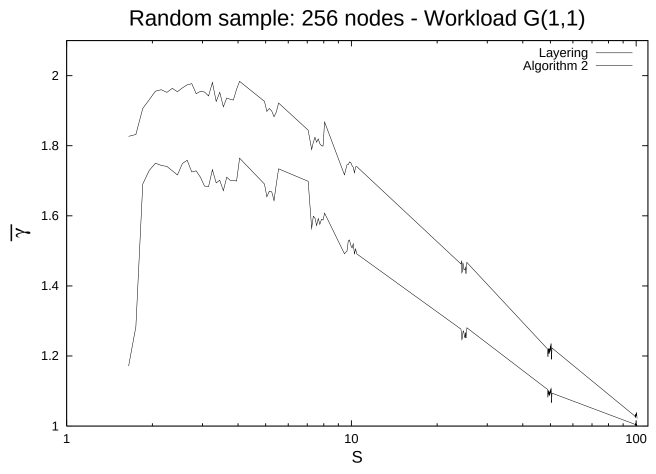


Figure 3.28: Comparison of algorithms increase in *cpv* for random graphs sample

results for irregular structures. Algorithm2 minimizes the expected critical path value increases as compared with the layering technique due to the capability to exploit local resynchronizations instead of using only global barriers. Experiments with random generated topologies and workloads have been conducted to compare which algorithm produce SP approximations with lower expected critical path value increment (see section 4.1.2). We study mean values of the γ indicator (as defined in section 3.6.1). All our experiments with different graph sizes and workload models confirm the $\overline{\gamma}$ trends for each algorithm. For example, Fig. 3.28 illustrates how Algorithm2 typically finds better solutions for two different samples of 128 and 256 nodes graphs respectively. The size of each sample is 1000 graphs. The synthetic random workload model used for this example is highly deviated $\tau(v) \rightsquigarrow N(1,1)$. Details about the experiments design and more results are shown in section 4.1.

Specifically, for synchronization density S values below 2, the highly unstructured graphs are much better transformed by Algorithm2. For small S values, the distribution of the loads across the same topology has an important impact on γ, incrementing the dispersion of the results for any technique considered (see discussion about Fig. 4.6 in section 4.1.2). For these random irregular graphs both full SP-ization algorithms offer very similar results (if not the same). Nevertheless, the Algorithm1 presents higher time complexity and could generate different results depending on the input order of the graph nodes. The second SP-ization algorithm (Algorithm2) presents interesting improvements: Its time complexity is tightly bounded, the output is always the same, and it ensures no critical path value increase for UTC graphs. Hence, we consider Algorithm1 superseded by Algorithm2 for general purposes.

We conclude that for highly regular applications, the solution obtained with a layering technique (or bulk synchronous parallelism) is similar to a nested-parallelism solution, but the layering technique computes the solution faster. For more irregular problems, nested parallelism is more appropriate and Algorithm2 may obtain better results than the Layering tecnique at only a logarithmic time complexity increase on the number of graph nodes.

Algorithm	Space	Time	UTC-cpv	Regular graph	Irregular graph
Layering	$O(m+n)$	$O(m+n)$	Yes	Good	Bad
Algorithm1	$O(m+n)$	$O(m \times n)$	No	Good	Good
Algorithm2	$O(m+n)$	$O(m+n\log n)$	Yes	Good	Good

Table 3.3: Algorithms comparison

3.6.4 Analytical models

Deriving an analytical model for the potential performance degradation, due to the loss of parallelism introduced at the high abstract level of programming, is not an easy task. We must derive approximation models for the critical path value of SP and NSP DAGs.

For SP graphs, analytical upper bounds and mean expected value of *cpv* may be derived under certain conditions. In absence of any workload information, we assume the simplified case where the load in each node is an i.i.d. (independent identically distributed) random value with a given distribution:

$$\tau(v \in V) \rightsquigarrow \mathbf{D}(\mu, \sigma)$$

In this case, we may apply *order statistics* results [95] to estimate the expected value of the parallel composition of m tasks. Results for serial composition (addition of i.i.d. variables) can be found in simple statistics literature (see e.g. [10]).

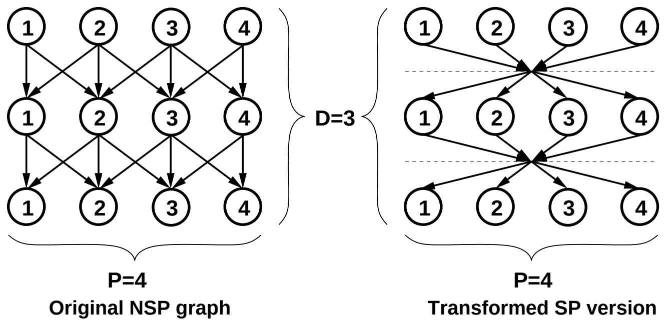


Figure 3.29: Neighbor synchronization example

Thus, the serial composition of n layers, each of them formed by m parallel tasks, is easily derived. Thus, for very simple and regular SP graphs, we can derive formulae for the expected *cpv*. However, when the parallel sections have different number of tasks, the formulae may not be so easily derived.

Unlike in SP graphs, general cost estimation is analytically intractable unless the workloads have a negative-exponential distribution [164]. However, as task workloads are close-to-normally distributed (partly as result of the Central Limit Theorem), negative-exponential workload distributions are extremely rare. Thus, a full accurate analytic cost model is not possible. We can try to derive approximations to the *cpv* of NSP DAGs. In [183] we presented an study about the approximation of the *cpv* of two example regular NSP DAGs (pipeline and neighbor synchronization structures). These examples represent the basic models of regular structures, with D layers of P tasks, connected with non-transitive edges in an almost perfect distribution of S edges per node ($S = 2$ for macro-pipeline and $S = 3$ for neighbor synchronization). See section 4.1.3 for a full characterization of this important class of graphs and applications. The SP versions of these graphs are easily obtained with the Layering technique, applying full barrier synchronizations. For these regular structures our resynchronization algorithms (Algorithm1, Algorithm2) obtain similar solutions. In Fig. 3.29 we show an example of the original NSP neighbor synchronization structure and its SP approximation. A full discussion of experiments with these regular structures is presented in section 4.1.3.

Although other random distributions may be used, in the following discussion we will assume all nodes exhibit an i.i.d. Gaussian distribution.

$$\tau(v \in V) \rightsquigarrow N(\mu, \sigma)$$

In the SP version, the formulae for the critical path value of a layer (a parallel composition of P nodes), and the full graph (series composition of D layers) are approximated by [95]:

$$\overline{cpv_P} = \mu + \sigma\sqrt{2\log(0.4)P}$$
$$\overline{cpv_{SP}} = D(\mu + \sigma\sqrt{2\log(0.4)P})$$

For normal distributions the approximation error is in the percent range.

To apply the same order statistics approach to the NSP original graph, we approximate its *cpv* with the *cpv* of a virtual core SP DAG. This virtual core is related with the synchronization density and width of the original graph. The core is composed by the same number of layers as the original graph D, synchronized by barriers; but the width of the layers differs. We compute the theoretical width of the core graph P' as a function of the original P and S parameter values. Notice that the core does not really exist, and the P' value may be a non-integer number:

$$P' = S + \log(P/2)$$

Again, order statistics are used to derive a formulae that approximates the *cpv* of the original NSP graph from the core graph:

$$\overline{cpv_{NSP}} = D(\mu + \sigma\sqrt{2\log(0.4(S + \log(P'/2)))})$$

The approximation error is now higher as a result of the core approximation of the NSP graph. Making simple substitutions we obtain a $\overline{\gamma}$ approximation that agrees with our experiments within 10% and 25%, depending on the example graph, and has similar asymptotic behavior. A coarse, but meaningful simplification of the formulae for (typical) large P values is given by:

$$\overline{\gamma} \approx \frac{\mu + \sigma\sqrt{\log(P)}}{\mu + \sigma\sqrt{\log(S)}}$$

Indeed, for graphs representing this class of regular applications, the asymptotical influence of P is clearly logarithmic, while the effect of S is inverse, which is in perfect agreement with the results presented in section 4.1.3. The effect of the workload distribution is also in agreement with our measurements (considering the typical cases where $P \gg S$).

Unfortunately, these analytic approximations may not be safely extended to any other, specifically more irregular topology, which limits the generality of the analytical study.

3.6.5 Conclusions about SP-ization impact

In this section we have propose a general measure γ, based on critical path analysis, for the potential performance impact of an SP-ization on a given graph. The SP-ization techniques studied in previous sections have been evaluated in terms of their behavior and impact on different graph classes. The study shows that our Algorithm2 is a good general-purpose SP-ization technique, only matched by the simple Layering in specific highly regular structures, where both solutions are similar, but the time complexity bound of the Layering technique is even lower. No structural impact measure, obtained only from the topology of the original and transformed graphs, has been yet found to be directly related to the *cpv* alteration, representing the modeled application performance. More complex models, based on other topological parameters (D, P, S), are more promising but still not accurate enough. Moreover, even when simple random distributions are considered for workload distributions, general analytical models for the *cpv* modification are not possible; formulae for NSP graphs *cpv* cannot be derived for stochastic workloads. Approximations for some regular structures have been presented, but they cannot be extended for any graph topology. Thus, in many cases, only experimental work may give us an idea of the impact of a transformation for given graph classes. Fortunately, experimental measures are simple (measuring *cpv* of original and transformed graphs). Nevertheless, modeling an application with a graph may be done at different levels of implementation detail with different accuracies. Predictions obtained with graph models should be compared with measures obtained with real applications to determine if general tendencies are preserved.

3.7 Summary

In this chapter we have presented a theoretical approach to the NSP vs. SP comparison problem. Application synchronization structures have been represented by graphs. Thus, we have used graph theory to formally define and study the characteristics of SP and NSP structures. Simple methods to resynchronize local NSP structures have been studied. Furthermore, algorithms to resynchronize full graphs have been presented. These algorithms try to minimize the potential loss of parallelism created by new added dependences. Our last algorithm presents interesting features (no increment of critical path for unit time cost graphs, and tighter time complexity bounds), that make it useful for experimental or production work.

We have also introduced a study about measures of the NSP to SP transformation impact in terms of structural modification of the graph, and critical path value increment. In the absence of experimental workload information, a graph should be provided with stochastic workloads. Order statistics are a useful tool for deriving the mean *cpv* of simple SP graphs, due to their compositional nature. Although similar *cpv* analysis is intractable for NSP graphs, some analytic approximations to the *cpv* modification are possible for typical regular structures. This analytic formulae predicts the asymptotical behavior of the *cpv* after a simple transformation, as a function of basic graph and workload parameters. The results, which are in agreement with experimental results presented in the next chapter, give us an idea of the general tendencies of performance when regular applications are programmed in an SP PPM. Unfortunately, this kind of analysis cannot be extended to generic, more irregular NSP graphs. As a consequence, a further experimental study is necessary to state if the predicted performance behavior for regular structures can be extended to other application classes. This study is presented in the next chapter.

The theoretical study of the NSP structures has shown serious limitations derived from their inherent complexity. SP compositional nature and limited dependences complexity present many advantages for analytical study. This is the origin of the many good properties of the SP PPMs, in terms of formal software development techniques, analyzability, and program cost modeling.

Our theoretical study of the NSP and SP task graph structures has produced interesting results and tools (like the transformation algorithms), as well as a deeper insight about the problems associated with NSP structuring. It has also provided clear directions in which way to conduct the experimental study presented in chapter 4.

Chapter 4

Experimental study

"My attention, for the last three years, had been repeatedly drawn to the subject of Mesmerism; and, about nine months ago it occurred to me, quite suddenly, that in the series of experiments made hitherto, there had been a very remarkable and most unaccountable omission."

The Facts in the Case of M. Valdemar, 1845
EDGAR ALLAN POE

In this chapter we describe the work we have carried out to experimentally measure the expected performance impact when applications are programmed in SP restricted PPMs, compared with more generic (NSP) solutions. The space of NSP graph topologies is immense and impossible to check exhaustively. Moreover, most NSP graphs do not represent any useful parallel application. Thus, we direct our search in two directions to cover the most interesting applications in parallel programming. We propose two experimental frameworks based on:

1. Synthetic graphs: We construct sets of graphs representing a random sample of the NSP graph space, and randomly interconnected regular topologies. We measure the effect of SP-ization for simple graph parameter values.

2. Empirical graphs: In this framework we focus on graphs obtained at different abstraction levels from real parallel applications, covering the relevant NSP SA classes. We are guided by the examples and classification of applications SA presented in section 2.6.

Our main interest is the overall effect of programming applications located in the NSP classes using SP models. We are first trying to establish if the performance effects found in the theoretical study are general effects, and if they can be extended to all application classes when an "ideal" transformation algorithm is used. The mean critical path analysis is our basic experimental tool to measure the performance in our graph models. An extended analysis of performance effects follows. This study covers several phases. We investigate empirical prediction mechanisms for the expected performance effects when using SP forms to represent generic NSP synchronization structures. For simple graph structures we can further study the expected performance effects of simple application modifications, as scaling up, adding more iterations, or changing local synchronization patterns when the application is in SP form. Thus, in our study we have selected simple graph parameters (see definitions of P, D, S in section 3.1.3) to study the impact of SP-ization techniques in graphs which present different topological characteristics. We experimentally relate their values with the potential and real performance loss of applications when mapped to an SP form. After study the synchronization structures of simple applications in the more abstract level, the problem of extending the study to real applications is tackled. This study includes an important methodology section about how to model applications with graphs at different detail levels, and how to transform them to SP form with our techniques, measuring the potential performance loss with critical path analysis (see section 3.6). Thus, the exploration of the SP-ization effects is open to representative graphs of more irregular application classes. Indeed, we investigate the propagation of the P, D, S predicted effects on γ, to the lower run-time level Γ, before benefits of SP programming are exploited. We also research the effects of load balancing and other common parallel programming techniques for irregular applications when an SP programming framework is used. We compare results obtained in more abstract levels, with performance measures of the equivalent real applications, running in different parallel architectures.

4.1 Synthetic graphs

In this section we present the first experimental framework. This part of the study is oriented to evaluate the mean performance effects of our "ideal" SP-ization transformation on random, irregular topologies, representing a sample of the whole graph space. We test if the γ tendencies related to the simple graph parameters P, D, S derived from the theoretical study (see section 3.6), are general effects found in generic graphs.

The experiments are based on constructing sets of synthetic DAG topologies, generate different synthetic workload distributions for the nodes, and compare the *cpv* in the original graph with the *cpv* of an SP approximation generated with a suitable SP-ization technique. After the experiments we relate γ measurements to topology and workload characteristics.

The phases of each experiment may be summarized as:

1. Generate a synthetic topology $G(V, E)$.
2. Transform G to SP form: $G' = T(G)$. (We apply Algorithm2, which uses no workload but only topological information).
3. Repeat:
   (a) Generate a synthetic workload distribution for the nodes in the original graph:
   $$\tau(v), v \in V$$
   (b) Copy the same workload information to the transformed graph. Nodes introduced by the transformation have zero load:
   $$\tau'(v \in V') = \begin{cases} \tau(v) & \text{if } v \in V \\ 0 & \text{if } v \notin V \end{cases}$$
   (c) Compute comparison indicator:
   $$\gamma = \frac{cpv(G')}{cpv(G)}$$

In the following sections we present techniques to generate synthetic workloads and topologies. Different topology sets are presented and analyzed.

4.1.1 Workload modeling

Synthetic workloads must be supplied for the nodes in the generated graphs. No specific patterns or regularities between topology and distributions should be used in this part of the experimental framework. Thus, the fairest assumption is to consider each node workload $\tau(v)$ to be an i.i.d. (independent identically distributed) random variable. Considering that we will use graphs with big number of nodes, we will assume Gaussian distributions for the workloads:

$$\tau(v \in V) \rightsquigarrow N(\mu, \sigma)$$

The relative increment of the critical path value is not affected by proportional modifications of the mean and deviation parameters. Consider the example graphs in Fig. 4.1. G' is an SP approximation for G. The number inside each node represent the workload $\tau(v)$ of that node. The new grey node in G' has been introduced by the transformation technique. Thus, it is only a synchronization point with no load $\tau(v) = 0$. For the loads in the example we obtain the following mean and deviation values: $\overline{x} = 1.1667, s_{n-1} = 3.1047$. The critical path values are 4 and 5 for G and G' respectively. Thus, the relative increase of the critical path is $\gamma_\tau = \frac{cpv(G')}{cpv(G)} = 1.25$. Consider now the same graphs, but assume

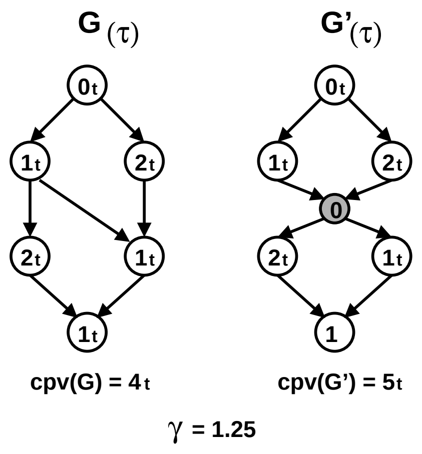


Figure 4.1: Example of relative critical path value increase

a workload distribution where $\tau'(v) = \tau(v) \times 2$. The mean and deviation are now doubled: $\overline{x} = 2.3333, s_{n-1} = 6.2093$. The critical path values will be 8 and 10 for G and G' respectively. The relative increment is the same: $\gamma_{\tau'} = 1.25$. This example illustrates that the exact values of the workload distribution parameters μ and σ are not so important on themselves. Their ratio is much more relevant. Thus, we define a unique parameter for task workload variability:

Definition 4.1.1 *We define the* relative deviation *or* variability *(ς) of a random workload distribution as the proportion between the deviation and the mean:*

$$\varsigma = \frac{\sigma}{\mu}$$

For our experiments we decide to generate different workload distributions based on different ς values, representing from well-balanced computations to highly irregular workloads:

$$\varsigma \in \{0.1, 0.2, 0.5, 1\}$$

For simplicity, we always fix the mean to a constant and change the deviation accordingly to selected ς values. To make the result analysis more intuitive, we choose 1 as the fixed constant mean, being the corresponding final deviations equal to the chosen variabilities:

$$\mu = 1; \sigma \in \{0.1, 0.2, 0.5, 1\}$$

For each generated topology and each ς value, we draw 25 random workload distributions (with 25 different seeds for reproducibility of experiments). The critical path is measured in both topologies, G and G', with each workload, and mean γ computed.

4.1.2 Random sample of the graph space

In the first experiment sets we test a sample of random task graphs, with no specific topological restrictions, to obtain an idea of the general trends of SP-ization effect in performance.

Most DAGs in the graph space do not represent typical parallel applications (scalable computations with replicated patterns), but irregular structures that can only be generated by the most unstructured, dynamic and data dependent programs. Our experiments will show general trends that will be improved when more realistic topologies are studied (see following sections).

Random topology generation technique

To sample the NSP topology space we want to generate graphs with similar probabilities for any topology to be selected. After considering several methods, we have chosen a standard task graph generation technique originally devised for graphs representing heterogeneous parallel applications [7, 181]. In this technique, every possible edge has the same probability to exist in the graph. To assure that a DAG is generated, the nodes are numbered, and only edges with a source node number lower than the target number are considered.

Formally, let $V = \{v_1, v_2, ..., v_n\}$ be the set of nodes in G and p the edge probability factor. Then, this technique produce edges in the graph with the following probabilities P:

$$\begin{array}{ll} P[(v_i, v_j) \in E] = p, & \textit{if}\, 1 \leq i < j \leq n \\ P[(v_i, v_j) \notin E] = (1-p), & \textit{if}\, 1 \leq i < j \leq n \\ P[(v_i, v_j) \notin E] = 1, & \textit{if}\, i \geq j \end{array}$$

The parameter p will let us direct the search of the whole DAG space along the edge density axis (measured by the synchronization density S). For a given p, the mean number of predecessors/successors becomes larger with the number of nodes in the graph. However, the maximum number of edges for a given n is $n(n-1)/2$. Thus, we can select p as a function of n to generate graphs with approximately the same synchronization density independently of the size:

$$p = \frac{nS}{n(n-1)/2}$$

The complexity bounds of this generation technique is related to the graph size. This technique traverses all possible edges in the graph, checking randomly if the edge is or is not added to the graph. Thus, the time complexity of the technique is $\Theta(n(n-1)/2)$. It uses only the space needed to store the graph.

This technique may generate non-connected graphs, especially for low p values. Recall in section 3.3.1 that SP-ization techniques work on $STDAG$ graphs.

We use the technique presented in definition 3.1.17 to build a 2-terminal DAG, possibly adding two new synchronization nodes, to connect the generated graph. The original disconnected subgraphs are then parallel sections of the final $STDAG$.

Chosen parameter values

We generate graphs for a wide range of node numbers. From small ones (32 nodes) to big ones (1024 nodes):

$$n \in \{32, 64, 128, 256, 512, 1024\}$$

For each size, we want to test topologies ranging from very low to very high synchronization densities. The maximum synchronization density is limited by the graph size. For small graphs, the highest S values are to be discarded.

$$S \in \{0.5, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 5, 7.5, 10, 25, 50, 100\}$$

For a given pair of (n, S) values we compute p and generate 100 topologies based on a set of 100 seeds in order to guarantee reproducibility of experiments. Thus, more than 1000 topologies are generated for each graph size.

Results

In this section we present remarks obtained from results observation. Except when it is otherwise stated, the points in the plots represent the $\overline{\gamma}$ for all the topologies which x axis parameter is in a narrow histogram slot. They are drawn as curves to show tendencies, and for clarity when several curves are drawn in the same plot.

1. General under-logarithmic effect related to graph size:

   In Fig. 4.2 we show the general under-logarithmic γ tendency on the number of nodes. This tendency is similar to the one predicted with the theoretical approach in section 3.6. Nevertheless, each point of these curves represents the mean values of γ for hundreds of graphs with very different shapes, leading to high deviations. A more detailed study is needed. We want to know if, as in regular structures, this tendency is specifically derived from P and S parameters. And if it is possible for a given graph size, to more accurately predict the γ values as a function of P, D, S or related parameters.

2. Topological parameters dependence on S:

   As we show in Fig. 4.3, in these irregular random topologies, the P and D parameters are highly correlated with S. If S is low, many nodes or

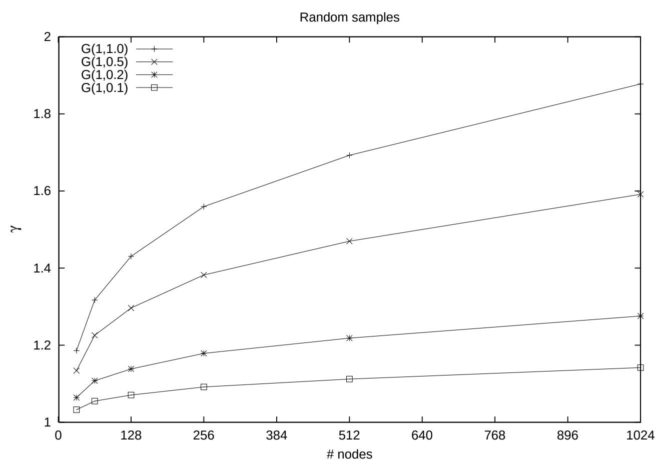


Figure 4.2: General γ tendency on graph size

subgraphs are disconnected after the first stage of the generation technique is applied. Thus, they are parallel sections of the constructed *STDAG* (high P and low D). As S increases, the probability of more nodes and subgraphs to be serialized is higher (low P and high D). Thus, the research is focused to the synchronization density related parameters.

In these examples we measure the parameter S after transforming the generated DAGs to connected *STDAGs* (the graph that is actually transformed). Although S is similar to the original edge density, it is slightly modified due to added edges when connecting the graph in an *STDAG* form. For very low values of edge density, many edges are added to connect the highly sparse generated graphs.

3. Correlation with ς (workload model):

The plots in Fig. 4.4 show γ values obtained for medium *(a)* to big *(b)* sized random graphs transformed with Algorithm2. Each curve on the same plot corresponds to a different workload model, with ς values from unbalanced computations $\varsigma = 1$ to highly balanced computations $\varsigma = 0.1$. The workload balance is a basic factor for the impact of SP-ization. Low values of ς minimize the impact of SP-izations because accumulated path values are very similar along the graph. Thus, new synchronizations have few probabilities of serialize parts of two highly different loaded paths. For random workload models with high ς, unbalanced task loads are spread randomly across the whole graph. Thus, added dependences may serialize highly unbalanced accumulated loads, modifying the critical paths and increasing

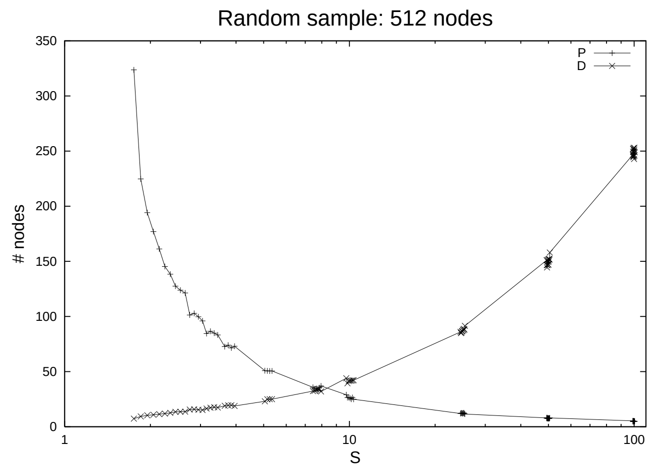


Figure 4.3: Dependence of topological parameters on S

their values.

In the plots, it can also be appreciated how the Algorithm2 SP-ization offers good solutions for graphs with an S value lesser than 2 (see also section 3.6.3).

4. General dependence on R_s:

   The plots in Fig. 4.4*(a)* and *(b)* come from graphs with different number of nodes. The curves obtained for different sized graphs with the same workload model, differ not in the shape, but in the slope. We use the parameter $R_s = |E|/|V|^2$, that measures the relative number of edges in a graph of $|V|$ nodes, to predict the behavior of γ more independently of the graph size. In Fig. 4.5 we present smoothed curves for mean γ relative to R_s, for all graph sizes tested and normal workload distribution ($\varsigma = 1, N(1,1)$). Curves drop to the left due to the improved results obtained with Algorithm2 for S values below 2. For bigger graph sizes, the R_s point that correspond to $S = 2$ is lower. Thus, the maximum γ value for a given graph size is found approximately in a value of $R_s = 2/|V|$.

5. Maximum dispersion of values around $S = 2$. Less predictability:

   In Fig. 4.6 we show one point for the γ value of each different topology (mean of 25 different workloads). As we may appreciate, the maximum dispersion of the points is found around an S value of 2, where the $\overline{\gamma}$ values are also the highest. This indicates that our predictions based on $\overline{\gamma}$ values are less accurate for the topologies with S values near 2. Topological structures with $S \approx 2$ present many different ways to be transformed to

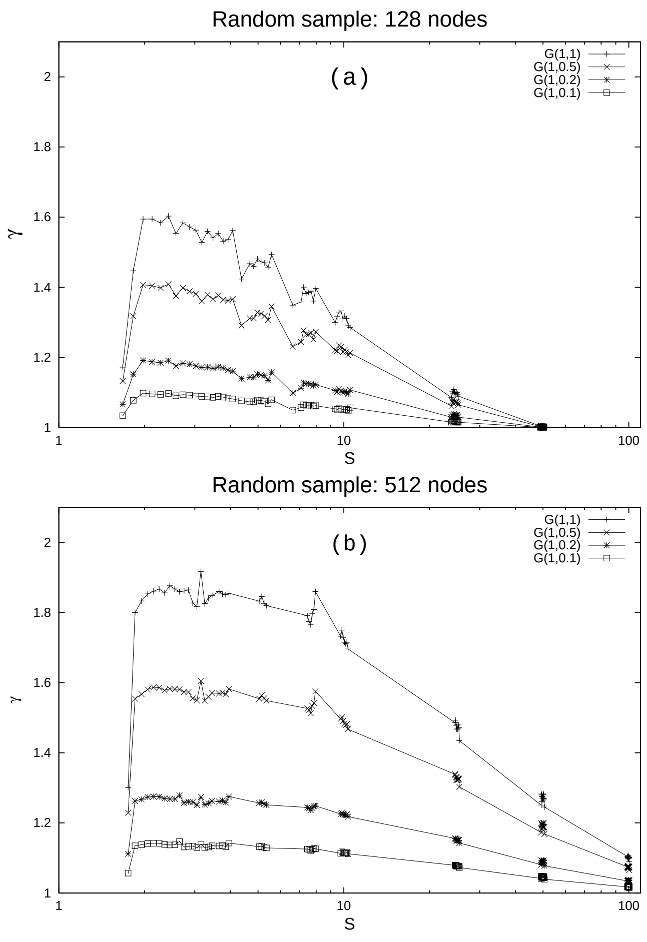


Figure 4.4: Dependence of γ on S and ς

SP form. Only algorithms that make use of the workload information on the nodes may find the best topological transformation. In the class of algorithms which work without workload information, our Algorithm2 finds a compromise solution by preserving the *cpv* for UTC graphs and looking for local synchronizations where possible.

For low deviations ($\varsigma = 0.1$), the dispersion trend is the same, although less noticeable than for high deviations ($\varsigma = 1$). The reason is the increased probability of regular workload distribution across the topology. The higher the relative balance of the workload, the lower expected γ values and the higher accuracy of our predictions.

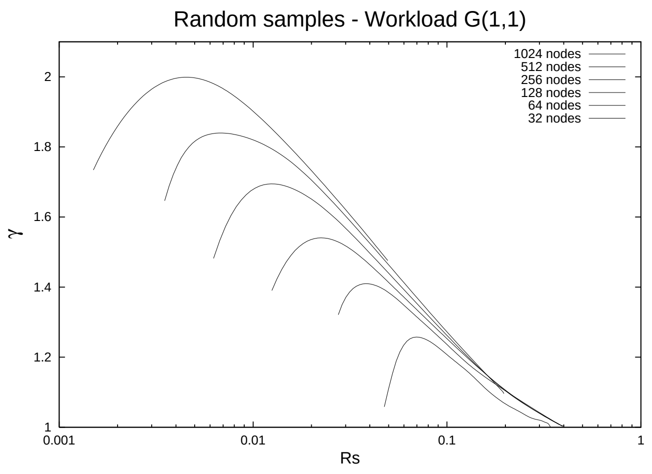


Figure 4.5: General trend dependence on R_s

We conclude that for random graphs with n nodes, and a complete random workload distribution, the general trends of SP-ization impact at programming abstraction level (γ) may be predicted depending only in basic parameters: Topological (graph sizes $|V|$ and $|E|$ to compute R_s) and workload based (ς). Predictions are more accurate the further the S parameter is from value 2.

4.1.3 Meshes

Most random topologies may represent highly dynamic or even no real parallel application at all. However, parallel application design methods and paradigms tend to produce topology and workload regularities to exploit program scalability. A typical parallel program is designed in a way that increasing the number of processors more similar parallel tasks are executed to compute a smaller part of the result. Many tasks represent running instances of the same code pieces, working on different data. Thus, a high correlation between the execution time of tasks and their topology role is found in most parallel applications. Unfortunately, it is not possible to realize such a correlation only from the task graph topology. Nevertheless, many applications present, after mapping, high regular structures that replicate computation layers, as wide in tasks number as processors are available.

Consequently, we introduce a new collection of experiment sets based in graph meshes of tasks, organized in equal sized layers, connected by random and replicated synchronization patterns. Motivation for the importance of these structures is found in most applications inside the (NSP,NME,NDS) SA class (see section 2.6).

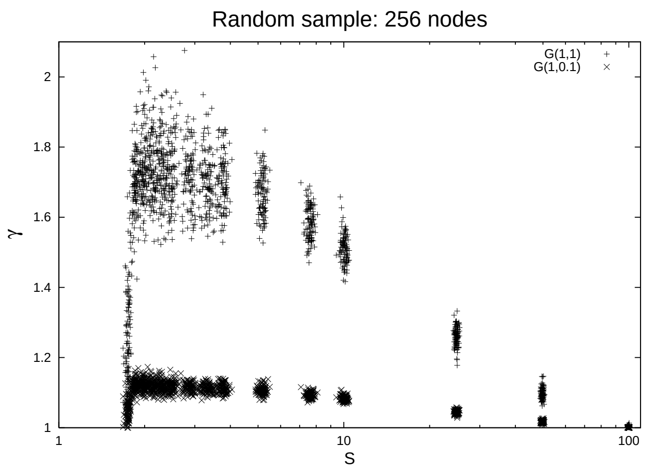


Figure 4.6: Results dispersion for random graphs

Meshes: definitions and notations

We introduce here the definitions and notations needed to understand how to build a synthetic graph mesh from simple parameters.

Definition 4.1.2 *We define a* Mesh *to be a DAG built by a collection of D ordered and numbered subsets of nodes (call layers) of equal size P, with edges only between consecutive layers:*

$$\begin{aligned} M = (V, E): \quad & L = \mathcal{P}V; |L| = D; |L_i| = P, \forall i = 1, ..., D; \\ & \forall (v, w) \in E; v \in L_i, w \in L_{i+1} \end{aligned}$$

Mesh sizes are defined by P (layer size) and D (number of layers) parameters directly. The edges of a mesh will be defined by a function that maps a node j in a layer i to nodes $j_1, j_2, ..., j_n$ in layer $i + 1$. Both, random and deterministic functions are possible.

Definition 4.1.3 *Let $M = (V, E)$ be a mesh, $L = \mathcal{P}V$ the layers of the mesh, such that $|L_i| = P : i = 1, 2, ..., D$. Let $\xi : v \in L_i \mapsto \mathbb{N}; \xi(v) \in [1, P]$ be a numbering of the nodes in a layer. We define a* Synchronization Function (ρ) *as:*

$$\rho : \xi \to \xi^a; a \in [1, P]$$

This function defines the set of edges between each consecutive pair of layers in the mesh:

$$E = \{(v, w) : v \in L_i, w \in L_{i+1}; \xi(w) \in \rho(\xi(v))\}$$

In Fig. 4.7 we show an example of a mesh generated by $P = 4, D = 3$ and a deterministic ρ, different for each $\xi(v)$.

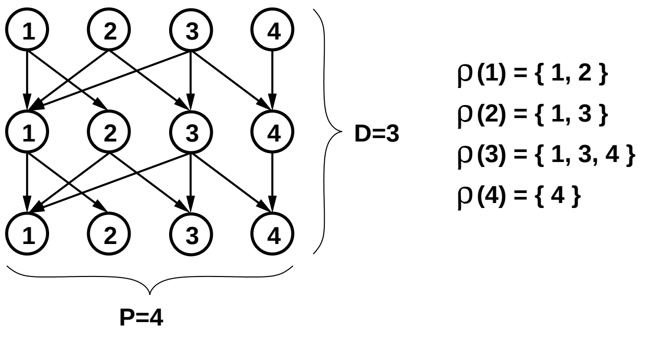


Figure 4.7: Example of a mesh graph defined by P, D, ρ

Topological parameters

Synthetic meshes may be constructed depending on D, P and ρ. The first two parameters define the graph sizes and the third the interconnection pattern. The synchronization density S is equal to the mean arity of the synchronization function. All these three parameters may be modified while the others are fixed. Thus, we can explore the effect of each one independently of the others.

In meshes, the edges have source and target nodes in consecutive layers. Thus, a mesh have no transitive edges: $(v, w) \in E \Rightarrow \nexists h \in V : v \preceq h \preceq w$. The consequence is that graph meshes are equal to their transitive reductions $M = M^{-}$, and S parameter is a highly reliable indicator of the amount of dependences propagated through a node, layer by layer.

Random meshes

In our first set of experiments with meshes we want to check the effect of P, D, S parameters on γ, for random synchronizations between layers. The random mesh generation technique chosen is based on creating the same number of outgoing edges for each node [181]. The number of edges per node is determined by the value of S parameter. The successors will be randomly selected among all nodes in next layer, based on an uniform random distribution $U[1, P]$.

To assure connectivity in the graph and a correct layer organization (nodes in the same layer must have the same depth level), the first outgoing edge for any node will be the edge $(v, w) : \xi(v) = \xi(w)$. Only $S - 1$ edges will be randomly selected. When S is not an integer, we create edges such that all nodes has $\lfloor S \rfloor$ or $\lceil S \rceil$ outgoing edges, and the mean number per node in the layer is as approximated to S as possible.

Formally, the procedure to create random meshes may be described as follows: Let A be the set of node numbers in a layer, and B a random subset of A with the cardinality:

$$\begin{array}{l} A = [1, P] \subset \mathbb{N}; \\ B \subset A : |B| = \lfloor P \times (S - \lfloor S \rfloor) \rfloor \end{array}$$

The synchronization function selected to build random meshes is:

$$\rho(\xi(v)) = \{\xi(v)\} \cup \{c_i \rightsquigarrow U[1, P], i = 1, ..., s\}$$

$$s = \begin{cases} \lfloor S - 1 \rfloor & \text{if } \xi(v) \notin B \\ \lceil S - 1 \rceil & \text{if } \xi(v) \in B \end{cases}$$

For example, let us suppose a mesh with $P = 10$ and $S = 2.36$. The cardinality of the B set will be $|B| = \lfloor 10 \times (2.36 - 2) \rfloor = \lfloor 3.6 \rfloor = 3$. Let us suppose that B set is randomly selected to be $|B| = \{4, 8, 9\}$. The s value, that represents the number of randomly chosen edges for a node, is computed as:

$$s = \begin{cases} 1 & \text{if } \xi(v) \in \{1, 2, 3, 5, 6, 7, 10\} \\ 2 & \text{if } \xi(v) \in \{4, 8, 9\} \end{cases}$$

Thus, all nodes will have one predetermined edge $(\xi(v), \xi(v))$, seven of them will have one random edge and three of them will have two random edges $(\xi(v), U[1, P])$. There will be 23 edges between each layer. The final synchronization density for one layer will be $S = 23/10 = 2.3 \approx 2.36$

Chosen parameters

We experimentally test sets of synthetic topologies with up to thousand nodes with the following parameter values and motivation:

1. Square meshes, to detect the effect of S alone, for a given graph size:

$$\begin{array}{l} (P, D) \in \{(8, 8), (16, 16), (24, 24), (32, 32)\} \\ S \in \{1.1, 1.2, 1.4, 1.6, 1.8, 2.0, 2.5, 3.0, 3.5, 5.0, 7.5, 10, 25\} \end{array}$$

2. Fixed P, to detect the effect of D:

$$\begin{array}{l} P = 16 \\ D \in \{4, 8, 16, 24, 32, 64\} \\ S \in \{1.1, 1.2, 1.4, 1.6, 1.8, 2.0, 2.5, 3.0, 4.0, 5.0, 8.0, 12.0\} \end{array}$$

3. Fixed D, to detect the effect of P:

$$\begin{array}{l} P \in \{4, 8, 16, 24, 32, 64\} \\ D = 16 \\ S \in \{1.1, 1.2, 1.4, 1.6, 1.8, 2.0, 2.5, 3.0, 4.0, 8.0, 12.0, 16.0, 24.0, 32.0, 48.0\} \end{array}$$

The Layering and the improved Algorithm2 transformation techniques obtain similar results for $S > 2$. In Fig. 4.8 we show an example of how both transformation techniques obtain similar results when the high synchronization density prevents our algorithm to create small local synchronizations, but forces full barriers between layers. This effect always appears for S values higher than 2. Thus, for these kind of graphs we can use the faster Layering technique safely. We extend our study to huge graphs with up to hundred thousand nodes, that can be manipulated in reasonable time with the Layering transformation technique:

1. Square meshes, to detect the effect of S alone:

   $$(P, D) \in \{(100, 100)\}$$
   $$S \in \{2, 3, 4, 5, 10, 20, 30, ..., 100\}$$

2. Fixed P, to detect the effect of D:

   $$P = 100$$
   $$D \in \{10, 25, 50, 75, 100, 200, 300, ..., 1000\}$$
   $$S \in \{2, 3\}$$

3. Fixed D, to detect the effect of P:

   $$P \in \{10, 25, 50, 75, 100, 200, 300, ..., 1000\}$$
   $$D = 100$$
   $$S \in \{2, 3\}$$

In all cases the workload distributions are computed as described in 4.1.1.

Results

The experiments show the following results:

1. Decreasing impact for higher S values:

   The effect of high synchronization density values ($S > 2$), is similar as discussed for random topologies in section 4.1.2. In Fig. 4.8 we show this effect for different values of S in a 16×16 random mesh. Increasing values of S indicate more dependences already in the graph and shorter distance to an SP form. Thus, the impact of SP-ization is quickly diminished when S increases.

   In complete random topologies (see section 4.1.2), P and D presented a correlation with S due to the random sampling technique. Specifically, values lower than 2 indicated few layers and a collection of sparse nodes. Thus, the graph distance to SP form was short and γ was quickly decreasing with

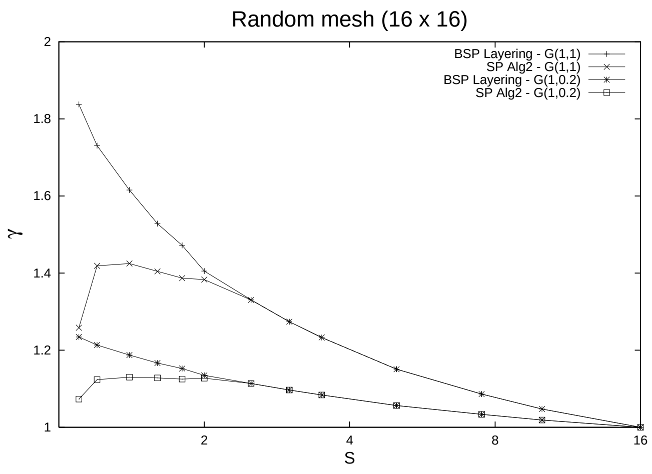


Figure 4.8: Effect of high S values in random meshes

smaller values of S. However, in meshes we are fixing P and D parameters, and changing S independently. In Fig. 4.8 and Fig. 4.9 we may appreciate the different behavior obtained with Layering and Algorithm2 transformation techniques for random meshes. When Layering is applied, γ continues the same exponential like increasing tendency for very small S values. The application of our improved Algorithm2 transformation technique cancels the exponential growing tendency, and it achieves even decreasing results for low deviated load distributions. However, it does not achieve the high diminishing effects like in random topologies. In the plots of Fig. 4.9, we observe the γ decreasing effects only for very small values of S and especially for low P values. The reason is the small distance from these graphs to SP forms. Recall the random meshes generation technique used. It creates a base SP mesh graph with $S = 1$ and adds extra randomly chosen edges. The number of added edges for each layer is an integer number computed as: $\lfloor P \times (S - 1) \rfloor$. For small values of the parameters very few edges or even no extra edges are added to the base SP graph, leading to γ values close to or even 1.

2. No applicability of R_s parameter alone:

   A side effect of the previous discussion is that R_s is not a good indicator of the potential impact of an SP-ization in a random mesh. In random graphs P and D were related to S. In random meshes this is not true. Thus, γ values are different for the same value of R_s if P and D values differ. Only very general tendencies may be determined using the parameter R_s alone. We must further explore the effects of D, P parameters independently.

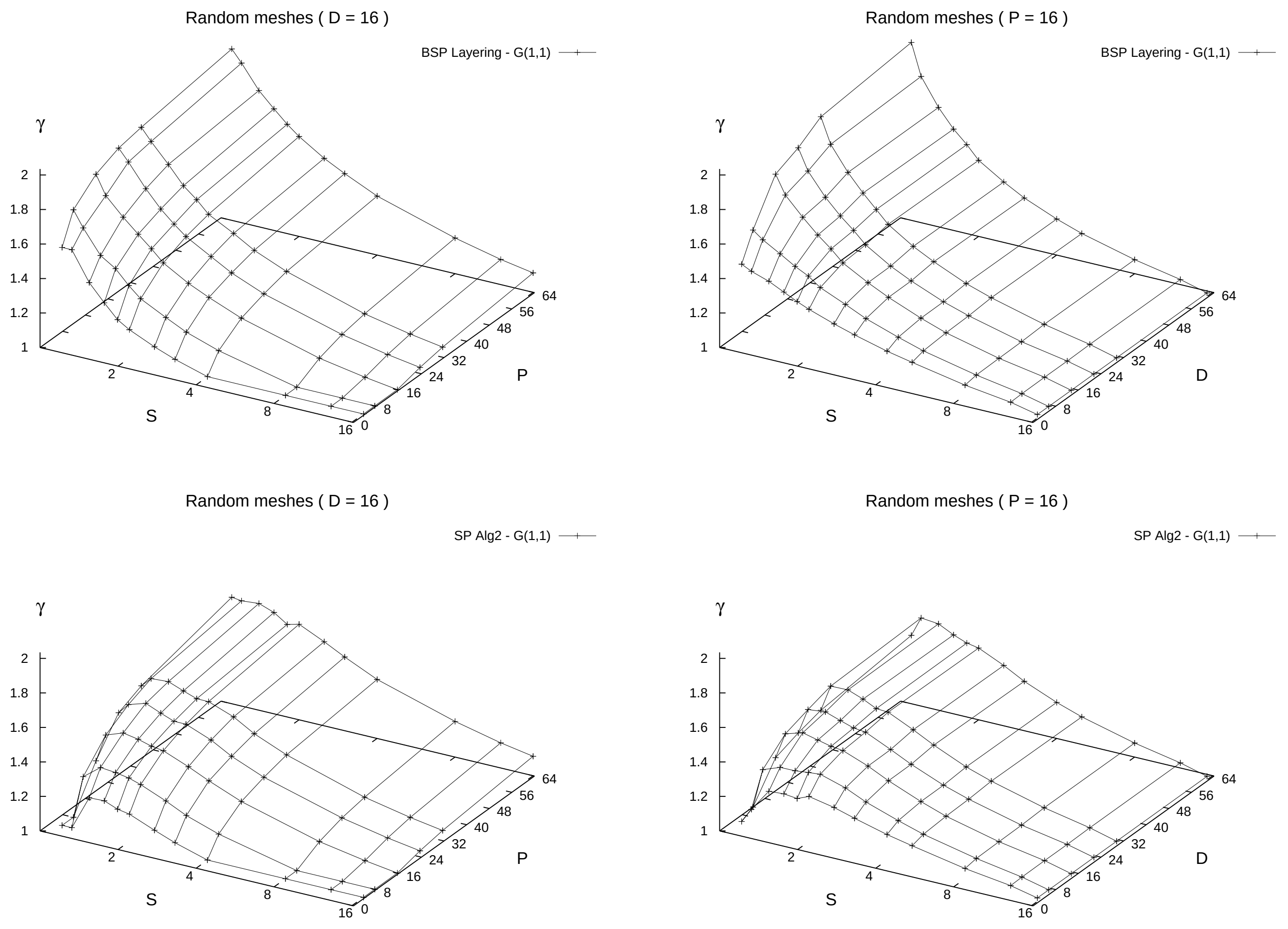


Figure 4.9: γ dependence on S,P and D in random meshes

3. Limited effect of D:

   In Fig. 4.10 we can see how γ stops to grow at a certain value of D. Although difficult to appreciate for small sized graphs, it can be also noticed in Fig. 4.9. Let us consider the i_{th}-node in layer j. Dependences from this

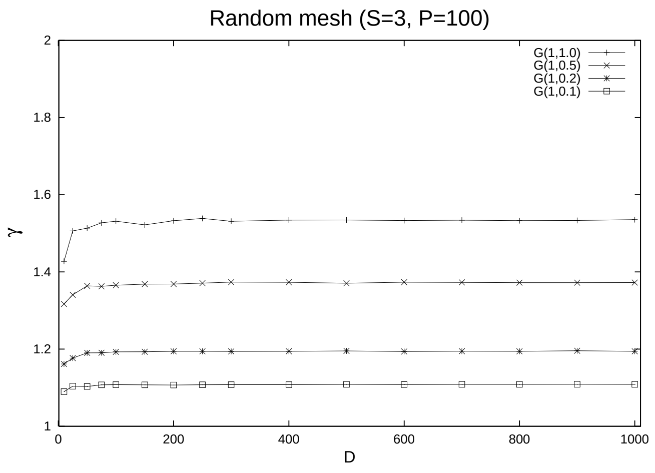


Figure 4.10: Limited effect of D in random meshes

   node are propagated to other nodes across layers $j+1, j+2, ...$, until all nodes in a further layer $j+c$ depends on the original node. At this point, an SP-ization technique is not adding dependences from the i_{th}-node of layer j to any other node in further layers $j+d, d>c$, because all of them were already dependent on it. The speed by which dependences are propagated to next layers is dependent on S. The number of nodes in a layer is P. The limiting effect should completely appear for $D > P/(S-1)$. The observations show that in general it appears even before. In the original NSP graph, the number of dependences propagated from the i_{th}-node in layer j to other nodes in layers $j+1, j+2, ...$, is growing through each layer. Thus, the diminishing effect is beginning to work since layer $j+2$, reaching the maximum at layer $j+c$.

   This limiting effect is canceled in special cases of unbalanced synchronization structures described and discussed below.

4. Logarithmic like effect of P:

   In Fig. 4.11 (and also in Fig. 4.9 in a smaller scale) we may appreciate that for fixed D and S values, the SP-ization impact increases with a logarithmic like function of P. This effect presents similar slopes for all mesh topologies with the same S value, and a D value enough to achieve its limiting effect

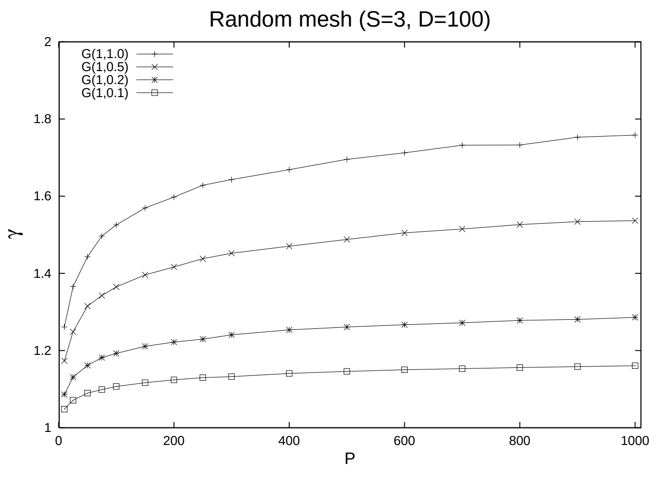


Figure 4.11: Exponential like effect of P in random meshes

(see previous discussion about D effect). The slopes are lower for examples with higher values of S, as expected.

5. Workload effect:

   As it may be appreciated in Fig. 4.10 and Fig. 4.11, the most relevant factor for the SP-ization impact is the variability of the workload ς. Graphs representing balanced computations ($\varsigma \leq 0.2$) present almost no relevant effect when transformed to SP form. When computations are highly unbalanced ($\varsigma = 1$), the probability of serializing highly loaded nodes during the SP-ization increases. The effect is highly predictable when the loads are randomly distributed, as the probabilities increase with equal chances across the same topology.

Unbalanced synchronization meshes

Motivated by the study of strange γ effects in specific application mesh topologies (as e.g. static macro-pipelines, see section 4.2), we have found a new topological characteristic, with an important impact on γ. This characteristic is not directly related with the parameters we have studied previously. This study reveals more details about the deep relation of γ and the way dependences are propagated across layers through the edges.

The problem appears in meshes were the edges are somehow *oriented in the width axis*, such that dependences from some nodes are not propagated to any other part of the graph equally. Let us consider the example in Fig. 4.12. The nodes in the right side of the graph do not propagate dependences to the left

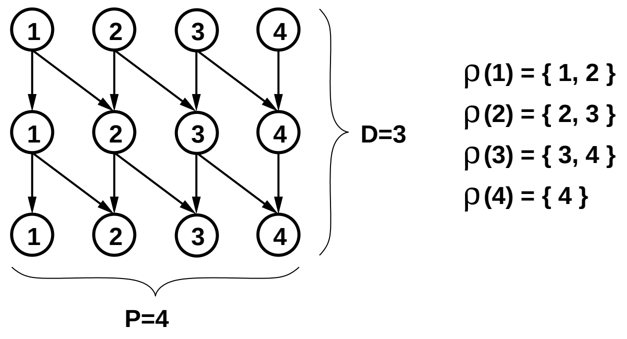


Figure 4.12: Example of unbalanced synchronization mesh

side of the graph, no matter how many layers are considered. This *orientation* is graphically dependent on the numbering ξ chosen. We must introduce some more notation and terminology to formally characterize this new problem. Because this *orientation effect* barely appears along several layers in random generated meshes, we focus our study to analytical measurements in meshes with deterministic ρ functions.

Definition 4.1.4 *For meshes with a deterministic ρ function, we define the* Synchronization characteristic graph *of a mesh $\Omega(M)$ as a directed graph (possibly cyclic), build as:*

$$\begin{aligned}\Omega(M) &= (V_\Omega, E_\Omega) : \\ V_\Omega &= L_i \\ E_\Omega &= \{(v,w) : v, w \in L_i, \xi(w) = \rho(\xi(v))\}\end{aligned}$$

For deterministic ρ functions, the synchronization characteristic graph is unique for a given P value, and a change on the nodes numbering function ξ, will produce an equivalent homeomorphic graph. An example of the Ω graph for the example in Fig. 4.12 is shown in Fig. 4.13.

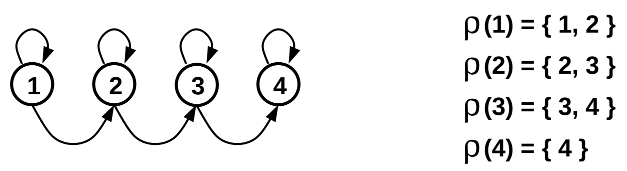


Figure 4.13: Example of synchronization characteristic graph

When the synchronization characteristic graph of a mesh is disconnected, it indicates that two different subgraphs are composed in parallel. Each subgraph should be studied separately. The Algorithm2 transformation technique detects the connected components as local NSP problem combinations and synchronize them separately. However, Layering technique would resynchronize both subgraphs together with full barriers in a non-efficient way. For connected Ω graph we study the presence of nodes that cannot be reached from other nodes.

Definition 4.1.5 *We denote by* synchronization balance, *$\omega(M)$, the proportion of edges found in the transitive closure of the synchronization characteristic graph of a mesh M. Let be $\Omega(M)^+ = (V_\Omega, E_\Omega^+)$ be the transitive closure of $\Omega(M)$:*

$$\omega(M) = |E_\Omega^+|/|V_\Omega|^2$$

This value, that will be in the range $\omega(M) \in [0,1]$, indicates the proportion of nodes that are propagating dependences to other nodes independently of the number of layers traversed. The value 0 is only possible for completely disconnected layers. The value 1 is found in graphs were all nodes can be reached from all other nodes. In Fig. 4.14 we show the transitive closure and the synchro-

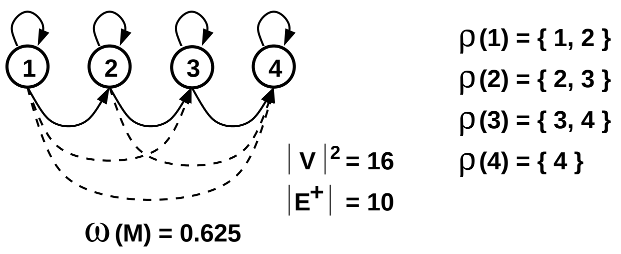


Figure 4.14: Example of $\omega(M)$ measure with the Ω graph

nization balance value for the previous example mesh. A value of $\omega(M) = 0.625$ indicates that many nodes cannot be reached from other nodes independently of the number of layers considered.

Meshes with connected Ω graphs and ω values of 1, do not present any γ effect different from the ones previously discussed, based on the topological (P, D, S) and workload (ς) parameters. However, meshes with connected Ω graphs and lower than 1 synchronization balance values, will suffer the following pathological effects:

1. Limited effect of S parameter:

   If we add edges to a mesh, that do not increase the synchronization balance, the synchronization density increases, also the number of dependences propagated, but not the number of nodes that are not reached from other certain nodes. Thus, the beneficial effect of these added edges is highly limited.

To test this effect, we have designed an experiment in which we produce meshes with increasing S values, but forcing the new edges to target neighbors already reachable in the Ω graph through transitive dependences. We define the following synchronization function for a given P and a new parameter s:

$$\rho(\xi(v)) = \{t : \xi(v) < t \leq (\xi(v) + s) \leq P\}$$

An example of meshes generated by this technique are shown in Fig. 4.15.

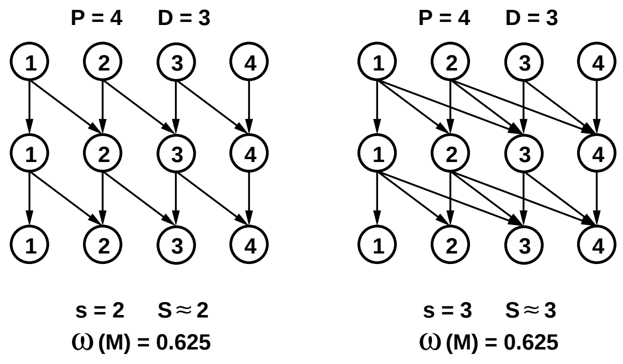


Figure 4.15: Example of meshes with higher S and the same ω

The s parameter is very similar to the final S of the generated mesh, especially when $s \ll P$. As we are interested in the effects produced for S ranging from 2 up, for our experiments we will use $P = 100$, considering $S = s$.

In Fig. 4.16 we show how increasing the number of edges (indicated by the S parameter) in a complete unbalanced mesh (plot *(a)*), does not produce the beneficial negative exponential-like decreasing effect on γ, found in random and typically balanced meshes of the same sizes (plot *(b)*). The effect is canceled after adding approximately 4 or 5 edges (the dependences are quickly propagated in the only possible direction).

2. Non-limited effect of D parameter:

   In Fig. 4.17*(a,b)* we show the effect of D increase, for unbalanced meshes. We present two examples. Both of them have been created with the previous discussed technique. They are structures with unbalanced neighbor edges with $P = 100$, and $s = 3$ and $s = 5$ respectively. Both graphs have the same number of non-reachable nodes, $\omega = 0.505$. The plots show how the limited effect of D, found for other graphs with $\omega = 1$, (compare with Fig. 4.10) does not appear. As ω value is the same, the final trend for high D values is the same. What changes from $S = 3$ to $S = 5$

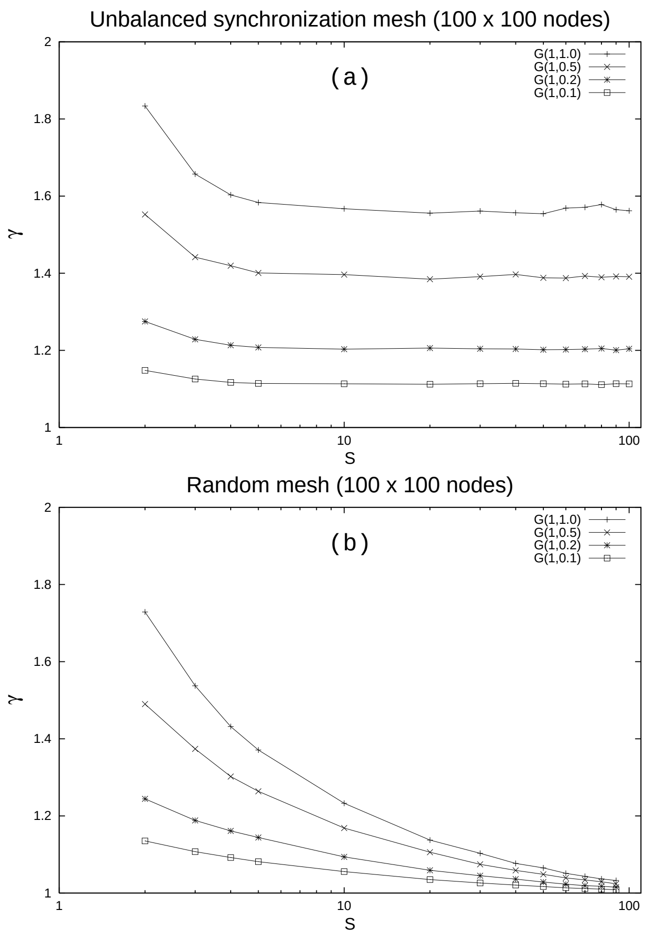


Figure 4.16: Limited effect of S in synchronization unbalanced meshes

is how quick the dependences are propagated in the only available direction. Thus, S still measures how quick the general trend imposed by ω is achieved. In both cases, we observe some irregularities in the slope near the point $D \approx P/S$. At this point, the propagated dependences have been spread along the full layer width, and the limiting D effect curve meets the general tendency curve imposed by ω. From this point on, both curves $(S = 2, S = 3)$ are similar.

With values of S lower than 2, the Ω graph is typically disconnected, and the subgraphs should be studied separately. For connected Ω graphs, the ω lower values correspond to graphs with $S \approx 2$. We conjecture that the extra dispersion of γ values related to S near 2, observed previously, is produced by

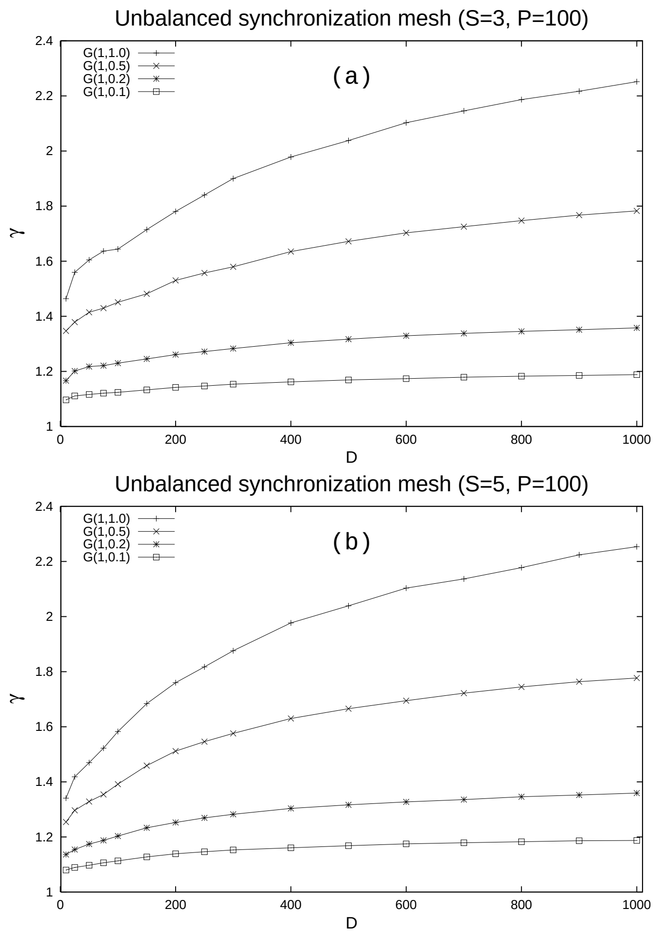


Figure 4.17: Non-limited effect of D in synchronization unbalanced meshes

these pathological effects not previously accounted for.

The pathological effects are higher for lower values of ω, although no direct relation has been yet established, because of the difficulties found to automatically generate different synchronization unbalanced topologies with the desired ω values. It is an open question if $D, P, S, \varsigma, \omega$ parameters are enough to accurately estimate γ for graph meshes.

Correlated workload meshes

In the previous study, due to the absence of real workload information for synthetic graphs, we are assuming an i.i.d. workload for every node. In real applications with not completely regular tasks loads, it is typical to find some kind of correlation between the workload distribution and the topology. Consider, for example, a mesh representing a macro-pipeline. If one of the pipe stages is more time consuming than others, we will find a *column* of tasks more loaded than the others. If a cellular-automata like program needs to compute some more complex intermediate results after some normal iterations, we will find a mesh were some *rows* or layers of nodes are more loaded than the others.

To detect if the presence of this correlation between workload and topology is beneficial or negative for the SP-ization impact, we have designed some more experiments with random meshes. We will consider meshes with fixed P, D and S values, and we will change the workloads to create such vertical or horizontal correlations. The modified load parameters μ, σ will be proportional to the original ones to keep the same variability across the whole graph.

Let us consider the following workload models:

Vertical correlation: The load is modified in a given column c in a given proportion p:

$$\tau(v) = \begin{cases} x \leadsto N(\mu,\sigma) & \text{if } \xi(v) \neq c \\ x \leadsto N(p\mu, p\sigma) & \text{if } \xi(v) = c \end{cases}$$

Horizontal correlation: The load is modified in a given layer r in a given proportion p:

$$\tau(v) = \begin{cases} x \leadsto N(\mu,\sigma) & \text{if } d(v) \neq r \\ x \leadsto N(p\mu, p\sigma) & \text{if } d(v) = r \end{cases}$$

Multiple vertical correlation: The load is modified in a given proportion p, in a given number of columns n, distributed along the graph with a fixed stride $s = P/n$:

$$\tau(v) = \begin{cases} x \leadsto N(\mu,\sigma) & \text{if } (\xi(v) \bmod s) \neq 0 \\ x \leadsto N(p\mu, p\sigma) & \text{if } (\xi(v) \bmod s) = 0 \end{cases}$$

Multiple horizontal correlation: The load is modified in a given proportion p, in a given number of layers n, distributed along the graph with a fixed stride $s = D/n$:

$$\tau(v) = \begin{cases} x \leadsto N(\mu,\sigma) & \text{if } (d(v) \bmod s) \neq 0 \\ x \leadsto N(p\mu, p\sigma) & \text{if } (d(v) \bmod s) = 0 \end{cases}$$

We are interested in detecting how the position of columns or rows with modified load, and the load modification are affecting γ. Thus, we design the following experiments. Let M be a random mesh with $P = D = 64$ and $S = 3$. We carry out the following experiments, were some of the parameters have been adjusted in view of the results discussed below:

1. Vertical correlation of one column, changing the column position. As the graph is symmetric, and the dependences are randomly distributed across the full graph width, we expect symmetric results moving the column from the center of the mesh to each extreme:

$$p = 2, c \in \{1, 2, 4, 8, 16, 32, 49, 57, 61, 63, 64\}$$

2. Vertical correlation of one fixed column, changing the workload modification. We test both, lower and higher values of the load in the selected column:

$$c = 32, p \in \{0.5, 0.8, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 4.0\}$$

3. Horizontal correlation of one layer, changing the layer position. As the *cpv* is accumulated through the graph up-down, we test modified layers all along the graph:

$$p = 32, r \in \{1, 2, 4, 8, 16, 32, 49, 57, 61, 63, 64\}$$

4. Horizontal correlation of one fixed layer, changing the workload modification. In view of the results of our first experiments in this category, we detect that we need to increase the load much more than in vertical correlations to get representative results:

$$r = 32, p \in \{0.5, 1.0, 1.1, 1.2, 1.5, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0\}$$

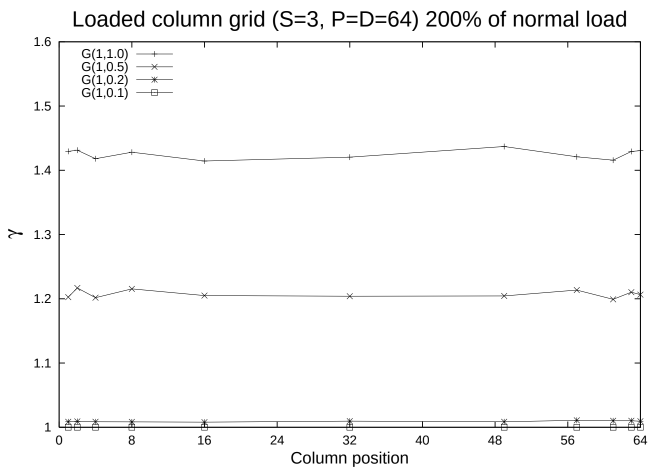
5. Multiple column correlation, with different number of columns to generate all the possible integer strides for $P = 64$:

$$p = 2, n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 21, 32, 64\}$$

6. Multiple layer correlation, with different number of layers to generate all the possible integer strides for $D = 64$:

$$p = 6, n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 21, 32, 64\}$$

All the experiments will be carried out with different workload variabilities $\varsigma \in \{0.1, 0.2, 0.5, 1.0\}$, and drawing 25 times random workload distributions for each topology. The results obtained from these experiments can be summarize as follows. For the following discussion, keep in mind that γ is minimum when the critical path of the NSP graph has the more loaded nodes of each layer:



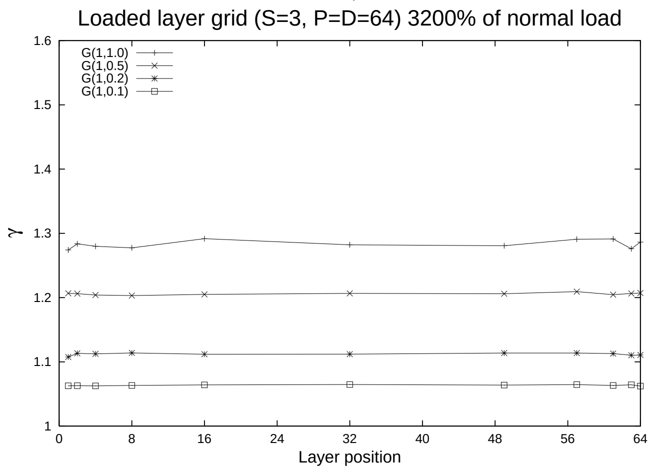


Figure 4.18: Independence of load correlation position

1. Independence of the column or layer position:

   As it is shown in Fig. 4.18, the position of the column or layer which load is modified is not really important. In the case of vertical correlation, as along as the edges in the mesh are chosen randomly, the dependences are propagated with equal probabilities, independently of the column position. In meshes with deterministic synchronization functions, it should be possible to observe little γ differences when the modified column position changes. In the case of horizontal correlation, all full paths must cross the layer, independently of the layer position, getting the same probabilities of being affected.

2. Beneficial effect of the vertical correlation:

In Fig. 4.19 we may appreciate the beneficial impact of increasing the workload in one modified column. A lower load than in other columns does not significantly modify the γ values, because the more loaded nodes in each layer are the normally loaded nodes. The maximum accumulated path value through several edges, is always got from one of the normally loaded nodes. On the other hand, when the modified load is increased above the normally loaded nodes, the paths that cross the highly loaded column more times, get more and more probabilities to become the critical path. At the same time, the nodes in the column get more and more probabilities to be the more loaded nodes in the layer, especially when the variability is small. Thus, the critical path in the NSP version gets more probabilities to have exactly the more loaded nodes in each layer, minimizing γ. As it

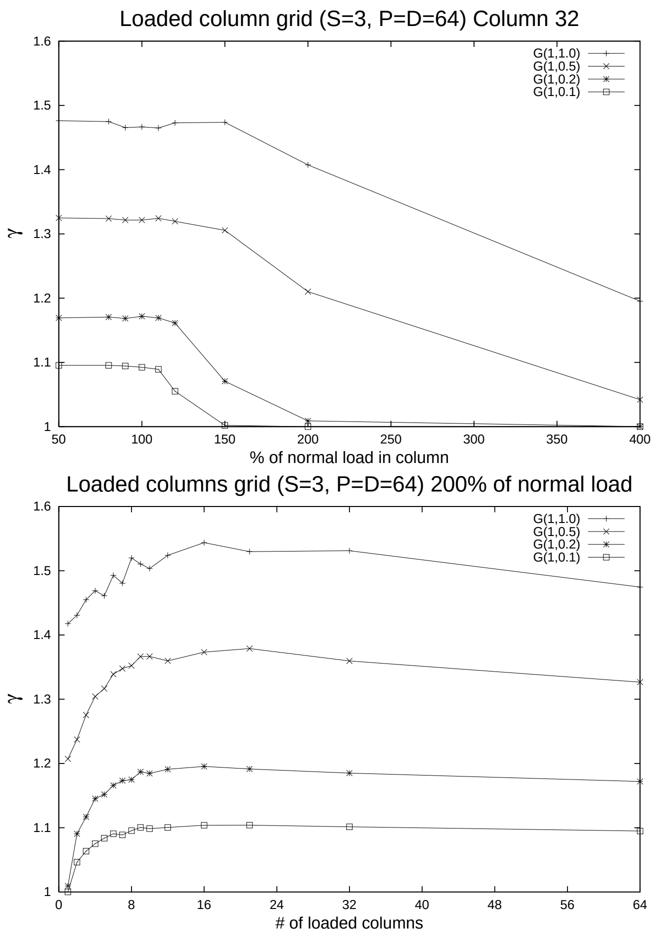


Figure 4.19: Beneficial effect of the vertical correlation

is also shown in Fig. 4.19, this beneficial effect immediately disappears if there are several columns with the same load modifications in the mesh. The reason is that there are more probabilities for the critical path to cross a highly loaded node (in one of the modified columns) which is not exactly the more loaded node in the layer (being in other of the modified columns). Fortunately, in applications with vertical correlation (like some pipelines) is typical that most columns have different mean load values. The existence of this small load differences between columns lead to a middle point on the beneficial effect.

3. Beneficial effect of the horizontal correlation:

   In Fig. 4.20 we can see that in the case of horizontal correlation, small modifications of the load does not affect γ. Although all paths must cross the modified loaded layer, there are not so many probabilities for the critical path to cross exactly the more loaded node in that layer. However, when the load in the modified layer is highly increased, in a much bigger proportion than the other layer nodes, the paths that cross exactly the more loaded node in that layer have more and more probabilities of being the critical path themselves, as the other layers loads become less significant in the total path value. In the same figure we can also appreciate that increasing the number of loaded layers is potentially beneficial until a given point. The reason is that the effect previously discussed for one layer is applied more and more times. However, when the number of layers increases too much, the extra loaded nodes become too frequent, and they become the normally loaded nodes. Then, the full paths get the typical variability effects of the now more common nodes in the mesh, eliminating the beneficial effect of the correlation. This workload configuration with many layers more loaded than a few ones is not so typical in applications.

The important conclusion about this experiment, is that typical correlation between topology and workload may produce beneficial effects on γ in many circumstances. Thus, our previous predictions with i.i.d. workloads can be considered a worst case for workload distribution, and previous γ predictions can be considered upper bounds of the expected $\overline{\gamma}$ in typical application structures.

4.1.4 Conclusions about synthetic graph results

Although not being a topological feature, the workload balance is the graph characteristic with the higher impact in the potential performance loss measured with critical path analysis (γ). Our main study is based on i.i.d. workloads due to the absence of real workload information. Nevertheless, more irregular workload distributions with typical application correlation in vertical and horizontal node instances may produce even lower expected γ results.

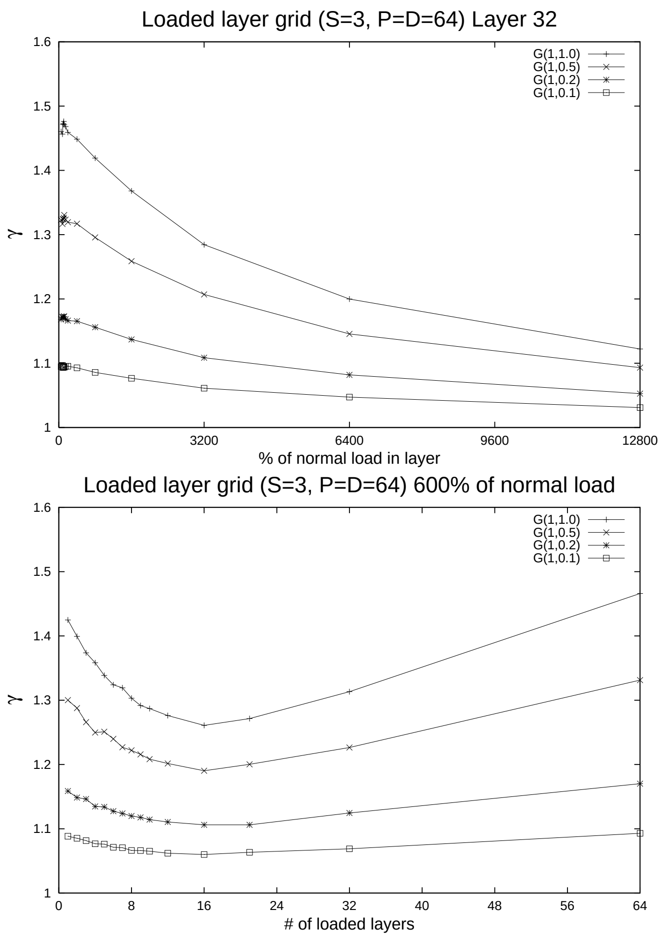


Figure 4.20: Beneficial effect of the horizontal correlation

For random or irregular graphs, the P and D parameters are typically correlated with S. Thus, S and R_s values easily determine the γ values. The dispersion of γ values is maximum around the critical point of $S \approx 2$ where γ values also reach their maximum.

More structured graphs, which nodes are organized in layers connected by random or replicative synchronization structures, do not present a correlation between the parameters S and P, D. If the synchronization structure across layers is random, or balanced (as measured with ω for deterministic synchronization structures), the γ values can be estimated with the workload characteristics and the simple topology parameters P, D, S. The values of γ reach their maximum for $S \leq 2$. Further increase of S immediately limits the γ increase. The number of layers in the mesh is only important until $D \approx P/S$. More layers do not further affect the potential performance loss. Thus, γ is highly predictable as a function of very simple topological and workload parameters.

For unbalanced synchronization structures ($\omega < 1$), pathological effects are observed in the S and D effects. Future work should relate these observations to ω values.

4.2 Real Applications

In this section we present an study of the performance effect of using different SP and NSP programming techniques with real applications. Our purpose is to determine the potential performance loss produced at programming level due to restrict relevant synchronization structures to SP form. We choose application examples which are representative of important SA classes (see 2.6). We use graphs to model applications at different detail or abstraction levels. Modeling techniques and their accuracy are studied. Transformation techniques and γ predictions previously discussed, are studied in structures from real applications. We present results on how γ is propagated to run-time level in real implementations Γ. The main trends of this loss are studied before applying any improvement derived from SP programming. Thus, no advantages of SP programming will be exploited in our experiments during implementation or run-time. Finally, we specifically focus our study on more irregular applications, showing how typical load balancing and data-partitioning techniques lead to more regular structures, feasible for SP-programming.

4.2.1 Experiments design

Experiments are conducted to compare information obtained from programming level cost models with real implementations. Results are studied to extract parameters non-dependent on the application which predict the mean performance effects of restructuring programs for SP programming frameworks.

We first focus our study in applications in the NDS classes, where the structure of the application is fixed for some simple parameters after mapping (mainly the number of processors).

The experiments are designed as follows:

1. Select a representative application of a static NSP SA class.

2. Implement the program in both NSP and SP versions, for different machine architectures and/or programming models.

3. Run programs obtaining load and performance measurements.

4. Derive programming level graph cost models.

5. Estimate mean program behavior with synthetic workloads and statistical load measurements.

6. Compare estimations with real performance measurements. If accuracy is not enough, refine cost models at a lower implementation level and go back to phase 5.

7. Relate application behavior and SP-ization impact to structure parameters.

For dynamic classes, structure is data-dependent and cannot be easily derived. For these cases an exhaustive study is not always possible. Availability of simple codes is limited, applications trend to be much more complex, they typically include hard-wired optimizations based on the machine architecture, and finally, many alternatives of implementation exists for almost any algorithm. Input data may have a great impact in an specific structure, as dynamic scheduling and mapping techniques are used.

Thus, our experiments are oriented to exploit available samples of running traces, obtained by execution monitoring. Task graphs are built from the tracing information. The stages of each experiment are:

1. Find examples of structures (task graphs) generated by executing existing implementations of an application, with different real input data, on specific machines. If possible, we will gather detailed real workload information in run-time.

2. Apply the Algorithm2 transformation technique, presented in 3.5, to the sample structures.

3. Compute and compare performance (*cpv*) in the original and transformed structure and relate it to structural parameters.

Applications selected

Along the lines presented in the applications classification in section 2.6, we select the following representative examples of relevant NSP classes:

1. Static NSP applications:

   Static macro-pipeline: It is is a good representation of simple structures created by multiple iterations of a shifting memory access pattern. Many parallel non-synchronized loops and data mappings create structures similar to this one.

   This application also presents the minimum synchronization density S parameter value possible for complete regular applications in which all processors execute the same piece of code with the same communication pattern. Nevertheless, each iteration provides a full chain NSP composition (see 3.3.3), that needs full layering synchronization to be transformed into SP form.

Moreover, the dependences between processors are not propagated in an homogeneous way, but in an specific direction of processors numbering after data partition. It leads to the biggest possible number of dependences added for any $S = 2$ structure after SP-ization, and it presents the pathological effects described in section 4.1.3 for structures with $\omega < 1$.

Thus, it is a extreme case for SP-ization impact.

1D Cellular automata: This application represents the neighbor synchronization structures. Many regular and scalable applications are mapped to this structures. Is is specifically representative of grid computations and PDE solvers. Even more complicated stencil based applications are mapped to this structure if a 1-dimensional data partition is used. In fact, we have chosen to implement a typical 2D grid computation mapped by rows, to produce a 1D cellular automata structure with real and representative computation loads (see an example of modeling this mapping in Fig. 4.23).

For this kind of neighbor synchronization and grid applications, the 1D cellular automata kernel present the minimum S parameter value ($S \approx 3$), being the application example most potentially affected when it is transformed to SP form.

FFT: It is an important kernel in many parallel applications and has been widely studied. Its *butterfly communication structure* is the most typical example of solving networks.

After the local computation phase, FFT is an intensive communication application, as all the local data is sent in each communication. In each iteration the communication phase interchanges data with further remote processors in a linear numbering. However, the binary tree pattern may be exploited with special mappings and implementations to improve locality in specific network models (see e.g. [156]).

LU reduction: Most matrix factorization algorithms (e.g. QR or Cholesky) presents similar SA. It is a complex application for graph cost model derivation as discussed in section 4.2.3. At program level it present a triangular synchronization structure that must be mapped at implementation level to another different form for regularity and scalability. This mapping leads to decreasing task load values along iterations.

2. Dynamic application classes:

Simulations based on graphs: Most physical or chemistry simulations are based on a PDE iterative sparse-matrix solver. The matrix structure represents the adjacency of the joint points of a 3D mesh which models the studied object. For these applications, the synchronization structure generated is completely dependent on the data mapping, typically based on a graph partitioning algorithm.

As example of the structures produced by these applications we generate mapping level task graphs of a simple PDE solver style program running on graphs already partitioned with a free and state-of-the-art multi-level partitioning software for unstructured graphs (METIS [117, 167]). Example input 3D models are chosen from the structural engineering area, from a collection of free test data intended for use in comparative studies of algorithms for numerical linear algebra (Matrix Market [146]).

Sparse-Matrix factorization: This application is a good representative of structures generated by direct solver techniques for sparse-matrix computations. As an example of the performance impact of SP-ization in these class of applications, we have apply our transformation algorithm to some graphs generated by monitoring the execution of a domain decomposition and unstructured sparse-matrix factorization software [55, 123, 124] for finite-element problems. The automatically obtained graphs are provided with real workloads.

These two problems covers the typical synchronization structures generated by parallel implementations of the main iterative and direct solvers for sparse-matrix computations.

Machine architectures

At implementation level a parallel program is compiled and optimized for an specific machine. When executed, it uses costly mechanisms to spawn, synchronize and communicate tasks. Implementation details and the underlying architecture of the machine become important. For simple applications and kernels we want to study the main performance effects in different programming models, and also different machine architectures. We have selected available machines to cover different architecture models and typical configurations of them:

Shared memory architectures: The programming techniques used in these machines are straightforward, and the programmer is not normally facing the data distribution or scheduling details directly.

Our study is focused on a leading edge technology shared-memory architecture: CC-NUMA. Our available machine is an Origin2000. CC-NUMA machines have representative properties for performance evaluation of synchronization techniques. The use of memory hierarchy improves performance, while cache-coherence protocols and automatic process migration try to hide machine level details to the programmer. Nevertheless, the efficient use of memory locality is not an easy task even with compiler assistance. Delay times for data access and synchronizations are less stable than in other architectures, especially for full collective communications, like barriers issued across the whole system [102].

Distributed memory architectures: The main parallel programming model used for this kind of machines is *message-passing*. The programmer faces problems as data distribution or scheduling details inherently, increasing the developing effort.

We use two key types of distributed memory machines that have representative properties for performance evaluation of synchronization techniques. CrayT3E is a mesh-based computer, with hardware and protocol improvements to minimize the overhead of distant processors communication. The special-purpose hardware is highly efficient. A Beowulf system (a cluster of PC computers linked by a high speed Ethernet switch [176, 177, 151]) normally presents higher communication costs. As the underlying message-passing tools are prepared to work in generic/all-purpose networks, the implementation details can create irregularities in the network traffic or communication delays. Both machines are at the budget extremes for high performance computing. CrayT3E is an expensive specifically designed machine, while a Beowulf is an optimized way to create a supercomputer from generic, all-purpose, and in comparison cheap, computer hardware.

Programming models and code generation

After determining the applications and machines, we must select a convenient programming model to codify the NSP and SP versions of each program. The minimum requirement for a programming model to be selected are:

1. Codes must be portable with minimum or none modifications to every architecture tested.

2. A systematic code transformation technique must be devised to derive SP versions from NSP versions of the final code.

3. A systematic technique to extract programming or implementation level graph models from the code must be devised.

4. It must provide similar performance as compared with other native or more specific models.

According to the previous requirements, we consider the MPI message passing interface as the best candidate for our experimental framework for the following reasons:

1. It is a portable API. Programs implemented in MPI can be compiled and executed in almost any parallel machine due to standard MPI implementations.

2. As MPI is a full standard interface of the well-known message-passing model, many applications are already studied and implemented on this model (see e.g. [189]). Real codes for some of the selected applications are available.

3. It is a performance efficient and reliable tool. Most vendors provide their specifically optimized implementations. Generic but efficient implementations (e.g. *mpich*) are also available.

4. Message-passing model forces explicit communication. Scheduling, data-partition and any other mapping transformations must be hard-wired in the code. Thus, a complete monitoring of communication activities at high level is possible. In section 4.2.2 and section 4.2.3 we introduce systematic ways to extract task graph models from codes in different programming paradigms. We especially study the message-passing problems and solutions, including an example for MPI. Message-passing interfaces simplify task and communication identification because communication is always explicit.

5. Transforming NSP MPI codes to SP form is easy because of the explicit communication. Communication phases are formed by grouping consecutive communication primitives, with no computation code in-between (see section 4.2.2). Synchronizing the programs to simulate the added dependences needed for SP-ization may be as simple as adding barrier synchronizations after communication phases. Probably, there exist other and better methods to transform the original code to SP form, but this approach is simple, systematic, and a typical worst case, where no code manipulation is done except to add dependences through barriers. The technique is suitable to exhibit an application potential degradation of performance due to the extra synchronizations when programmed in an SP PPM.

6. Message-passing libraries as MPI allow very fine tuning of the codes for performance. The library implementations, specifically for MPI, are fast and efficient.

   In shared memory machines, there are other interesting and widely known programming models as OpenMP, directed to portable and efficient devel-

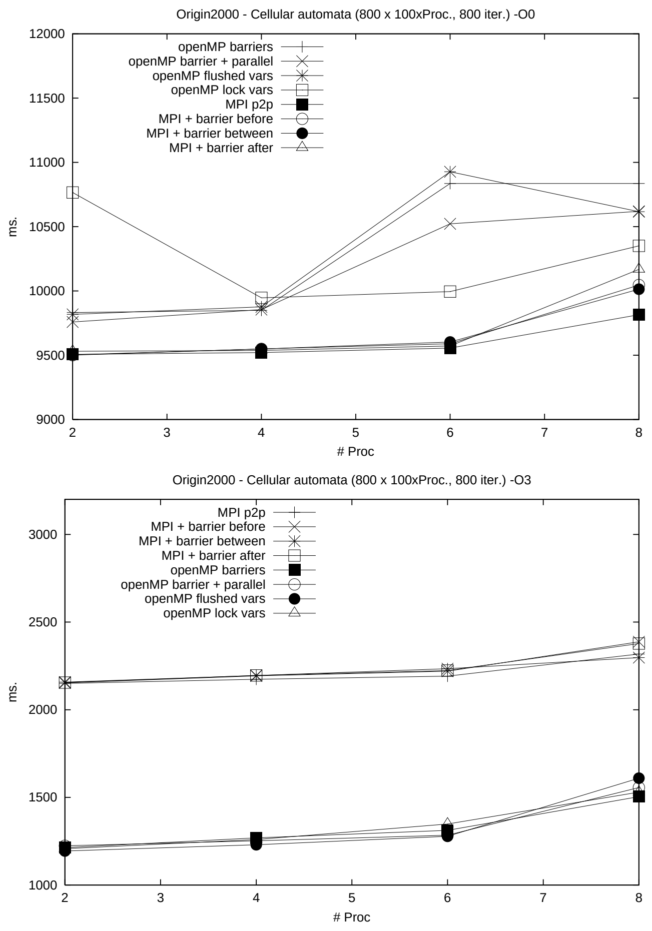


Figure 4.21: OpenMP vs. MPI implementations in Origin2000

opment. We have tested different implementations of several applications with OpenMP and MPI to compare their relative efficiency or detect differences in the effect of SP-ization for so different programming models. In Fig. 4.21 we show the performance obtained in an Origin2000 machine for a simple cellular automata program, implemented in several different ways. The plots correspond to the same codes compiled with no compiler optimization (`-O0`), and with aggressive compiler optimization (`-O3`) respectively.

The codes include: (1) OpenMP SP code that executes the iteration loop inside a parallel region, with full synchronization barriers before and after copying of frontier shared data; (2) OpenMP SP code which spawns and collapses a parallel region inside each iteration, with only one explicit barrier needed for synchronization; (3) OpenMP NSP code based on control variables *flushed* across the memory system, with active waitings; (4) OpenMP NSP code synchronized through lock variables; (5) MPI NSP code based on simple point to point communications; (6) MPI SP code with a barrier added *before* communication; (7) MPI SP code with a barrier added *between* send and receive parts of the communication; (8) MPI SP code with a barrier added *after* communication, before computation phase. Data sizes are scaled up with the number of processors to keep the tasks load independent of the number of processors. Task loads are highly regular for this problem, thus SP-ization impact should be negligible. Our results indicate that OpenMP and MPI implementations are similar in performance, for both: NSP and SP versions. Results are independently of code restructuring, change of primitives or synchronization system, or even the barrier placement. MPI shows a more stable behavior than OpenMP versions when we do not allow compiler optimizations, which is interesting for our study (as we discuss below). Code versions using native OpenMP perform better than MPI when aggressive compiler optimization is used. However, the performance degradation is a constant delay due to extra process creation and manipulation in MPI, compared with the efficient native thread creation system used by the OpenMP implementation. MPI results are still efficient and completely similar regarding the performance trends, and the NSP to SP code restructuring.

Once the programming model is selected, we discuss other implementation details. We must be careful about code or compiler optimizations. Fine tunings that are not portable across machines must be avoided. We are mostly interested in simple direct codes that implement the basic communication scheme for each application. For efficient software development we must rely in compiler optimizations and efficient run-time environments tuned to the specific target machine. However, we do not yet have a programming framework that really exploits all SP properties for optimization. Moreover, our study is focused to detect the potential performance loss due to transformations at programming level. Advantages obtained during implementation phase are impossible to be fairly evaluated nowadays, as they can be produced by SP compiler transformations, run-time scheduling, or even by other non-related compiler optimizations, like better sequential code manipulation, cache trashing reduction or internal buffering optimization (partial studies of SP optimization advantages exist, and they point to good performance advantages obtained due to implementation transformations when restricted SAs are used, see e.g. [57]).

Thus, we must avoid aggressive optimizations. Compiler code manipulation (loop reordering, unrolls, buffering optimizations), may change the synchronization patterns in such a way that: (1) implementation model of the resulting transformed code is impossible to be known or derive even at run-time; (2) the low level programs resulting from NSP and SP structures are so structurally different that they are not comparable anymore. For these reasons, for our experiments we have selected no compiler optimization at all (we include the `-O0` flag in all compilation jobs).

For each application considered we generate an NSP and a related SP version based on the same original code:

1. MPI, NSP version: Based in point to point or basic collective communications.

2. MPI, SP version: The former version with added barriers after communication phases.

First we program a basic NSP version of the application using simple MPI point to point communications. This reference version may be refined to a second NSP version using collective operations[1]. We compare the NSP code with an SP version created by adding barrier synchronizations after the communication phase of each stage or loop iteration.

In the experiments with synthetic graphs we made the assumption of i.i.d. task loads for any degree of parallelism. To be able to compare results and trends obtained from synthetic graphs, with results obtained with these new real application experiments, we use *scaled up* problem sizes in order to keep the mean of the task loads as independent as possible of the number of processors. Problem sizes are also loosely adapted to the relative hardware speed across machines to obtain performance results in the same order of magnitude, and similar communication to computation ratios.

Measures include the total execution time of the parallel section of each code, as well as the mean and deviation of task and communication times. We consider a task to be a continuous sequential computation, from the point after a wait for synchronization has been performed (one or more communications or a barrier) to the next one (see following sections for more details). The experiments are conducted up to all the available processors (2 to 8 in the Origin2000, 16 to 128 in the CrayT3E, and 2 to 16 in the Beowulf system).

4.2.2 Application cost models at programming level

Applications may be modeled with different detail level (recall discussion about implementation trajectory represented in Fig. 3.25, section 3.6.1). An application synchronization structure is transformed from its original programming

[1]MPI standard states that collective operations may or may not be synchronized. It is implementation dependent [140].

shape during mapping and implementation phases. At programming level, with no resource restriction, all possible parallelism can be exploited. In the mapping to resources phase, data partition may affect the task structure of the application. The implementation of the communication/synchronization mechanisms may also create new low level structures. Thus, different task graphs models will be used at different implementation levels. From simpler ones at the higher abstraction levels, to more complex and detailed ones at lower levels.

In this section we introduce procedures to model real applications with task graphs at programming or mapping level. These graphs are cost models when provided with synthetic or real workloads. Our cost models will be as simplistic as possible while they will provide at least asymptotically accurate performance predictions.

At the programming abstraction level, the specification of an algorithm is adapted to the synchronization structures available in the programming language and/or model used. Mapping constraints are not considered. Thus, the program could express all the parallelism available in the application in a very fine grain. The synchronization structure is derived manually from the algorithm specification or program. A graph representing tasks and dependences can be generated to represent it. In the case of MPI model, some mapping decisions (like data-partitioning among processors and other code adaptations to use a fixed number of processors) are taken by the programmer and hard-wired in the code. The mapping level graphs can be derived from MPI codes using the mechanisms described in this section.

For dynamic applications where the communication/synchronization structure is data dependent, the exact task graph can only be generated at run-time, and will be different for different executions. Moreover, even the simplest and most regular codes are usually parameterized with, at least, the degree of parallelism or the number of iterations of a parallel repetitive computation. Thus, task graphs are representations of a class; they represent the overall structure produced at programming level for a given application (for any number of processors or iterations). Simpler static and high regular applications will be modeled by a very small amount of graphs that will have the same synchronization patterns, even if depth level and degree of parallelism change. More dynamic applications should be modeled with a higher number of graphs, enough to represent the typical structures that can be generated for different data.

Graph derivation mechanisms

At programming or mapping level, costs for communication or synchronization mechanisms are not an issue to consider. Their structure or cost cannot be evaluated until lower implementation details are considered. Thus, a very simplistic task graph model will be perfectly accurate to represent the structure of the application.

Nodes (Tasks): Each node of the graph represents a task. We consider a task to be an atomic activity which can be executed independently of the local state of other activities (tasks).

Edges (Dependences): Edges will represent only precedence of tasks imposed by the program semantics (data dependences or other synchronization needs).

Mutual exclusion: Graph edges represent ordered precedence constraints between tasks. Thus, they are only appropriate for CS. At programming or high abstract mapping level, there is not a way to translate ME synchronizations to directed task graph edges. The ME synchronization mechanism is solved in scheduling time, thus, it is an implementation dependent or run-time matter. In these lower levels, when ME is solved, an execution order will be forced between mutual exclusive tasks, but we cannot predict it at high abstraction levels.

To represent non-ordered synchronization (ME) in our programming level model we propose to use a different *label or color* for mutual exclusive nodes. Formally, we use a function that maps subsets of nodes to *mutual exclusion identifiers*. Nodes associated to the same identifier must be mutual exclusive. A node mapped to the empty set represents a node that is not mutual exclusive with any other one. No explicit ordered dependence will be added with edges between nodes due to mutual exclusion.

$$ME = \{m_1, m_2, ..., m_n\}$$

$$\kappa : V \rightarrow M \subseteq ME$$

Identifying tasks and dependences must be done manually from program specifications, and using the appropriate information associated with the programming model. In some models, especially those which use implicit communication through shared-memory, we must have enough information about the low level semantics and of such tools to determine which memory accesses or primitives of the language are local and which others imply a synchronization and therefore the end of a task and the beginning of another one. In explicit synchronization models as message-passing, it is easy to determine the start and end points of a task. The execution of pieces of code between communication directives is a task. In the case of MPI, that exhibits explicit communication and synchronization primitives, the identification of tasks and dependences is direct. We consider a group of communication primitives with no computation code in-between a *communication phase.* A task (graph node) is a sequential computation, beginning at the end of a computation phase, and ending before the next communication phase. Dependences (graph edges) may be extracted from the parameters of communication primitives that indicate the source and target tasks. When data-dependent parameters are used, the application is dynamic, and several graphs must be derived for typical data values.

Workload information

After identifying the tasks, we must classify them regarding their execution time characteristics. The graph can include as many types of task nodes as necessary $(V_{t1}, V_{t2}, ..., V_{tn})$. Nodes with the same type will share the same statistical workload model. However, for simplicity it is interesting to reduce the number of different task types. Most of the times, especially for highly parallel and scalable applications, the kernel of the application can be modeled with only one type of tasks which executes similar codes.

Formally, we split the tasks set V into different subsets. Nodes in each subset will be of a different *type*. Random workload distributions with different parameters are associated to the load of each node type.

$$T = \mathcal{P}V = \{T_1, T_2, ..., T_n\},$$
$$T_i = \{v \in V : \tau(v) \rightsquigarrow D(\mu_i, \sigma_i)\}$$

In complete absence of workload information we will assume all tasks to be i.i.d. (independent identically distributed). Thus, if no information about workload distribution is available, only one node type will be used.

Static regular applications modeling

We describe here examples and notations for modeling static regular structures. We introduced in [183] a simple language and an associated tool that allows easy synthetic graph reconstruction, based on the expression of regularities by parameterizable synchronization functions. This language may be used to easily construct the graphs associated with regular application structures discussed in this section.

Many typical scalable application structures are created by replicating the same local synchronization pattern for every task in each iteration. This applications may be modeled by meshes with a special synchronization function applied to each node in a layer (see mesh definitions and notations in section 4.1.3). For these replicative interlayer connection systems, the synchronization function may be defined as a *stencil* or local pattern of communication (see e.g. [162]).

Definition 4.2.1 *Let $M = (V, E)$ be a mesh. Let ρ be a synchronization function. ρ is a* Stencil *iff exists $R(\rho) \subset \mathbb{Z}$, called* Signature of the Stencil, *such that:*

$$R(\rho) = \{r_i; i = 1, ..., a \leq P\} :$$

$$E = \{(v, w) : v \in L_i, w \in L_{i+1}; \xi(w) = \xi(v) + r \in R(\rho)\}$$

In other words, the cardinality of layers P, and the stencil signature $R(\rho)$, define a collection of number pairs A, in the range $[1, P]$, that define the numbers of source and target nodes of edges between two consecutive layers:

$$A = \{(a, b) : a, b \in [1, P]; b = a + r \in R(\rho)\}$$

$$E = \{(v, w) : v \in L_i, w \in L_{i+1}; (\xi(v), \xi(w)) \in A\}$$

Definition 4.2.2 *A* Stencil Mesh *is a triplet $M' = (P, D, R(\rho))$, that defines a mesh graph $M = (V, E)$ with $|L| = D; |L_i| = P$ and E defined by the stencil signature $R(\rho)$.*

Stencils define synchronization functions based on local synchronization patterns. For example, the signature $R(\rho) = \{-1, 0, -1\}$ defines the synchronization pattern of meshes representing 1D cellular automates or neighbor synchronization structures. Fig. 4.22 shows the stencil mesh $M = (4, 3, \{-1, 0, 1\})$. The edges between layers are defined by the following A set, where the number pairs are defined by $P = 4$ and $R(\rho)$:

$$A = \{(1, 1), (1, 2), (2, 1), (2, 2), (2, 3), (3, 2), (3, 3), (3, 4), (4, 3), (4, 4)\}$$

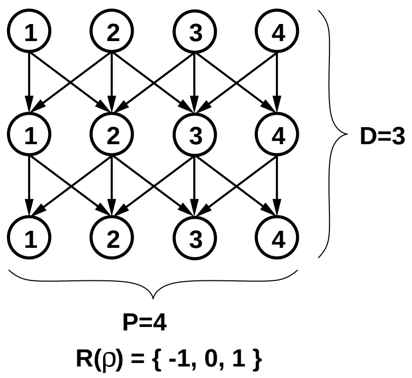


Figure 4.22: 1D Cellular Automata mesh defined by a stencil

The numbering of meshes nodes may be extended to $\mathbb{N}^n$, to more conveniently represent synchronization structures commonly found in applications based on 2D,3D cellular automates, quad- and oct-trees, etc. In those cases, the parameter P is represented by an n-tuple of natural numbers ($P \in \mathbb{N}^n$) and the signature of the stencil will be a collection of $\mathbb{Z}^n$ tuples. The A set will be formed by pairs of n-tuples. For example, consider the following 2D mesh: $M = ((4,4), 3, \{(-1,0), (0,1), (0,-1), (0,1), (0,0)\})$. This mesh represents 3 iterations of a 5-star stencil 2D cellular automata with 4×4 nodes in each layer. The nodes and the synchronization pattern are shown in Fig. 4.23.

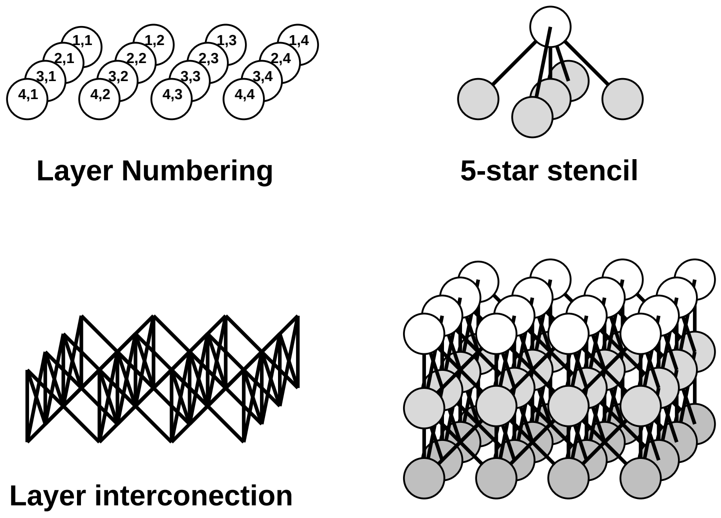


Figure 4.23: 2D Cellular Automata mesh defined by a stencil

For stencil functions, S is related to the number of elements in the stencil signature: $S \leq |R(\rho)|$. Boundary nodes may have less leaving edges than signature elements because the target numbers may be outside of the numbering range: $(\xi(v) + r \in R(\rho)) \notin [1, P]$. However, for large sizes of P, S becomes closer to the signature cardinality: $\lim_{P\to\infty} S = |R(\rho)|$. Thus, we consider $S \approx |R(\rho)|$ as a good approximation.

We present now the stencil mesh models for the two highly regular static applications selected for our experimental framework in section 4.2.1:

Static macro-pipeline: This simple structure is created by a 2 elements stencil signature ($S \approx 2$):

$$M = (P, D, R(\rho)) : R(\rho) = \{0, 1\}$$

1D Cellular automata: This application has been used as example previously. The signature has 3 elements ($S \approx 3$):

$$M = (P, D, R(\rho)) : R(\rho) = \{-1, 0, 1\}$$

In both cases, the computation to execute in each task is the same. Thus, all nodes will be of the same type for workload modeling. At programming level, each task computes one data element with the local data and the remote data received (one or two elements depending on the application). At mapping level, a big amount of data is partitioned among P processors. If the input data size is n, let $k = n/P$ be the number of data elements to be processed locally for each processor. Let c be the computation time needed to process one data element:

$$\forall v \in V : \tau(v) \approx k \times c$$

Other typical application structures are represented by graphs defined by synchronization functions that are changing with the number of layer i or node $\xi(v)$, or where the P parameter is also variable along the layers. We describe here the graph models of other static applications selected for our experiments in section 4.2.1. They present regularities that allows to express them with parameterizable and more complex synchronization functions:

Butterfly networks (FFT): For this kind of structures, D parameter is dependent on P, because the number of iterations needed to complete an FFT algorithm depend on the data size: $D = 1 + \log_2 P$. The synchronization function for this structure is dependent on the number of the layer. Let $L_i, L_{i+1}; i = 1, ..., D-1$ be two consecutive layers of the mesh. We define the *butterfly function* $f_i : [1, P] \to \{-1, +1\}$ as:

$$f_i(a) = 1 - 2 \times \lfloor (((a-1) \bmod 2^i)/2^{i+1}) \rfloor$$

The synchronization function may be defined as:

$$\rho_i(\xi(v)) = \{\xi(v), \xi(v) + f_i(\xi(v)) * 2^{i-1}\}$$

An example of this structure for $P = 4$ is shown in Fig. 4.24. The local FFT function always uses one element of local data and one element of remote data. For this structure the synchronization density value is exactly $S = 2$.

At programming level, each node represents the execution of the FFT function for two data elements, and all nodes are of the same type for workload modeling. However, at mapping level, when data is partitioned among a fixed number of processors, the nodes in the first layer execute the full FFT algorithm for the local piece of data. If data pieces have k elements, the local computation complexity is $k \times \log_2 k$. The nodes in following layers execute only one FFT iteration, with local and received data as input. The computation complexity is only k. Thus, at mapping level, in this kind of application we must distinguish two types of nodes for workload modeling. For n data elements, let $k = n/P$ and let c be the computation time to process one data element:

$$\begin{array}{ll} \forall v \in L_1 : & \tau(v) = k \times \log_2 k \times c \\ \forall v \notin L_1 : & \tau(v) = k \times c \end{array}$$

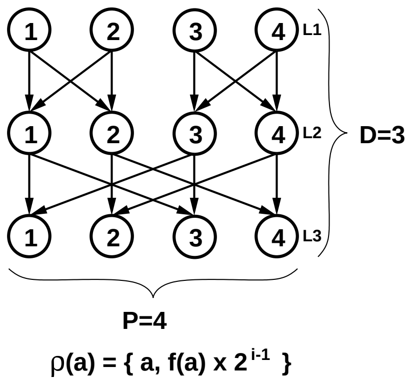


Figure 4.24: FFT butterfly network

Matrix factorization (LU reduction): We study the structure of a LU forward reduction algorithm without pivoting (see e.g. [79]). The structure of this application represents most factorization methods for dense matrices, like Cholesky or QR factorizations. A triangular shaped synchronization structure is generated. The code, parallelized by rows, is presented in Fig. 4.25. Given the sizes of the input matrix ($n \times n$), at programming

```
// LU ALGORITHM
DO k=1,n-1
    PARALLEL DO i=k+1,n
        $l_{i,k} = a_{i,k}/a_{k,k}$
        DO j=k+1,n
            $a_{i,j} = a_{i,j} - l_{i,k}a_{k,j}$
        END-DO
    END-DO
END-DO
```

Figure 4.25: LU forward reduction algorithm

level $P = n - 1$ and $D = n$. The structure presents layers with a decreasing number of nodes (column elements to be updated) along iterations (row updatings). In each iteration, a node computes the row that is needed for all the nodes in next iteration to update their rows. Synchronization patterns are: one to all from first node in a layer to all nodes in the next layer; and one to one for the rest of nodes. Thus, the synchronization function is different for different nodes in a layer.

The graph model of matrix factorizations is defined by the following declarations (let n be the dimensions of the input matrix $M_{n\times n}$):

$$D = n; P = n - 1$$

$$P_i = \begin{cases} 1 & \text{if } i = 1 \\ n - i + 1 & \text{if } i \neq 1 \end{cases}$$

$$\rho(\xi(v)) = \begin{cases} \{a : a \in [1, P_{i+1}]\} & \text{if } \xi(v) = 1 \\ \{\xi(v) - 1\} & \text{if } \xi(v) \neq 1 \end{cases}$$

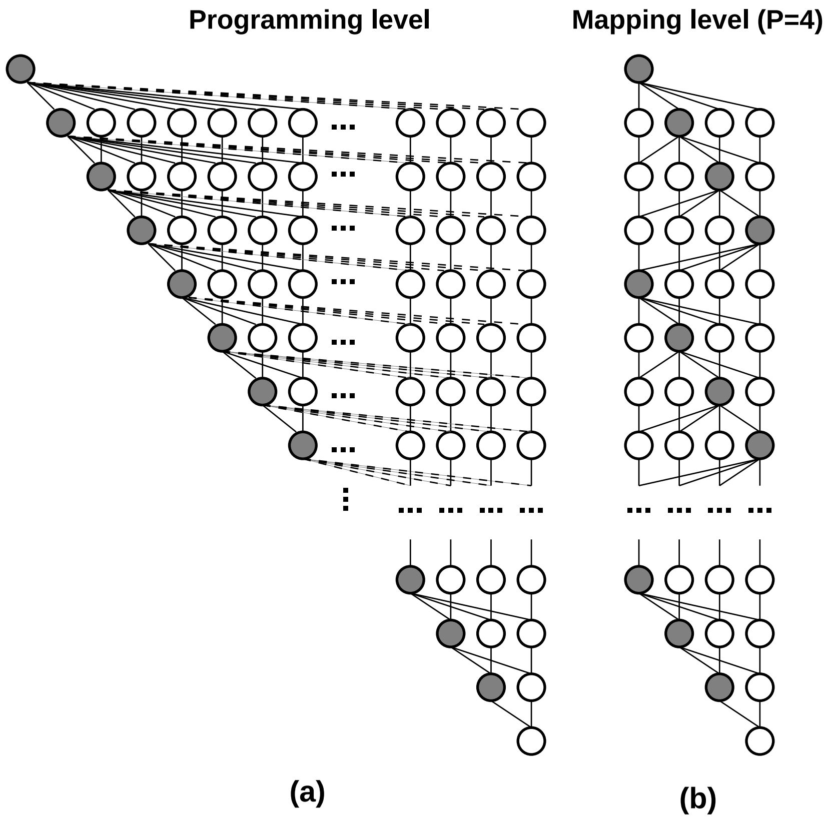


Figure 4.26: LU reduction: Programming level and mapping level graphs

An example of the structure generated is shown in Fig. 4.26*(a)*. The grey nodes represent the tasks that compute the row that must be made accessible to all other nodes in next iteration. For this applications S parameter may be analytically determined from the synchronization descriptions. The

S value is dependent on the P value:

$$\begin{aligned} |V| = &\ (n^2 - n)/2 \\ |E| = &\ n^2 - 2n + 1 \\ \\ S = &\ 2\frac{n^2-2n+1}{n^2-n} \\ \\ S < 2; &\ \lim_{n\to\infty} S = 2 \end{aligned}$$

Matrix factorizations present synchronization structures with a very low S parameter values.

Although this structure is similar to other factorization algorithms (as e.g. Cholesky factorization), the workload distribution will be different for each factorization algorithm. In fact, grey nodes in Cholesky factorization do more computation operations than the rest in the same layer. For our LU forward reduction algorithm each node in the same layer does the same number of element updates, but the number of updates is decreasing along iterations. Let be c the computation cost of one data element update:

$$\tau(v \in L_i) = \begin{cases} 0 & \text{if } i = 1 \\ c \times (n - i + 2) & \text{if } i \neq 1 \end{cases}$$

LU reduction is a problem with many different possible mappings and implementations that heavily change the synchronization pattern of the original program model shown in Fig. 4.26*(a)*. For example, a typical implementation achieves load and communication balancing by distributing rows of the matrix to processors, with a stride equal to the number of processors. Thus, for P processors, processor i will store the following set of matrix rows:

$$R_i = \{r_i, r_{(P+i)}, r_{(2P+i)}, r_{(3P+i)}, \ldots\}$$

Communication balancing is created because in each iteration a different processor computes and sends the row that all of them need to update the rest of their data in the following iteration. Cycling the processors that send one row to the others, changes the graph topology. Now, it is determined by n and P parameters:

$$D = n;$$

$$P_i = \begin{cases} 1 & \text{if } i = 1 \\ P & \text{if } 1 < i \leq n - P + 1 \\ n - i + 1 & \text{if } i > n - P + 1 \end{cases}$$

$$\rho(\xi(v)) = \begin{cases} \{a : a \in [1, P]\} & \text{if } \xi(v) - 1 = (i-1) \bmod P \\ \{\xi(v)\} & \text{if } \xi(v) - 1 \neq (i-1) \bmod P; i < n - P + 1 \\ \{\xi(v) - 1\} & \text{if } \xi(v) - 1 \neq (i-1) \bmod P; i \geq n - P + 1 \end{cases}$$

An example of the resulting mapping level graph for $P = 4$ is shown in Fig. 4.26*(b)*. For these mapping level graphs, the load is not so regular for nodes in the same layer, due to the different number of rows that each node may be processing. Thus, the workload model is more complicate. Let us assume $(n \bmod P) = 0$ for simplicity:

$$\tau(v \in L_i) = \begin{cases} 0 & \text{if } i = 1 \\ \lfloor (n-i+1)/P \rfloor \times row & \text{if } i \neq 1, \xi(v) - 1 < (i-1) \bmod P \\ \lceil (n-i+1)/P \rceil \times row & \text{if } i \neq 1, \xi(v) - 1 \geq (i-1) \bmod P \end{cases}$$

$$row = c \times (n - i + 2)$$

We conclude that extracting graph models from programming level specifications is a simple task for typical static programs, where the synchronization patterns are regularly repeated for scalability. Mapping level graphs may me more complicate and highly different from the corresponding programming level graphs. As the data is spread across processors in different patterns, the synchronization structures are adapted to these new patterns. Nevertheless, it is still an affordable task. The graph models obtained clearly represent the task and synchronization structures of the applications, and may be used with automatic SP-ization techniques to obtain equivalent SP versions of the original application.

4.2.3 Application cost models at implementation level

When implementing an application for an specific machine model, new constraints appear. The communication/synchronization structures must be adapted to the low level mechanisms of the selected target machine model. ME may be transformed to static dependences through scheduling in some models, while others will relay this task to run-time contention in communication systems. Thus, the implementation of communication/synchronization mechanisms may transform the task graph, adding new details. Communication structure and communication delays are now an important issue. They are introduced as nodes of their own specific type. We will distinguish as many node types as needed (tasks, point to point communications, barriers,...). Nodes of the same type will share a common workload distribution.

The communication graph structure is dependent on the implementation of the underlying communication layer and parallelization tools selected. For example, different implementations of a message passing library (as MPI) may implement the communication structure of a broadcast collective operation in different ways (synchronized vs. non-synchronized, one to all point to point communications vs. a tree). Programming tools may also include specific scheduling algorithms that produce different transformations to the graph structure. Thus, knowledge of all the low level details of the programming model chosen for implementation is needed to derive accurate graph cost models.

Deriving implementation level graphs

We describe a general approach to derive task graphs from a program description in a given machine and implementation model. This approach may be done automatically for some models and programs. We specifically comment foundation for automatic construction of task graphs in message passing systems.

Tasks identification: Tasks are identified in the same way as it was done at programming level (see section 4.2.2). The execution of sequential code between two communication or synchronization operation (or collection of operations without computation in-between) is considered a task.

Communication model: For our graph models we must use a very simplistic communication representation. Otherwise, the graph will be too complex to derive or handle. The details of communication can be different in any parallel programming tool and even in each implementations of it. In general we must simplify as much as possible but with enough detail to get a trustful approximation.

We present here a simple modelization of common operations in the communication layer of the MPI interface. We consider two different implementations. One for CrayT3E and other for a Beowulf system (*mpich*). Both implementations share common characteristics that let us model our simple communication schemes in the same way. A graphic representation of each communication form discussed is shown in Fig. 4.27.

- Four types of communication nodes will be used $(V_{c1}, V_{c2}, V_{c3}, V_b)$
- When a point to point communication appears alone, it can be considered as a whole in only one node (V_{c1}).
- In the situation where a program is issuing several point to point communications one after the other, all of them should be divided in two nodes:
    1. The first phase node (V_{c2}) will correspond to buffering the message and initiating the real communication. This phase will also delay the beginning of the next communication.
    2. The second phase node (V_{c3}) will correspond to real communication and reception for the message, and it will delay only the start of the receiving task.

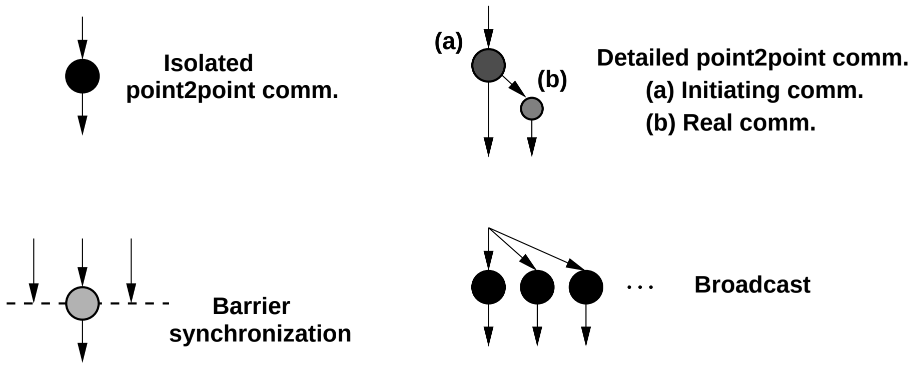


Figure 4.27: Communication models for MPI

- A broadcast communication will be represented by p (the number of processors) simultaneous point to point communications (V_{c1}). This will comply with the MPI interface that states that the implementation of a broadcast operation can or cannot be synchronized. Implementations of the broadcast operations can distribute the message spawning it through processors in different ways, being typical a virtual tree structure. Nevertheless, for simplicity we will consider all the nodes to have the same workload distribution. Measures in real machines support the accuracy of this simplification. A communication node will also connect the communication initiating task with the next task in the same processor, to represent the cost of issuing the broadcast.
- Barrier synchronizations will be modeled with a new type of nodes (V_b). In message-passing interfaces the barriers are typically implemented with a tree like communication structure. The cost is variable with the number of nodes involved in the barrier. Thus, a different type of node should be use for barriers with different number of processors. However, the tree-like structures have a logarithmic effect on the cost when the number of processors is increased. For simplification, only one type of node will be introduced for each range of processors number between powers of 2 ($V_{b2}, V_{b4}, V_{b8}, V_{b16}, ...$). Barrier times are easily predicted by direct measurement for any given number of processors.

Our model is simple enough to easily derive the implementation level task graphs, and accurate enough to get asymptotic predictions of the application behavior if proper workload models are provided for both, tasks and communications.

Example

In this section we show an example of how to use these simplified cost models to predict important information about the effects of SP-ization techniques when different MPI implementations of an algorithm are considered.

We have chosen the LU reduction application because it shows different performance effects when SP-ization is applied to different implementations of the same algorithm. These effects are not detected when using a programming level model, but they are predicted and explained with our simple implementation level graph cost models.

We discuss implementations of the forward reduction algorithm, mapped by rows interleaving as presented in section 4.2.2. Two implementations for the communication stage have been considered (See algorithms in Fig. 4.28):

IMP-1: A simple loop of point to point communications.

IMP-2: A broadcast operation.

SP versions of both implementations are easily constructed adding a full barrier synchronization after the communication stage of each iteration.

```
// LU IMP-1
DO iteration=0,n
    // COMMUNICATION
    IF mod(iteration,p) = myself THEN
        DO proc=1,p
            IF p ≠ myself THEN
                Send(proc,row)
            END-IF
        END-DO
    ELSE
        Receive(row)
    END-IF

    Barrier (ONLY SP VERSION)

    // COMPUTING: UPDATE ROWS
    ...
END-DO
```

```
// LU IMP-2
DO iteration=0,n
    // COMMUNICATION
    IF mod(iteration,p) = myself THEN
        Copy row in sending position
    END-IF
    Broadcast(row,mod(iteration,p))

    Barrier (ONLY SP VERSION)

    // COMPUTING: UPDATE ROWS
    ...
END-DO
```

Figure 4.28: LU reduction message-passing algorithms

The corresponding graph models for a mapping in 4 processors are shown in Fig. 4.29 The key to distinguish the types of nodes follows:

Tasks	V_t	White nodes
Full communication	V_{c1}	Black nodes
First phase communication	V_{c2}	Big dark grey nodes
Second phase communication	V_{c3}	Small dark grey nodes
Barriers	V_{bp}	Light grey nodes with dashed line

Tasks are executing updates on less data as iterations pass by. The mean load time of tasks is decreasing with the layer depth. In the mapping model we presented a workload model that was quite complicate; dependent on the number of layer (iteration) and number of node inside the layer (processor). We have tested other simplified workload models. For our example we have chosen to derive a very simplistic task graph with only one type of node for all tasks. We will use the same *Gaussian* random distribution to calculate the load in each node. Modeling any task load with the same random distribution is a very rough approximation. However, we find that using statistical information from sample executions, for mean and deviation parameters, the accuracy is enough for our purposes. It is the communication pattern the one which plays the important role in the results.

The statistical workload information can be obtained from sample executions or by any known prediction method. The results obtained will be highly sensible to the workload information accuracy, especially because we are using such a rough approximation of the real workload model. The mean and deviations used for task and communication nodes have been statistically obtained, from direct measures when executing codes of the MPI implementations discussed here. For tasks we use the overall mean and deviation when all tasks are considered together. Two machines with different communication times and characteristics are considered; a CrayT3E and a Beowulf system.

The graph models obtained are used to simulate performance behavior of the SP and NSP versions of each implementation. The results obtained from the models can be used to determine which implementation may be safely translated to SP (asymptotic behavior is not modified).

We present first an accuracy study, comparing predictions obtained from the graph models with execution times of real implementations in a CrayT3E and a Beowulf system. To supply graph models with workloads, we gather statistical information about mean and deviation values for the load on different types of nodes, from experiments with real codes. The size of the problem is scaled up with the number of processors, using matrices of double data size when doubling the number of processors. The initial matrix size has been empirically calculated for each machine to obtain similar task times. Table 4.1 shows the estimated parameters in the two machines considered, for the number of processors available. The load values have been rounded up before using them for graph simulations.

CrayT3E has faster mean communication times with lower deviations, even

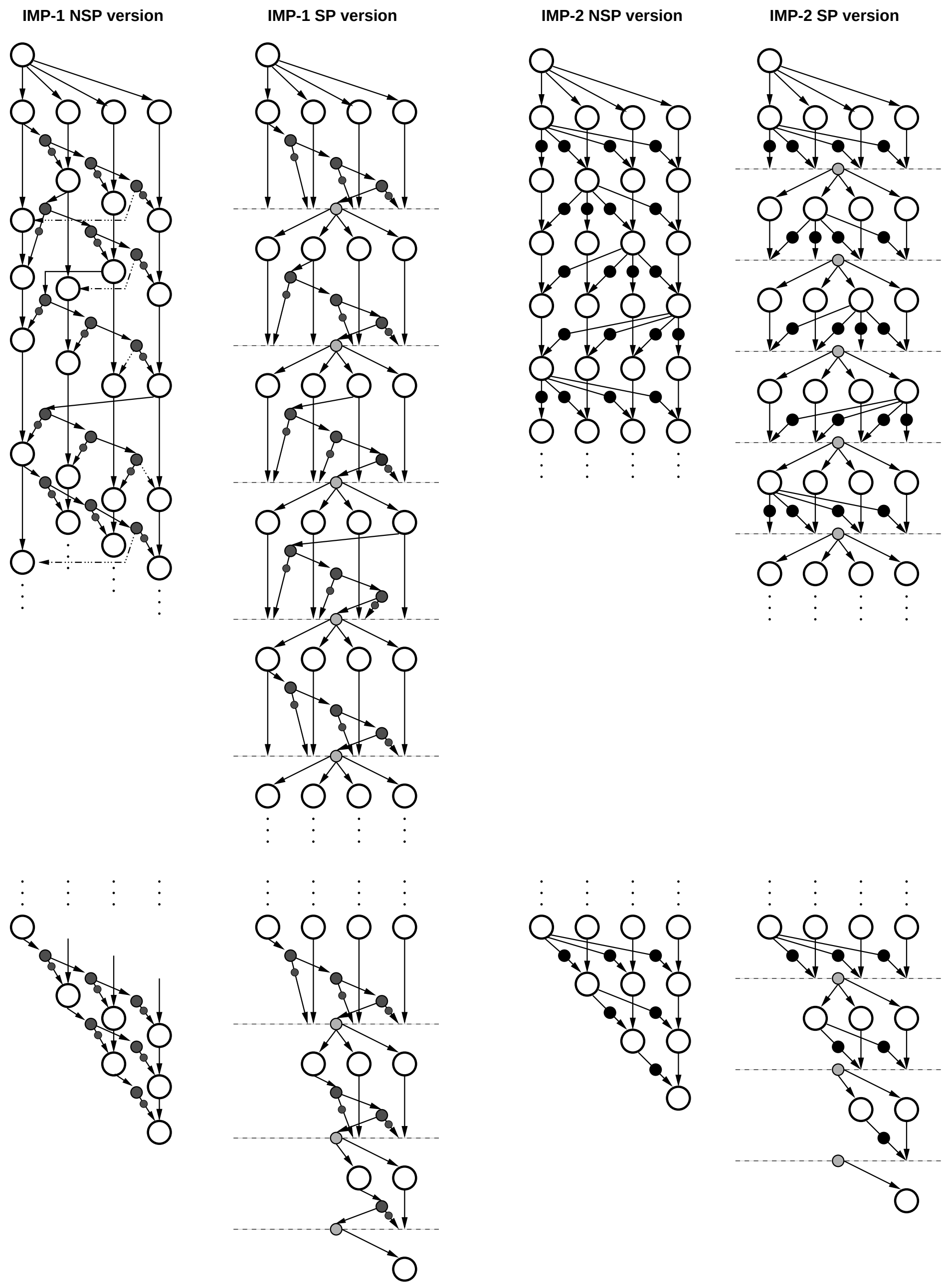


Figure 4.29: Implementation models of LU reduction with distributed rows

CrayT3E

Proc.	V_t		V_{c1}		V_{c2}		V_{c3}		V_{bp}	
	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2
16	19	0.28	0.66	0.00001	0.22	0.00001	0.44	0.00001	0.104	0.000001
32	19	0.28	0.88	0.00001	0.22	0.00001	0.66	0.00001	0.143	0.000001
64	19	0.28	1.32	0.00001	0.22	0.00001	1.10	0.00001	0.175	0.000001
128	19	0.28	2.22	0.00001	0.22	0.00001	1.98	0.00001	0.224	0.000001

Beowulf

Proc.	V_t		V_{c1}		V_{c2}		V_{c3}		V_{bp}	
	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2
2	19	0.31	1.00	0.0001	1.00	0.0001	0.00	0.0001	0.50	0.00001
4	19	0.31	2.00	0.0005	1.00	0.0001	1.00	0.0001	1.00	0.00010
8	19	0.31	6.00	0.0007	1.00	0.0001	5.00	0.0001	3.00	0.00100
16	19	0.31	9.00	0.0010	1.00	0.0001	8.00	0.0001	6.0 / 2.0*	0.00100

Table 4.1: Load estimated times (milliseconds)

for a large number of processors. Thus, the results of the simulations will be more reliable. The barrier synchronization system is also more efficient when scaling up. It is noteworthy the strange effect of barrier times for 16 processors in the Beowulf system. After a group of point to point communications, full barrier time still grows up (6.00ms). However, after a broadcast operation, the time is even smaller than with less processors (2.00ms). It seems that an optimization of either the MPI implementation or the hardware is carried out when a barrier is issued after a broadcast with all the processors in the system. Communication mean times in the Beowulf are in general not so much reliable, as unexpected peaks are commonly found.

Comparative results from real execution times and predictions with the graphs are shown in Fig. 4.30. In all cases the performance predicted times are similar to the real measures, and they show the same slope tendencies.

The first effect observed is that IMP-2 scales better than IMP-1. The graph model can be used to explain the effect. IMP-1 creates a strange communication pattern, that is not well balanced. The loop is always sending messages to processors in strict numbering order while the origin of communications is cycling. In Fig. 4.28 (NSP IMP-1), we can see that the first phase of each point to point communication, accumulated for all *send* primitives, is not evenly distributed to other processors. Depending on how significant is the mean load of the V_{c2} nodes compared to V_{c3}, the overall performance can be badly affected. Moreover, as more processors get involved, the delay grows higher. The relative importance of V_{c2} vs. V_{c3} loads is higher in the Beowulf system than in the CrayT3E, as shown in Table 4.1. However, we use many more processors in the CrayT3E. Thus, the final effect is even more noticeable in CrayT3E. The broadcast primitive of IMP-2 scales clearly better than the IMP-1 for both NSP and SP version.

Changing the loop indexes to cycle with the processor initiating the com-

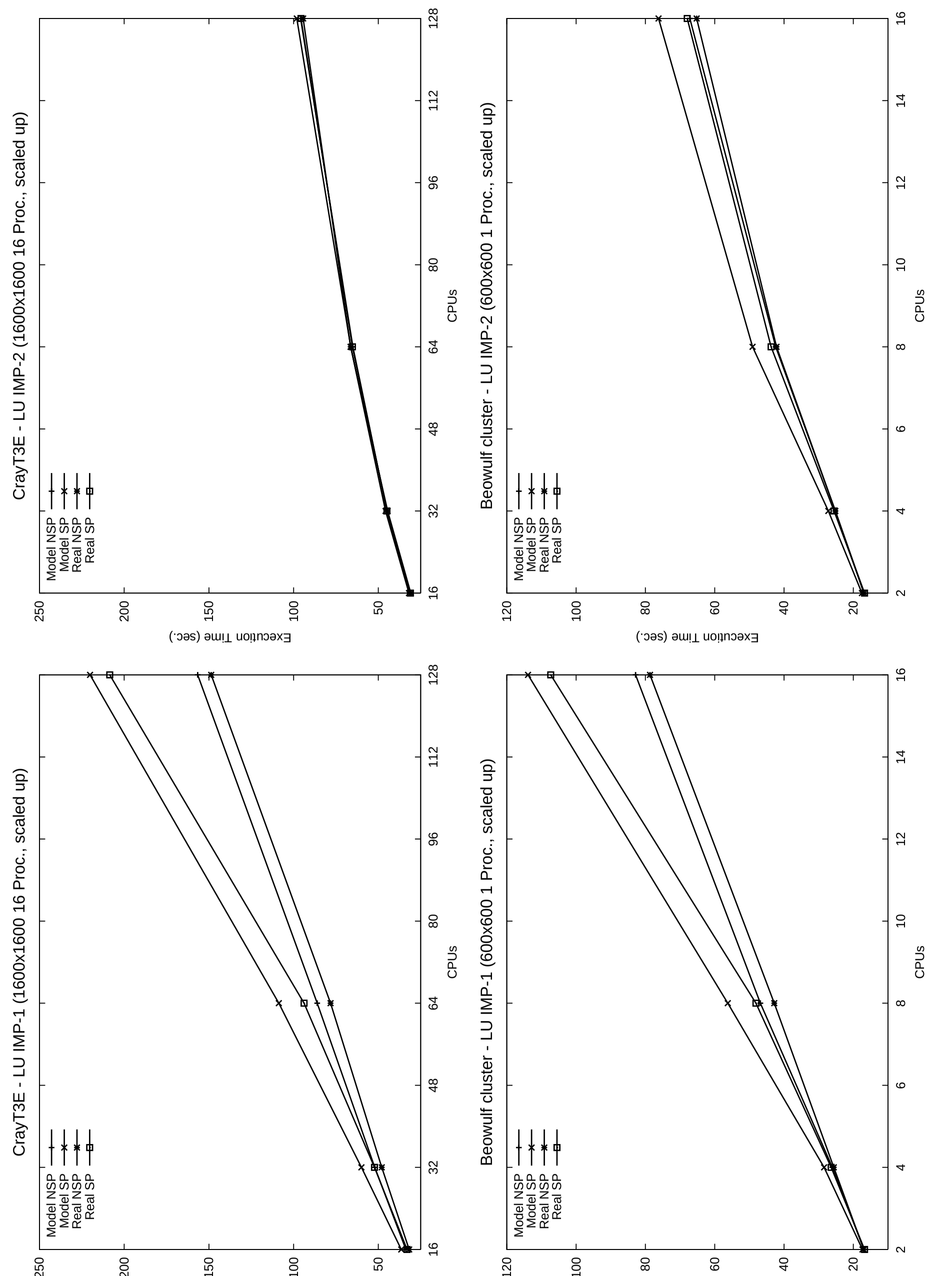


Figure 4.30: Performance comparison: Real times vs. predicted times

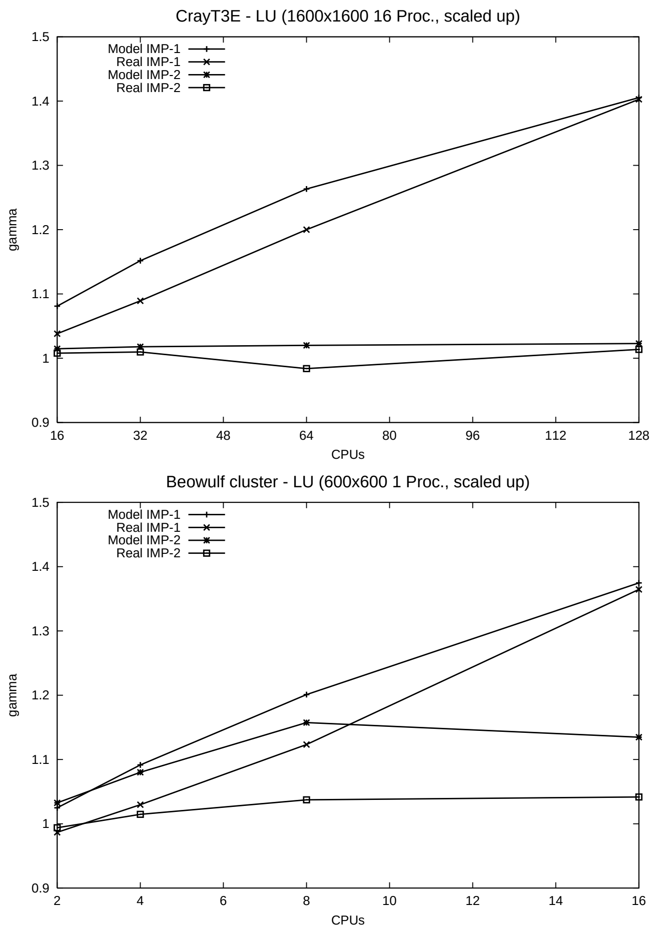


Figure 4.31: γ comparison: Real times γ vs. predicted times γ

munications will alleviate part of the problem. However, the SP version that adds a barrier after the communication stage is always delaying all processes up to the accumulation of all communications first phase. When the number of processors grows, the problem gets linearly worse. Fig. 4.31 shows the values of γ (performance loss due to SP-ization) for real and predicted results. It can be seen that IMP-2 is perfectly suitable for SP programming, as the γ values keep almost constant when the application is scaled up. The loss of performance in SP version of IMP-2, clearly seen in the Beowulf case, is generated by the times needed for barrier synchronizations when the number of processors grow up. Better barrier synchronization mechanisms will diminish this loss. The peeks in communication/synchronization times in the Beowulf also helps this grow.

We must also point out the curve slope differences between measured and predicted γ values. Since we used simplified graph models in our simulations, some irregularities appear, especially in the central part of the number of processors axis. In general, predicted results are higher than real measures, which is a consequence of the rounded up approximations we have used. As the number of processors (and thus the execution time) grows, the relative importance of accumulated rounding errors is smaller. All this could perfectly explain the shape differences of the γ prediction curves.

With the predictions obtained with our simplified graph models we can recognize that IMP-2 is in general better than IMP-1 due to the implementation on the underlying message-passing library. IMP-1 is especially not well suited for SP-ization with full barrier synchronization. On the other hand, IMP-2 behaves correctly in SP version, providing a very small loss of performance.

Conclusion

The previous study shows how very simple graph models can be used to asymptotically predict performance effects produced by synchronization structure modifications. For simpler applications, graph cost models derived at programming level will be accurate enough. When more complex mappings are used, more detailed models must be derived, at mapping or even implementation abstraction levels. However, very simple graph modelation techniques, that can be even automated to extract structure from codes, turn up to be accurate enough.

4.2.4 Static applications results

In this section we discuss the results of our study of γ and Γ for static applications. The experimental framework design was discussed in section 4.2.1. First, we obtain experimental measures of Γ from the execution times obtained with real MPI implementations of the NSP and SP versions of the selected applications for different machines. We extract workload information by monitoring the applications execution. Using statistical information about the real workload, we experimentally estimate γ with the cost models discussed in previous sections, to validate the simple graph modeling techniques for each application subclass. We compare our Γ results with γ predictions and general trends obtained for synthetic graphs, presented in section 4.1.

We more precisely define here the relative performance indicator we use for Γ. Our reference programming model will be the MPI with point to point (or basic collective) communications implementation.

Definition 4.2.3 *Let T_{MPI} be the execution time of the NSP version with point to point (or basic collective) communications. Let $T_{MPI+Barriers}$ be the execution time of the SP version generated adding barrier synchronizations after each* *communication phase. Then:*

$$\Gamma = \frac{T_{MPI+Barriers}}{T_{MPI}}$$

The following results are exposed:

1. Performance effects predicted with the graph models are similar to those obtained with synthetic graphs:

   In the case of Macro-Pipeline and 1D Cellular automata, the graph models are inside the synthetic meshes graph classes studied in section 4.1.3. In fact, Macro-Pipeline was used as foundation for the unbalanced synchronization meshes experiments. The 1D Cellular automata is also similar to the random graph meshes generated with $S = 3$. However, in random meshes, the edges were not propagating dependences only to *neighbor* nodes, but to further nodes with the same probability. In neighbor stencil based graphs, the dependences are spread across layers slower than for random synchronization functions, and the SP-ization should produce a little higher impact. In Fig. 4.32 we show that using random distributed workloads with the 1D Cellular automata graph model, we obtain very similar predictions as for $S = 3$ synthetic meshes. However, the γ results are slightly higher (compare with plot slopes in Fig. 4.10 and Fig. 4.11).

   The FFT application graph model present a low synchronization density parameter $S = 2$, and a number of layers dependent on the degree of parallelism $D = \log_2 P$. Thus, the number of layers is always low, and the critical parameter is the layers size P. In Fig. 4.33 we show the results of experiments with FFT graph models supplied with random workloads. The results confirm the same logarithmic like effect of parameter P on γ, for butterfly network structures.

   The LU reduction graph model derived at programming level, present interesting features. The S, P, D parameter values are dependent on the input matrix size n. Thus, the topology has always a similar triangular shape, more different from the synthetic meshes than the previous applications studied. Moreover, the workload model is dependent on the number of layer. For experiments with random workloads we propose a random workload model where μ_i is determined as a function of the layer index by the workload model proposed in section 4.2.2 for LU, and σ is computed as a function of μ_i and a chosen variability:

   $$\sigma_i = \mu_i \times \varsigma : \varsigma \in \{0.1, 0.2, 0.5, 1\}$$

   In Fig. 4.34 we show the effect produced on γ when we scale up the programming level structure. For this graph model, P, D parameters are equal

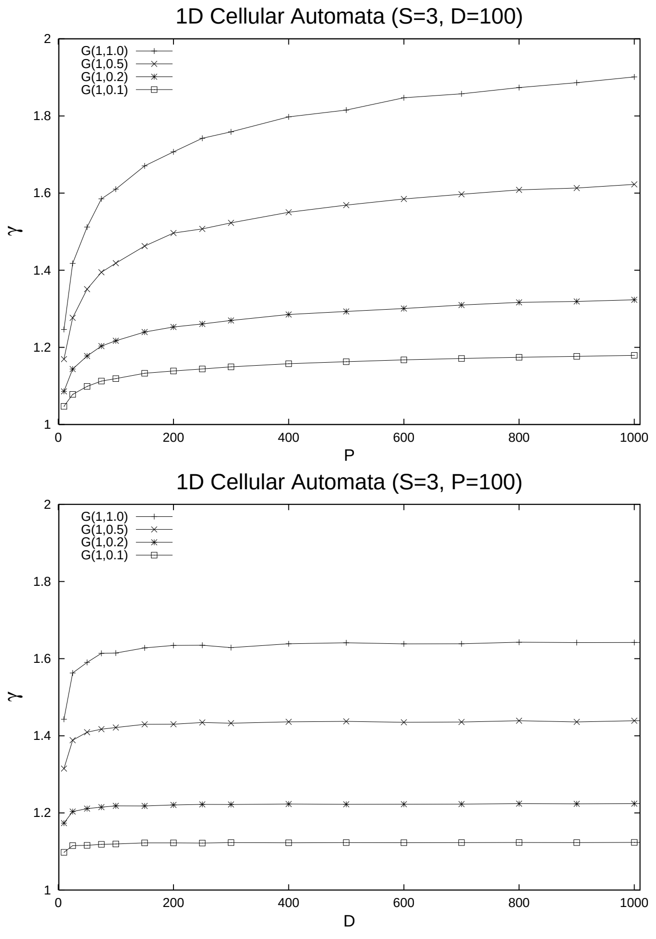


Figure 4.32: γ results for random workloads in 1D Cellular automata model

and determined by the input matrix size n. The S parameter is very low (below 2) and the full barrier synchronization cannot be avoided by the SP-ization techniques. Thus, as expected, worse results than for other applications are obtained. Although the plot slopes are higher than for other applications, the same logarithmic tendencies are observed.

The mapping level graph model has been also supplied with random workloads. In this case, n determines D but P is only restriction by $P \leq D$. Thus, we have conducted experiments to test the effect of both parameters separately with huge graphs (up to half million nodes). In Fig. 4.35 we show the γ plots for both experiments. As the minimum value of D is the same as P, the limited effect of this parameter, found in synthetic meshes,

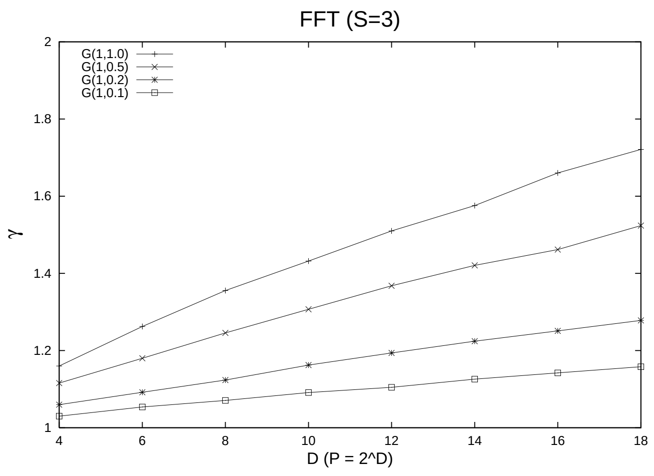


Figure 4.33: γ results for random workloads in FFT graph model

appears immediately, even in the lower possible values. However, we observe some irregularities and a very small increasing of γ when the input matrix size increase. The reason is the presence of a very small pathological behavior due to some unbalancing in the synchronization patterns, as discussed in section 4.1.3. The effect of P parameter is following the general logarithmic tendency, except for the irregularities produced by both: the small unbalancing in the synchronization patterns (as found in synthetic meshes) and the change of shape experimented by the graph with the P values. The pathological effect due to unbalanced synchronization pattern is producing the slope irregularity around $D/4$, but the triangular part of the graph dominates the behavior after $P = D/2$, producing another slope change.

It is interesting to notice that, considering the full range of results, the workload balance is much more important than the type of application or S parameter value. The big differences on γ among all the applications studied, are produced for big values of ς. For $\varsigma = 0.1$ the γ values are small, and the slopes are very similar (with less than 20% of difference among all applications and synthetic meshes), even for the biggest P values tested.

2. Task workload balance in static applications:

   A principle design of parallel applications is to distribute load across processors. For all static applications tested, the workloads are very well balanced. All task in these examples are executing the same piece of code for the same amount of data. As discussed in section 4.2.2, FFT or LU

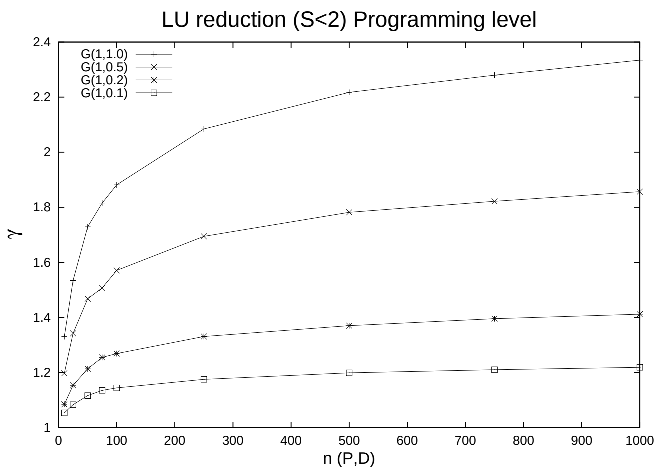
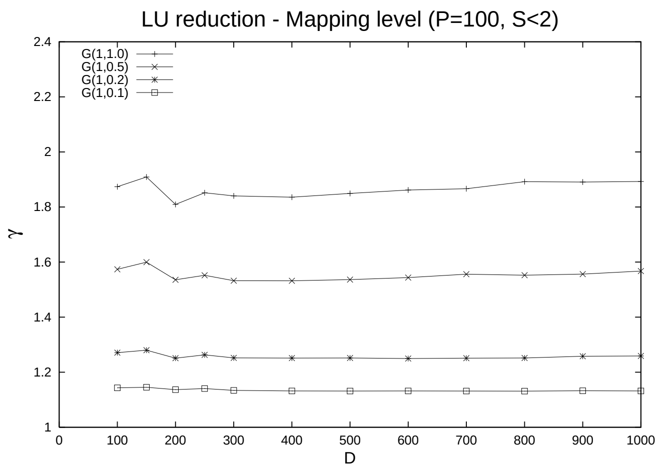


Figure 4.34: γ results for random workloads in LU programming level model

applications present this characteristic only layer by layer, that is perfectly enough to talk about a well-balanced computation.

Thus, as predicted at programming level when modeling the workloads (see section 4.2.2), the task loads are highly regular, showing in most cases a negligible deviation (see Table 4.2). In the table we can appreciate performance effects introduced in a very low level by the machine architecture. The execution times of tasks (sequential codes) become unstable only when the user task is sharing the processor time with operative system tasks. This effect never happens in the CrayT3E, as the operative system launches the user jobs in other free processors. In the Origin2000 (a cc-NUMA machine) it is noticed only when the number of processors used is equal to the maximum installed in the machine. The operative system is typically running in only one processor. Hence, only when this last processor must be shared with user processes, awful effects that degrades the user tasks performance appear (cache trashing, processes migration across processors, etc.). The Beowulf, representative of NOWs and low coupled systems, is the worst case. In these machines, most of the operative system tasks, and the MPI daemon operations, are executed locally in each node. Thus, the user tasks must share time with them. As the amount of processors increases, more communication and synchronization operations share the limited network bandwidth. Thus, their times increase. Moreover, the complexity of low level communication tasks also increases (collective operations, as barriers are a good example). Thus, the time of executing the same piece of code with the same data increases and becomes less predictable. This effect, typ-



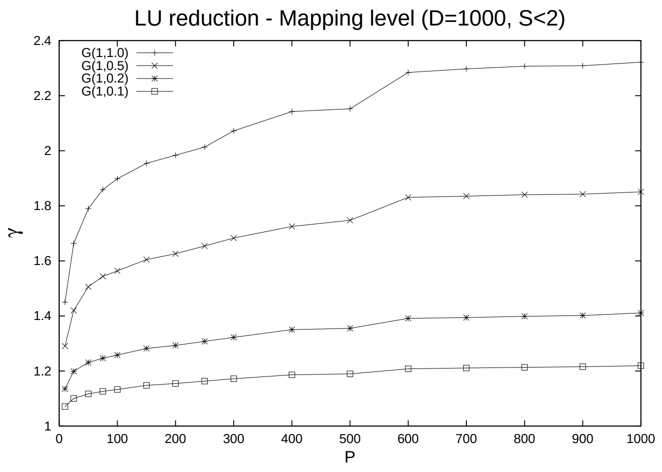


Figure 4.35: γ results for random workloads in LU programming level model

ical in low coupled systems (for which Grid computing is the extreme case), enforces an idea related to SP SA: potential beneficial effects can be obtained using hierarchical division of computations, in locally synchronized subparts (see e.g. [119, 118]).

However, the real workload variability, statistically measured, is really small even for the worst cases (saturated Beowulf):

$$\varsigma < 0.024; \quad \overline{\varsigma} \approx 0.005$$

This leads to extremely low performance losses for the SP versions.

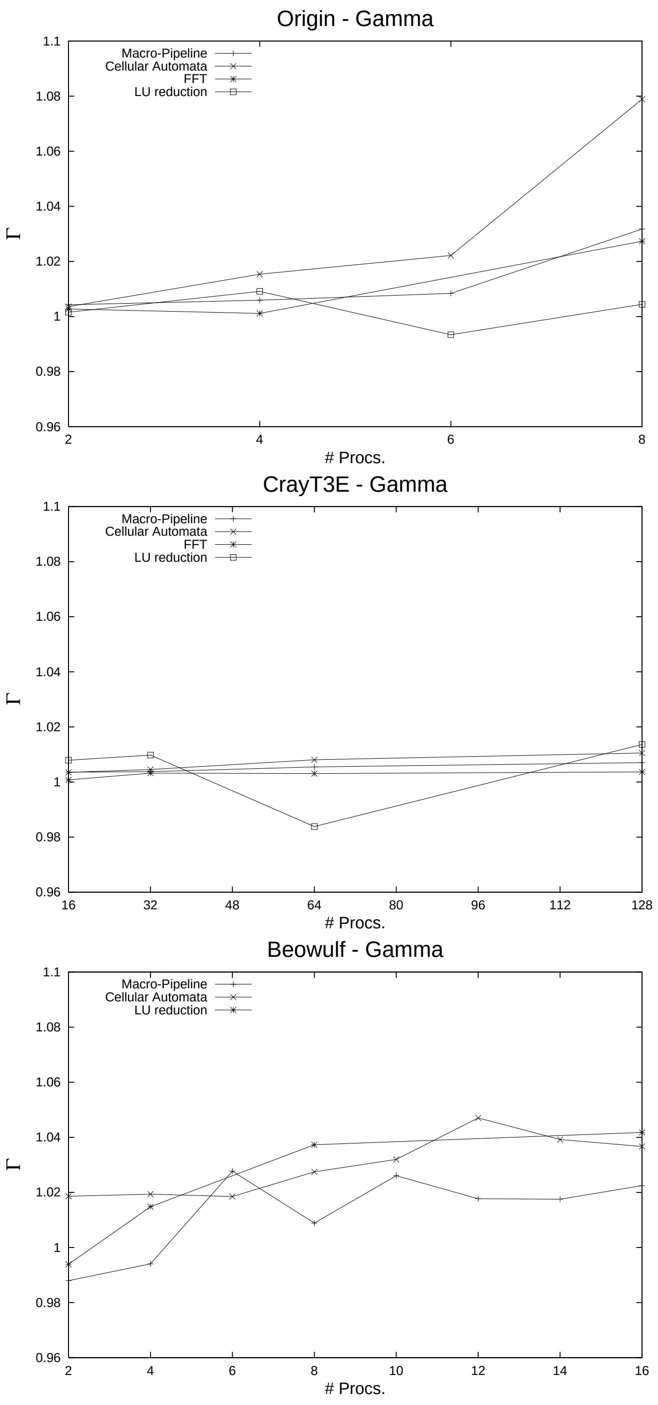


Figure 4.36: Γ results

Origin2000

	Macro-Pipeline		Cellular automata	
Proc.	μ	σ^2	μ	σ^2
2	23	<0.001	28	<0.001
4	23	<0.001	28	<0.001
6	23	<0.001	28	0.001
8	23	0.002	29	0.003

CrayT3E

	Macro-Pipeline		Cellular automata	
Proc.	μ	σ^2	μ	σ^2
16	52	<0.001	27	<0.001
32	52	<0.001	27	<0.001
64	52	<0.001	27	<0.001
128	52	<0.001	27	<0.001

Beowulf

	Macro-Pipeline		Cellular automata	
Proc.	μ	σ^2	μ	σ^2
2	21	0.003	27	<0.001
4	22	0.105	27	<0.001
8	23	0.136	28	0.002
16	23	0.288	28	0.002

Table 4.2: Statistical workload information for highly regular applications (ms.)

3. Very low performance degradation:

   In Fig. 4.36 we show the Γ plots that summarize the results obtained with real application codes in different machine architectures. We observe the general logarithmic tendencies when the applications scale up, predicted with the programming level models. However, with extremely low slopes due to the small relative deviation of the task loads. Irregularities in the plots are produced by different low level machine effects described below.

4. Machine architecture independence, and side effects:

   Different irregularities and strange effects in Γ plots are observed across machines (see e.g. the performance upgrading of LU reduction applications in CrayT3E and Origin2000 for some specific number of processors). All of them are easily explained by the different nature of machine architecture and operating system activities, that affect every application run.

   We observe that the most regular results are obtained in the CrayT3E, where the task loads are more stable and the communication costs are lower. The performance loss is less than 2% in the worst case for 128 processors. In the Origin2000, the barrier costs are comparatively higher, and it affects the performance. We also see the high impact of running the applications with the maximum number of processors available in the machine (8 processors in this case), when the user tasks share resources (as CPU) with the operative system tasks, that are typically running in only one processor. In the Beowulf system we appreciate the increasing costs that appear due to irregularities produced by task, operative system, and communication overlapping in every node. However, the performance degradation is still very low (less than 5% for the worst cases).

A remarkable case previously discussed is the LU reduction application. Recall the implementation considerations exposed in section 4.2.3. Even if no compiler optimization is used, the communication layer performs run-time optimizations when a collective communication primitive is followed by a barrier. This effect is observed for specific numbers of processors in the CrayT3E and Origin2000. In these machines, the MPI implementations are optimized by the vendor for the architecture and low-level hardware details. An improvement of performance, around 2%, is obtained in some cases.

Apart from this predictable irregularities, the performance degradation due to added dependences is proportional to hardware speed across machines. Specific machine effects with high impact in performance, affect in the same way to the NSP and the SP versions. For example, in the Beowulf system, we observe completely different communication time response when applications scale up from 6 to 8 processors (see Fig. 4.37). Nevertheless, they do not modify, or even improve, Γ results (the relative performance impact is decreased when a constant is added to both: NPS and SP execution times).

Thus, different architecture models do not create unexpected differences in the Γ tendencies. The general conclusions obtained from the results are the same in all cases. The real performance effects produced by changing the programming style or model to a restricted PPM, is independent of the machine architecture.

We conclude that: (1) General tendencies (e.g. logarithmic effect when scaling up) observed with synthetic graphs are found in real applications; (2) static, scalable applications are, in general, well balanced applications. Thus, as predicted with synthetic graphs, and the application specific graph models, the potential performance effect when programming these applications in SP programming models is extremely low; even when no SP specific optimization or run-time environment is exploited. For some applications (as LU reduction implemented with broadcast), highly structured synchronization is exploited by implementations at run-time level, even by non-specific NSP programming models as MPI.

4.2.5 Dynamic applications results

In this section we discuss the results of our study of γ and Γ for dynamic applications. Recall the experimental framework design discussed in section 4.2.1.

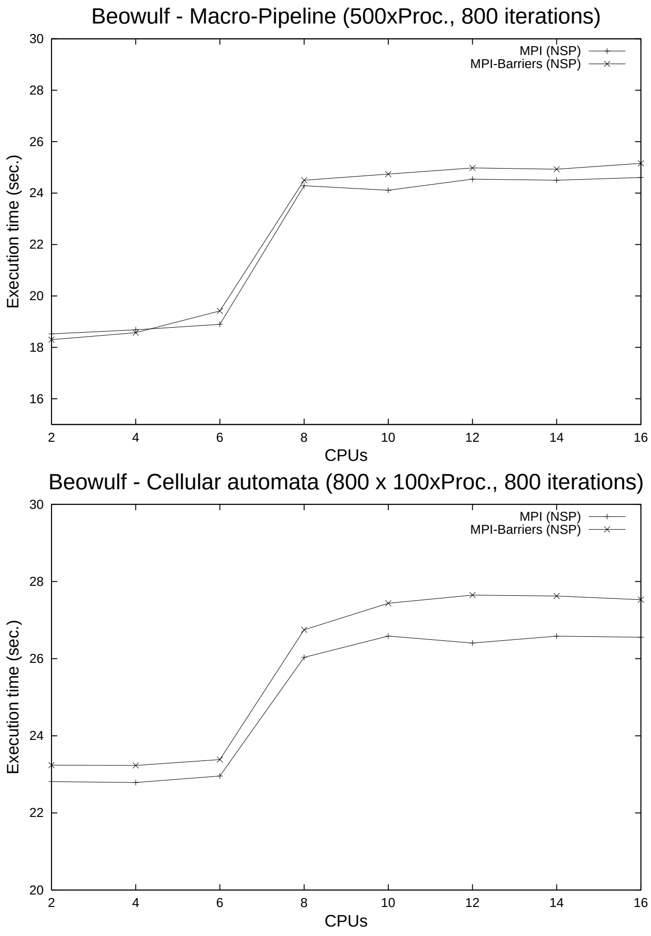


Figure 4.37: Execution times of some applications in the Beowulf system

Due to the data-dependent nature of these applications, we can only explore γ as a function of structures generated by specific applications for a given input data. Hence, we use *example* structures, representative of the typical structures generated by a given application. These example structures can be obtained at run-time by monitoring existing applications, or can be derived manually from the data structure and the code. The second method is clumsy, unaffordable for complex applications, and impossible when run-time decisions (as some ME or scheduling solutions) are inherent to the original code. For our experiments we have selected several available examples of task graphs generated manually or during run-time for two typical applications, representative of important and large application classes detected in the classification presented in section 2.6. Both are based on finite element solvers, and they represent the structures generated by typical iterative and direct solvers for sparse-matrix computations.

Iterative solvers and graph partitioning

Many finite element and PDE problems are solved by iterative methods applied to the sparse adjacency matrix that represents the problem graph. Structural engineering, chemical and physical phenomena simulations, and many other problems use these methods. The problem graph is distributed among the available processors by a partitioning algorithm that try to balance the load and minimize the communication due to links between graph nodes allocated in different processors. The solver algorithms apply the same computation for each iteration on the local nodes, and communicate the computed values that other processors need before the next iteration. Hence, for these problems, the computational load of a task is proportional to the number of nodes allocated in a given processor.

Given example input graphs and a partitioning algorithm, we can compute the data distribution for any number of processors. Thus, we can reconstruct the mapping level graph associated with the computation, including load estimations. The graphs can be used to estimate γ values for this class of applications.

We have selected six example graphs from the structural engineering field as study cases. The graphs are obtained from the *Everstine's collection*[2], to be found inside the *Harwell-Boeing collection* of sparse-matrices [58]. This public collection is available on the Matrix Market home page [146]. We have selected six graphs that present different structure patterns, and cover a wide range of nodes number, from the available in the full set (87,209,607,1005,1242,2680). From now on, we add the number of nodes after the name of each example for clarity. In Fig. 4.38 we show 3D models of the objects from which the matrices are obtained, and in Fig. 4.39 we show the sparse-matrices structures.

The graph partitioning algorithm selected is METIS (see e.g. [167]), that is a free and state-of-the-art multi-level partitioning software for unstructured graphs, that can be found in the METIS/ParMETIS home page [117]. We have used this software to partition the input graphs for 4,8,16,32,64 number of processors. The obtained data is processed to reconstruct the mapping level graphs, and to obtain the statistical information needed.

From the collected data we observe the following results:

1. Good load-balance:

   As the computational load is typically proportional to the number of nodes allocated in the local processor we can estimate the mean load and deviation with the number of nodes in each part. In Table 1 we show the workload

[2]These patterns were collected from various US military and NASA users of NASA's structural engineering package NASTRAN for use as a benchmark collection for variable bandwidth reordering heuristics. They have been widely used in benchmarks.

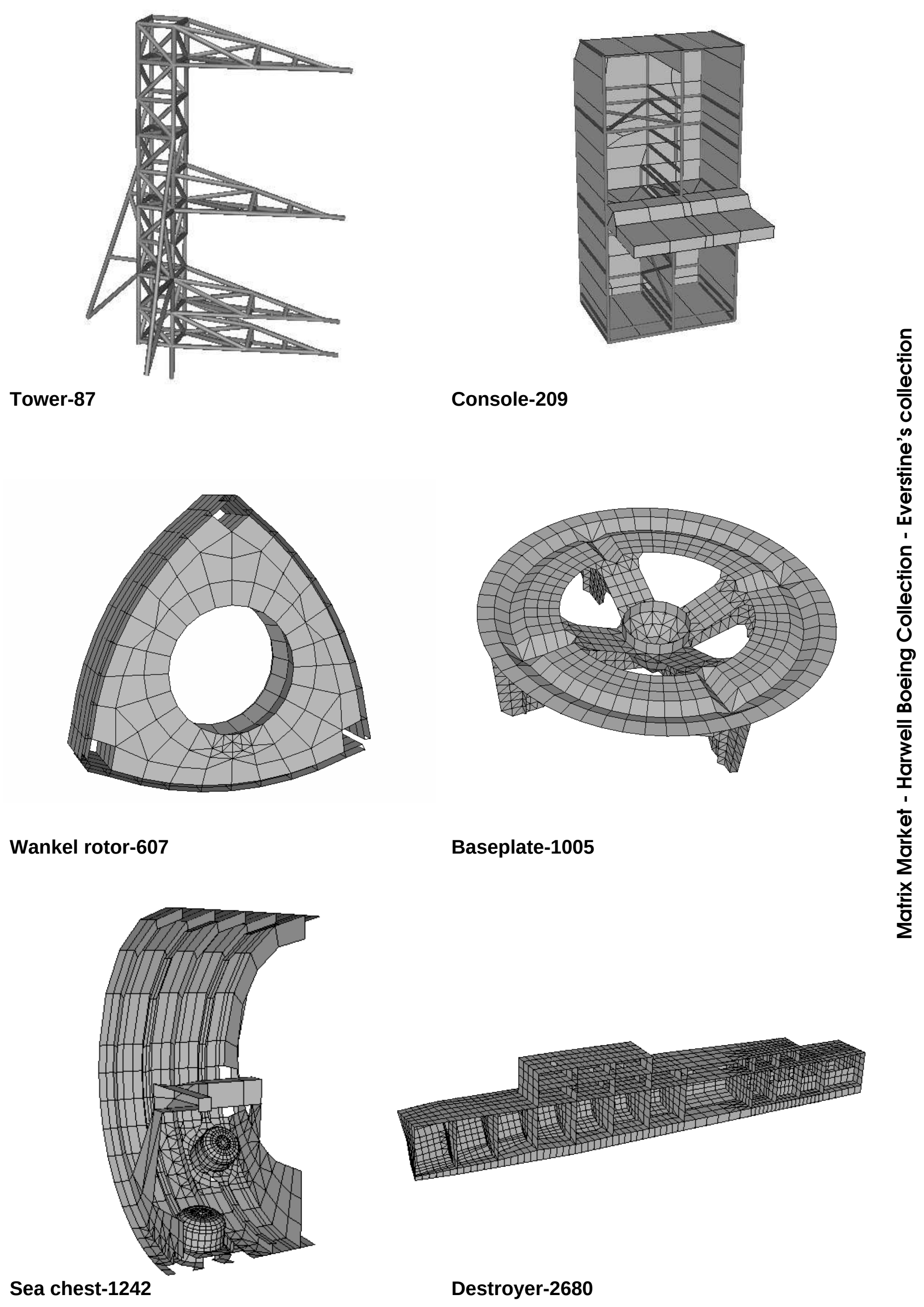


Figure 4.38: 3D models of the structural engineering examples

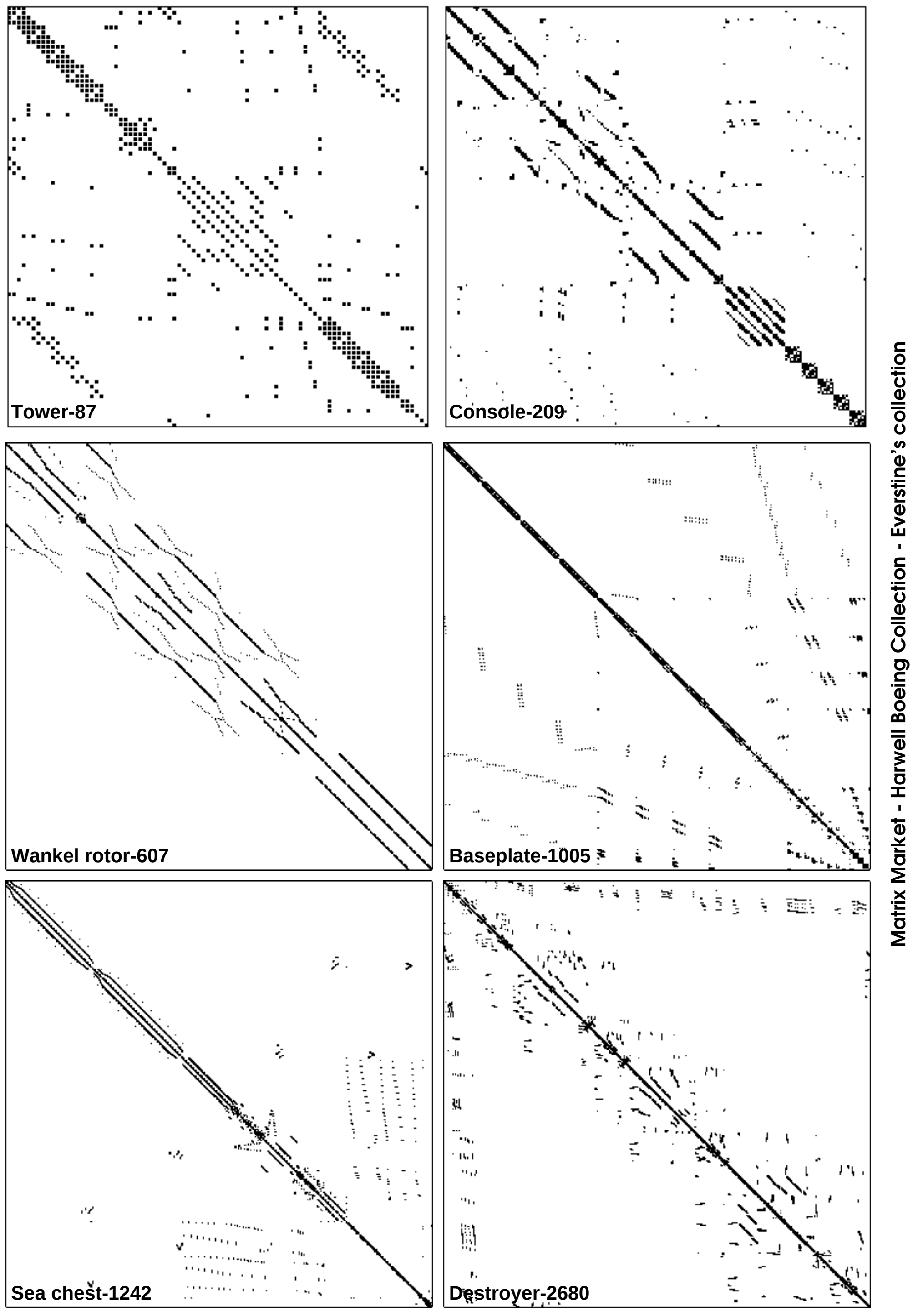


Figure 4.39: Sparse-matrices structure of the structural engineering examples

variability obtained for each example with a given number of processors. The graph partitioning methods are designed to create a well-balanced data

Example	# Procs.				
	4	8	16	32	64
Tower - 87	.0199	.1747	.2053	.1658	.5723
Console - 209	.0249	.0127	.1469	.1875	.1457
Wankel rotor - 607	.0505	.0267	.0422	.0466	.3334
Baseplate - 1005	.0183	.0199	.0212	.0442	.0663
Sea chest - 1242	.0204	.0190	.0176	.0176	.0680
Destroyer - 2680	.0155	.0172	.0241	.0243	.0306

Table 4.3: Estimated ς for partitioned iterative solver task graphs

partition. We observe very low variabilities, as the partitioning method is performing quite well. The only cases where the values are higher than a very small bound $\varsigma > 0.1$, are found when the number of nodes per processor is very low, and the parallelism exploitable is very poor (see e.g. the smaller example, Tower-87). For normal real computational problems, the load will be well distributed, leading to minimum performance effect when SP-ization is applied.

2. Regular structures:

   In Fig. 4.40 we present an example of a small mapping level graph generated for some iterations with the Sea chest-1242 example, mapped for 8 processors. Recall that reducing the number and load of communications among processors, and promoting some neighborhood, is an objective of the partitioning algorithms. Hence, we find that the structures obtained are very similar to the synthetic meshes studied in section 4.1.3. The shape of these mapping level task graphs is highly regular. They have a fixed number of nodes per layer (the number of processors for which the partition is computed) and each layer represents an iteration of the solver. The number of edges per node is determined by the partition computed. In Fig. 4.41 we show the S parameter measured for the generated graphs. Its values are found in a narrow range, and the general trend is that S increases logarithmically with the number of processors. When the number of nodes per processor is very low, we find again a case where there is not enough parallelism available and the number of communications decrease (see e.g. the Tower-87 example plot). Thus, the increase of P values is somehow compensated by the increase of S. This effect together with the small load variability observed, predict very low γ values for this applications type.

3. Load distribution correlation:

   There exists an important correlation in how the loads are distributed

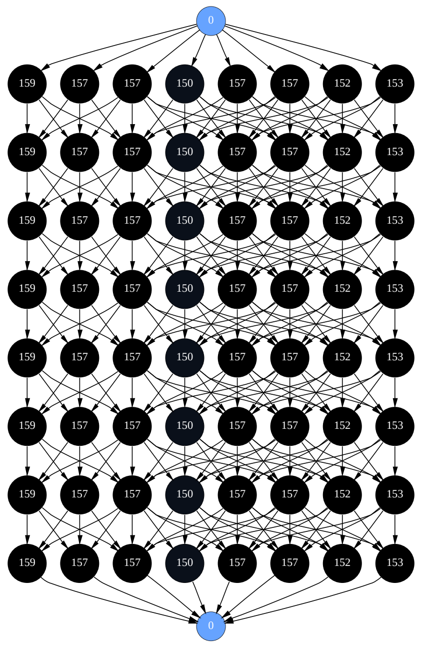


Figure 4.40: Example task graph: Sea Chest-1242, 8 processors, 8 iterations

across the graph. Different layers represent different iterations of the same computations. Thus, when the partition assigns different number of model nodes to different processors, the load distribution has a vertical correlation with the topology. If an implementation level graph is considered, where communication costs are accounted, the correlation can be even more noticeable. This correlation may produce a beneficial γ reduction, especially in these cases of low workload variability along several iterations (see section 4.1.3).

4. Negligible performance degradation:

   In our set of experiments with these graphs, using the loads estimated with the number of nodes allocated to each task in the partition, with have found no critical path value increase due to SP-ization except in cases of assuming high load variabilities for nodes. This will not be the case for this kind of applications, where a processor is replicating exactly the same task in each iteration. The real loads have extremely low variabilities in different instances of the same node. In this case, the SP-ization effect is typically neglected.

The conclusion is that if the partitioning algorithm is producing a good partition and no other run-time or machine details severely affect the load balance,

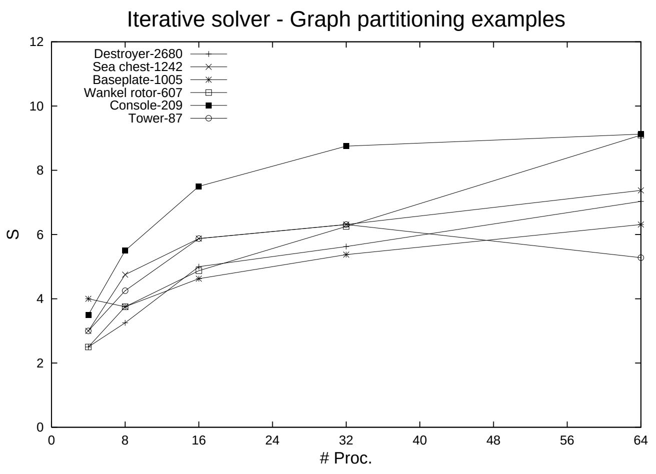


Figure 4.41: S parameter for partitioned iterative solver task graphs

iterative solvers for this kind of sparse-matrices can be programmed in nested parallelism programming models with negligible loss of performance. The extra synchronizations and barrier costs are not accounted, but SP programming advantages are also not considered. For example, the knowledge of the global communication structure may still be exploited to improve communication performance (see e.g. [57]).

Domain decomposition and sparse-matrix factorization

The DIANA software [55] is oriented to the structural engineering field. It includes several methods, based on direct solvers for sparse-matrices, to compute solutions to finite-element problems. The software include domain decomposition and sparse-matrix factorization modules [123]. Our example task graphs were obtained during the research conducted to parallelize the DIANA software package. They represent examples of domain decomposition and sparse-matrix factorizations of real data with different input sizes. The factorizations were implemented using a tool for parallel execution of unstructured problems (Tgex [124]).

For each of these graphs we apply the Algorithm2 SP-ization technique, and we measure γ comparing the *cpv* of both graphs. First, we apply different synthetic random workloads to the nodes. Then, we compare the results, with Γ estimations obtained when real workloads (measured at run-time) are considered in the nodes.

The number of nodes in our example graphs are: 59, 113, 212, 528, 773 and 2015. In Fig. 4.42 we showed the 113 nodes graph before and after the transformation. The first part of the graph, before it achieves its maximum width, is the domain decomposition phase. The rest of the graph represents the sparse-matrix factorization. An indication of the real workload distribution is shown; darker and bigger nodes represent more loaded nodes.

After experiments with these graphs the following results are exposed:

1. Topology regularities:

   Due to the nature of the application, the topologies present some regularities. Fig. 4.43*(a)* shows γ measures for the six example graphs when using modeled workloads with Gaussian distributions, as a function of the relative synchronization density (R_s) parameter. They approximately follow the expected tendency detected with synthetic random topologies in section 4.1.2; γ decreases with R_s. However, if we compare these plots with the equivalent plots for synthetic random topologies (see Fig. 4.5), we found that the points are below the expected mean values for completely random samples. The topologies of these sparse-matrix computations are not completely irregular, and they are not in the worst case topologies.

2. Workload distribution regularities:

   In Table 4.4 we show statistical information about the real task loads. We find that the workload is highly deviated. Few nodes concentrates the biggest part of the overall load. Nevertheless, we can see in the graphical

# nodes	ς
59	2.1
113	3.0
213	1.4
528	2.0
773	7.1
2015	2.6

Table 4.4: Statistical information of real workloads for sparse direct solver.

   representation of the graphs that the highly loaded nodes are not randomly distributed (see e.g. the position of darker nodes in the example graph presented in Fig. 4.42). We find some of them distributed among the beginning nodes of the domain decomposition phase, and some other ones at the first layers of the factorization phase. In Fig. 4.43*(b)* we show Γ estimations for the graphs considered with real workloads measured during execution. Information about the number of nodes n and the relative deviation (ς) is added to each point.

   Measured real workloads showed higher deviations than any of the Gaussian models used for each topology. However, Γ values are very low, and much better than expected. The reason for this is that real workloads are not completely distributed at random across the task nodes. They are

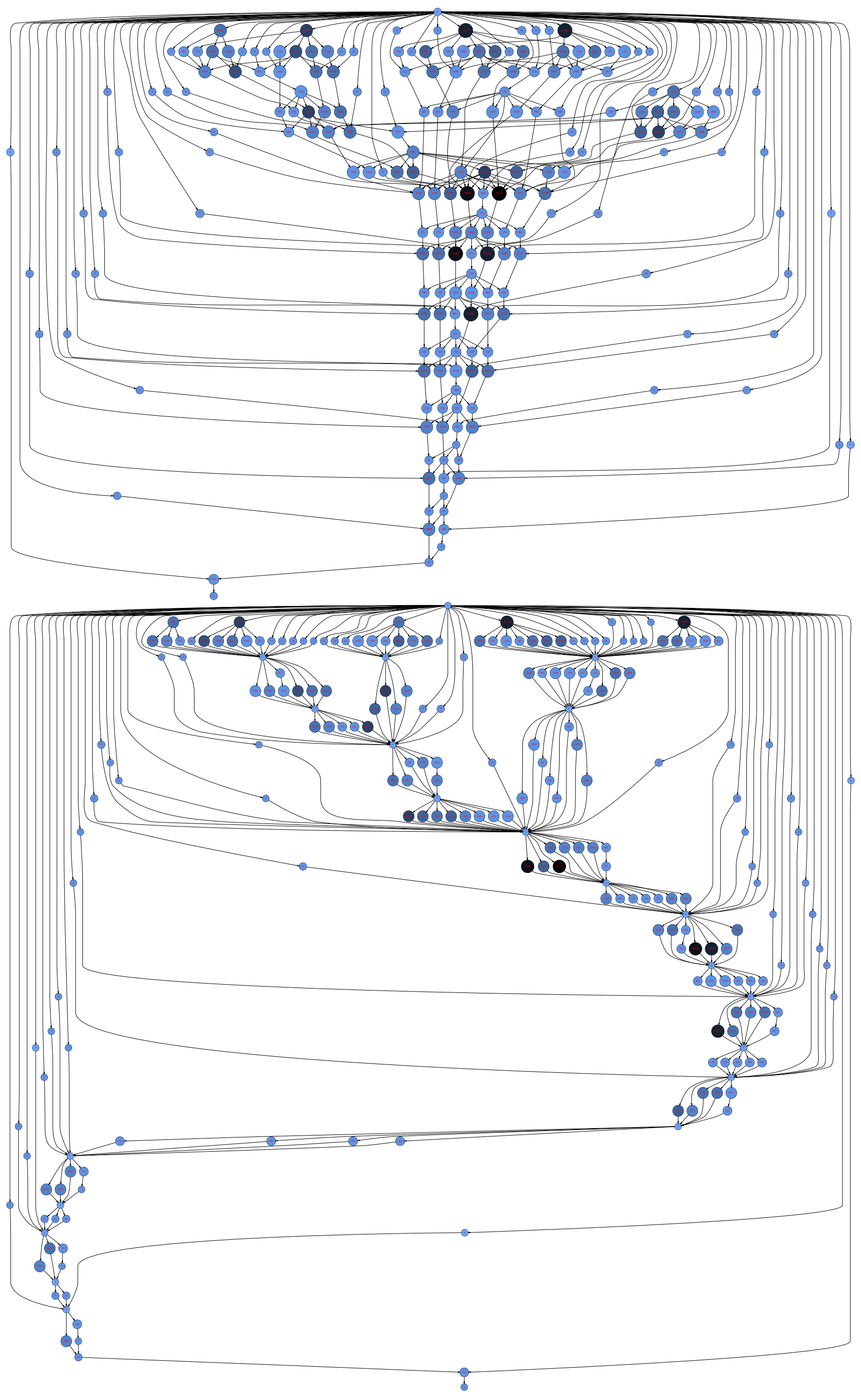


Figure 4.42: Example of a dynamic application graph and its SP version.

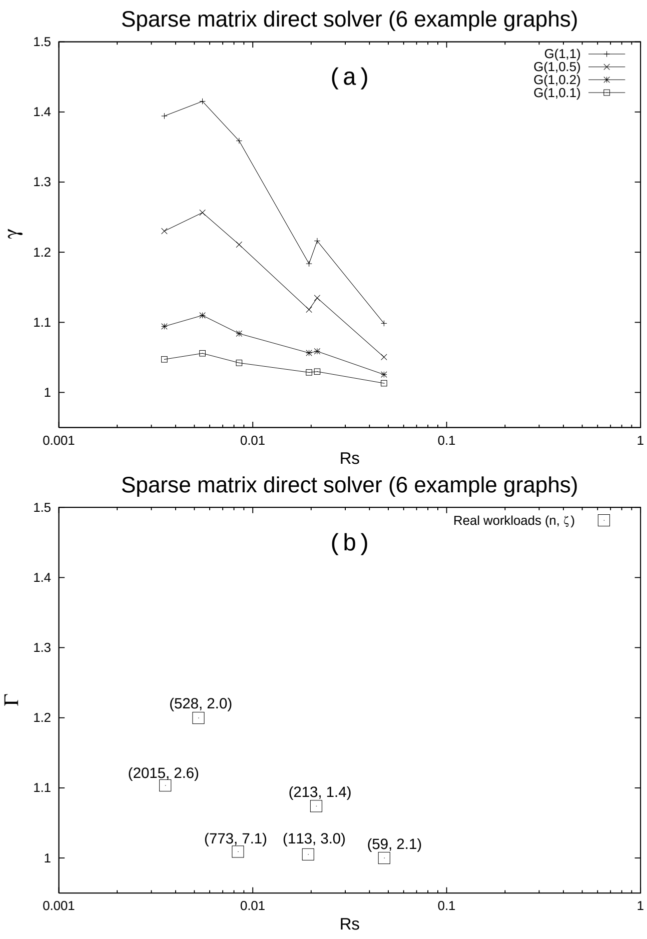


Figure 4.43: Results for sparse direct solver example graphs.

highly unbalanced, but there still exists a high correlation between topology structure (layers and local synchronization patterns) and workloads. This correlation produces a beneficial impact on γ, similar to the one presented for correlated workload meshes in section 4.1.2.

Workload parameters μ, σ are not enough to get an accurate estimation of the impact of SP-ization on a given unstructured topology. In general, the correlation between highly loaded nodes and layers will produce an improvement in performance when mapping to nested parallelism structures.

For this set of experiments we find that sparse-matrix solvers generate task graphs with enough topology and workload regularities to minimize the performance impact of SP-ization. However, workload parameters are not enough to get an accurate estimation of the impact of SP-ization on a given unstructured topology. In general, the correlation between highly loaded nodes and layers will produce an improvement in performance when mapping to nested parallelism structures.

4.2.6 Conclusions about real application results

In this section we have tested several types of real applications, comparing the real performance Γ in different machine architectures with tendencies observed during the γ study of their structures. A first conclusion is that the main γ effects detected with synthetic graphs (e.g. logarithmic effect when scaling up) are present in real applications. At the same time we find that they are propagated to the run-time level. However, the real applications present high regularities that minimize the performance impact of programming them in an SP parallel programming model. Even typical irregular applications use load-balancing or mapping techniques that create important topological and workload regularities. The small performance effects introduced at the programming level are independent of the underlying architecture. In many cases, implementation details and low-level machine effects appear to have more impact on the final performance than the choice of a restricted SP parallel programming model.

4.3 Summary

In this chapter we have presented an experimental framework to determine empirically the potential and real impact of using a nested-parallel SP programming model. First, we have discussed how to build synthetic workload distributions, based on i.i.d. random workloads, that can be used with any synthetic or real topology generated along the experimental study. We have introduced the methodology to construct random graphs in order to test a sample of the graph space, and synthetic meshes of nodes that represent regular applications. The graph meshes are used to systematically test the γ effects related to simple graph parameters that represent characteristics inherent to the application (synchronization density) or typical mapping variables (degree of parallelism or number of iterations). The study of the graph meshes includes the identification and analysis of other impact factors (as synchronization unbalance or workload to topology correlation).

For real applications, we have presented a criteria to select representative study cases, and to select machine models for a real performance study; along with the implementation techniques and tools to allow the NSP to SP structure comparisons. We have discussed the modeling techniques to: (1) extract task graphs from real applications at different levels of detail, and (2) construct graphs representing irregular or dynamic applications from the input-data structures.

Finally, we have introduced the framework to carry out a *cpv* study on all the generated or extracted graphs to obtain $\overline{\gamma}$ results and compare them with real Γ measures.

The results obtained in this study point out that the expected values of γ, used as an indicator of the potential performance impact of using an SP PPM, follow predictable tendencies. Indeed, these tendencies are determined by simple and easily measurable graph parameters. The increase of the degree of parallelism, measured with P, produces a general under-logarithmic increase on γ. The graph depth level, measured with D, has a completely limited effect on γ, except in pathological structures, for which we present a formal description and a possible indicator of the potential pathology factor (ω). The synchronization density represented by S (or R_s) is the topological parameter with the higher impact on γ. For the very small values found in sparse random or highly irregular graphs ($S < 2$), SP-ization techniques that exploit local synchronization techniques, as Algorithm2, may produce SP-forms with small *cpv* increment. Values of $S > 2$ have a quick negative exponential limiting effect on γ. However, around values of $S = 2$, the SP-ization techniques studied present the worst results, and the lower predictability for random topologies. As discussed in section 3.3.1, these could be the structures more suitable for other mixed transformation techniques based on both, added dependences and duplication of nodes. Nevertheless, the workload distribution is the critical factor for SP-ization performance impact. Its variability and possible correlation with topology highly determine the main γ tendencies. A well-balanced workload distribution immediately reduces or even neglects the potential increase of the *cpv* after an SP-ization. Moreover, most of the tendencies previously discussed are only fully appreciated for highly deviated workloads. Fortunately, real applications present very good workload conditions for SP-ization. In other case, the scalability and flexibility of the parallel application would be compromised. The experimentation shows that real workloads are usually well balanced and correlated with the graph topology. The values found present better characteristics than the synthetic workload models used during the first phase of our study, leading to negligible performance impact when using SP form synchronization structures to program real applications.

All these tendencies are propagated to the run-time low level. Even some classes of important irregular applications use data-partition and load-balancing techniques to produce scalable codes. These techniques create enough topology or workload regularities to neglect the potential performance degradation when programmed with a nested-parallel, SP, programming model. Indeed, some machine effects derivated from different hardware or parallel tools implementations appear to have more impact on the performance than using SP-restricted synchronization structures. We conclude that our experimental study clearly points out that using an SP parallel programming framework is a safe choice for most parallel applications, and potentially bad study-cases can be easily predicted.

Chapter 5

Conclusion

The line it is drawn
The curse it is cast
The slow one now
Will later be fast
As the present now
Will later be past
The order is
Rapidly fadin'.
And the first one now
Will later be last
For the times they are a-changin'.

The Times They Are a-Changin', 1963
BOB DYLAN

The field of parallel programming appears to be not yet mature enough to produce a consistent and established software development methodology. Parallel architectures and programming models still lack a common development direction based on a standard machine and programming model (like Von Neumann's in sequential programming). While machines and low level programming interfaces are oriented to exploit the maximum parallelism and performance in an application, more abstract programming models accept restrictions of expressive power, in terms of their SA, to obtain those analyzability characteristics that help in the design, programming, mapping, implementation and debugging tasks. This expressiveness vs. analizability trade-off needs to be carefully analyzed in order to establish which characteristics of a model are responsible for its good and bad properties of it, in terms of software development, implementation portability, and performance.

Being the determining factor of a programming model that characterizes the above trade-off, in this dissertation we have studied the SA concept and its relation to the properties of PCMs at different abstraction levels. We have classified the SA of well-known models and applications, and we have proposed and studied an important class: the SP, also known as Series-Parallel, or Nested-Parallelism SA. We have found that in the design of a PCM, the decision to restrict or not to restrict SA to SP class, is a critical one. SP vs. Non-SP is the SA barrier where important analyzability properties appear or disappear. Consequently, we have presented an in-depth study on the impact of the expressive restrictions associated to SP programming models to support our thesis that SP restricted models are the best choice to obtain both: highly beneficial software development characteristics, and a good level of expressive power for general-purpose parallel programming.

We have used a three-way approach to study the relevancy of the SA concept and the SP restriction for parallel programming models: (1) A conceptual study of SA, where existing programming models and applications are studied; (2) a theoretical approach, where the SP vs. NSP structures are deeply studied with the aid of graph theory; and (3) an experimental study, where empirical results are presented to validate our hypothesis about the potentially negative performance impact of using restricted SP models. In our study of SAs from these three points of view, we have made several contributions and we have produced significant results, obtaining relevant conclusions in support our thesis.

5.1 Contributions

In particular, the following contributions are made in this dissertation:

- SA description and classification.

  We have introduced the SA concept, and we have shown how it is related with the expressive power (EP), software engineering (SEC) and analyzability (AC) characteristics of a PCM/PPM, through a conceptual review and classification of well-known existing models at different abstraction levels.

- Applications classification in terms of SA.

  We have classified parallel applications in terms of the SA they naturally map to. The classification is useful for detecting application types that do not map directly to restricted synchronization PPMs, and to choose example applications, representing their SA classes, for further study of the application to PPM mapping techniques. Some mapping strategies are also discussed for the relevant classes.

- NSP vs. SP graph theoretical study.

  In order to assess the performance loss associated with the choice of an SP-restricted PPM for an inherently NSP problem, we have performed a graph theoretical study of the SP and NSP structures. We have presented a number of techniques to transform NSP structures to SP approximations that introduce minimum changes in topology or performance, including new full graph algorithms. Methods and metrics to measure the impact of such transformations in topology and potential increase of the critical path have been proposed.

- Analysis framework for performance impact of SA transformations.

  We have introduced an analysis framework to predict the performance loss at the programming abstract level as a function of SA. Given the relative importance of condition synchronization, we have specifically applied the approach to predict the performance differences of using NSP vs. SP-restricted programming models. The framework is based on the use of graph theory, topology classes, and task workload metrics. We have measured performance differences (γ) in terms of critical path.

- Simple graph modeling techniques for applications.

  We have introduced methods to model applications and workload with graphs, at different detail levels. The significance of the contribution is to show that very simple graphs, easily derived from specifications or even from real code, are accurate enough to predict tendencies and behavior of applications when synchronization structures are transformed to map them to different SA classes.

- Full experimental study using real applications.

  The study consists on a comparison of using programming models or languages in different SA classes to implement real applications, including the effects of typical implementation trajectories. Here we do not restrict ourselves to the highest abstraction levels (γ), but we use the above framework to discuss the performance effects of various mappings and implementation issues at lower level (Γ).

5.2 Conclusions

The contributions presented strongly suggest the SP SA as the most promising design concept for new portable, efficient and easy-to-use parallel programming models. PPMs in the SP SA class offer important advantages in terms of software engineering and analyzability characteristics, not available for less restricted models in the NSP class, with a modest trade-off regarding expressive power. The conclusions of this thesis are:

- SP SA leads to formal methods of software development and verification. SP restricted models and structures are associated with SP algebras and an extended automata theory. At the same time, more efficient scheduling, compiling and mapping techniques exist for SP restricted structures than for NSP structures.

- From the study of SA of existing parallel programming models we have found that only PPMs/PCMs that restrict CS structures include an easy-to-use and accurate cost model that may help in automatic mapping decisions. This is critical for portability of programs to different architecture models.
- Many application classes and parallel programming paradigms directly map to SP structures. For those application classes that do not directly map to SP models, systematic transformation techniques that minimize the potential performance impact have been proposed. Many examples of how to use them have been presented for synthetic and real application structures.
- Simple application parameters, like the maximum degree of parallelism, as well as workload characteristics may be used to predict the impact of an NSP to SP transformation, at different levels of detail with very simple cost models. Such predictions are accurate enough to predict the performance asymptotical behavior of different mappings of an application to SP programming structures.
- The performance degradation associated with SP programming is mainly related to poorly balanced and unstructured computations, that are difficult to program, verify and debug. In our application classification and experiments we find that these structures are far from typical or even inappropriate for parallel programming in general. High performance unstructured computations are programmed with hard-wired scheduling and load-balancing techniques that transform them in more structured and well-balanced computations, more suitable for SP programming.

The Synchronization architecture concept, and this study, validate some research directions previously introduced in restricted SA models (as e.g. BSP). Many previously intuitive ideas about the impact of SP programming have been formally or empirically verified in this study. This may help to focus the attention of parallel programming languages and models designers to the SP concept. SP, or nested-parallelism may lead to a more focused research direction to fill the gap between the two extreme points of the parallel programming world: machine architecture vs. high-level programming. The development of new and more abstract languages for SP restricted models may include new compiling and mapping techniques that exploit many beneficial features nowadays scattered among different programming models and their implementations.

Bibliography

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, J. Mellor-Crummey, S. Warren, and C.W. Tseng. Requirements for data-parallel programming environments. *IEEE Parallel & Distributed Technology*, pages 48–58, Fall 1994.

[2] A. Aggarwal, A.K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21. ACM, 1989.

[3] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[4] J.K. Aggarwal and P. Chillakanti. Software for parallel computating - a perspective. In A.Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*, chapter 12, pages 357–375. International Thomson Computer Press, 1996.

[5] K. Al-Tawil and C.A. Moritz. LogGP quantified: The case for MPI. In *Proc. 7th IEEE International Symp. on High Performance Distributed Computing (HPDC-7)*, Chicago, IL USA, Aug 1998.

[6] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proc. SPAA'95*, pages 95–105, Santa Barbara, CA USA, 1995. ACM.

[7] V.A.F. Almeida, I.M.M. Vasconcelos, J.N.C. Árabe, and D.A. Menascé. Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. In *Proc. of Supercomputing'92*, pages 683–691, Minn., MN, Nov 1992. IEEE.

[8] R. Alpert and J. Philbin. cBSP: Zero-cost synchronization in a modified BSP model. Technical Report Technical Report 97-054, NEC Research Institute, Feb 1997.

[9] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, Mar 1983.

[10] C. Ash. *The probability tutoring book: an intuitive course for engineers and scientists (and everyone else!).* IEEE Press, 1993.

[11] H. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, pages 74–84, Jul-Sep 1998.

[12] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory.* Universitext. Springer, 2000.

[13] J.L. Balcazar, J. Díaz, and J. Gabarró. *Structural Complexity I.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2nd edition, 1995.

[14] W. Bein, J. Kamburowski, and F.M. Stallman. Optimal reductions of two-terminal directed acyclic graphs. *SIAM Journal of Computing*, 6:1112–1129, 1992.

[15] S. Ben Hassen and H. Bal. Integrating task and data parallelism using shared objects. In *ACM ICS'96 Philadelphia*, pages 317–324, 1996.

[16] G. Bilardi, K.T. Herley, and A. Pietracaprina. BSP vs. LogP. In *Proc. 8th ACM symposium on Parallel algorithms and architectures (SPAA'96)*, pages 25–32, Padua, Italy, Jun 1996. ACM.

[17] J. Bircsak, P. Craig, R.L. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C.D. Offner. Extending OpenMP for NUMA machines. In *Proc. Supercomputing 2000*, page 48. ACM, 2000.

[18] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[19] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of 5th PPoPP*, pages 207–216. ACM, 1995.

[20] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. Annual Symposium on FoCS*, pages 356–368, Nov 1994.

[21] H. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, 1994.

[22] H.L. Bodlaender and B. de Fluiter. Parallel algorithms for series parallel graphs. In *Proc. 4th Annual European Symposium on Algorithms. ESA'96*, volume 1136 of *LNCS*, pages 277–289. Springer, 1996.

[23] H.L. Bodlaender and B. van Antwerpen-de Fluiter. Parallel algorithms for series parallel graphs and graphs with treewidth two. *Algorithmica*, 29:534–559, 2001.

[24] C. Boeres, V.E.F. Rebello, and D.B. Skillicorn. Static scheduling using task replication for LogP and BSP models. In *EuroPar'98*, volume 1480 of *LNCS*, pages 337–346. Springer, 1998.

[25] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library - design, implementation, and performance. In *Proc. IPPS/SPDP'99*, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.

[26] G.H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. HPDC'96*, pages 243–252. IEEE, 1995.

[27] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: design, implementation, and performance results. In *Proc. of the conference on Supercomputing '93*, pages 351–360, Portland, OR USA, Nov 1993. ACM.

[28] A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: A Survey.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, 1999.

[29] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[30] T. Bräunl. Parallaxis-III: architecture-independent data parallel processing. *Trans. on Soft. Eng.*, 26(3):227–243, Mar 2000.

[31] P. Brinch Hansen. *Studies in Computational Science: Parallel Programming Paradigms.* Prentice Hall, 1995.

[32] P. Brinch Hansen. An evaluation of high performance Fortran. *ACM SigPlan*, 33(3):57–64, Mar 1998.

[33] P. Brinch Hansen. An evaluation of the message-passing interface. *ACM SigPlan*, 33(3):65–72, Mar 1998.

[34] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560 – 599, Jul 1984.

[35] D.K.G. Campbell. A survey of models of parallel computation. Technical Report YCS-97-278, Department of Computer Science, University of York, Mar 1997.

[36] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proc. Supercomputing 2000*, page 5. ACM, 2000.

[37] N. Carriero and D. Gelernter. LINDA in context. *Communications of the ACM*, 32(4):444–458, Apr 1989.

[38] N. Carriero and D. Gelernter. A computational model of everything. *Communications of the ACM*, 44(11):77–81, Nov 2001.

[39] H. Cha and D. Lee. A hierarchical BSP model supporting processor locality. In *Proc. ICPADS'97*, pages 20–27. IEEE, 1997.

[40] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[41] B. Chapman and H. Zima. Extending HPF for advanced data-parallelism applications. *IEEE Parallel & Distributed Technology*, pages 59–70, Fall 1994.

[42] The Cilk Project.
WWW. On http://supertech.lcs.mit.edu/cilk/.

[43] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989.

[44] M. Cole. Frame: an imperative coordination language for parallel programming. Technical Report EDI-INF-RR-0026, Division of Informatics, University of Edinburgh, Sep 2000.

[45] M. Cole. Skeletal parallelism homepage. WWW, 2001. On http://www.dcs.ed.ac.uk/home/mic/skeletons.html.

[46] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178. ACM, 1989.

[47] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, McGraw-Hill, 2nd edition, 2001.

[48] CORPORATE The MPI Forum. MPI: a message passing interface. In *Proc. of the conference on Supercomputing'93*, pages 878–883. ACM, 1993.

[49] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proc. 4th ACM PPoPP*, pages 1–12, San Diego, CA, USA, May 1993.

[50] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken. LogP: a practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, Nov 1996.

[51] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kauffman Publishers, Inc., 1999.

[52] M. Danelutto. HPC the easy way: new technologies for high performance application deployment (extended abstract). In *Proc. Euro-PDP'03.* Computer Society, IEEE, 2003.

[53] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional skeletons for parallel coordination. In *Europar'95*, LNCS, pages 55–69, 1995.

[54] K.M. Decker and M.J. Johnson. Application specification and software reuse in parallel scientific computing. *IEEE Concurrency*, pages 71–77, Apr-Jun 1998.

[55] DIANA FE Program. WWW, Jan 2000. On http://www.diana.tno.nl/.

[56] C. Ó Dúnlaing. Some parallel geometric algorithms. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, volume 4 of *Series on Parallel Computation*, chapter 5. Cambridge University Press, 1993.

[57] S.R. Donaldson, J.M.D. Hill, and D.B. Skillicorn. Exploiting global structure for performance on clusters. In *Proc. IPPS/SPDP'99*, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.

[58] I. Duff, R.G. Grimes, and J.G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release i). Technical Report TR/PA/92/86, CERFACS, Oct 1992.

[59] R. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10, 303–318 1965.

[60] J. Eisenbiegler, W. Lwe, and A. Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *Proc. Advances in Parallel and Distributed Computing Conference (APDC'97).* IEEE, 1997.

[61] D. Epsstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98:41–55, 1992.

[62] A. Fahmy and A. Heddaya. Communicable memory and lazy barriers for bulk synchronous parallelism in BSPk. Technical Report BU-CS-96-012, Boston University, Sep 1996.

[63] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET–UCT series–parallel graphs on two processors. *Theoretical Computer Science*, 162:323–340, Aug 1996.

[64] M.J. FLynn. Some computer organization and their effectiveness. *IEEE Transactions on Computers*, Sep 1972.

[65] S. Fortune and J. Willie. Parallelism in random access machines. In *Proc. 10th Annual Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[66] I. Foster. Task parallelism and high-performance languages. *IEEE Parallel & Distributed Technology*, pages 27–36, Fall 1994.

[67] I.T. Foster and K. Mani Chandy. Fortran M: a language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26, 24–35 1995.

[68] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Feb 1992.

[69] A.J.C. van Gemund. Compiling performance models from parallel programs. In *Proc. 8th ACM Int'l Conf. on Supercomputing*, pages 303–312, Manchester, Jul 1994.

[70] A.J.C. van Gemund. *Performance Modeling of Parallel Systems.* PhD thesis, TU Delft, 1996.

[71] A.J.C. van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, Jul 1997.

[72] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. Technical Report TR-10-92, Center for Research in Computing Technology, Harvard University, Cambridge, Massachussets, 1992.

[73] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambrige University Press, 1988.

[74] P.B. Gibbons. A more practical PRAM model. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, 1989.

[75] P.B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACMSIAM SODA*, pages 638–648, 1994.

[76] P.B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. 9th annual ACM symposium on Parallel algorithms and architectures (SPAA'97)*, pages 72–83, Newport, RI USA, Jun 1997. ACM.

[77] P.B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write asynchronous PRAM model. *Theoretical Computer Science*, 196:3–29, 1998.

[78] G.H. Golub and C.F. van Loan. *Matrix Computations.* The John Hopkins University Press, 3rd edition, 1996.

[79] G.H. Golub and J.M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, 1993.

[80] J.A. Gonzalez, C. Leon, F. Piccoli, M. Printista, J.L. Roda, C. Rodriguez, and F. Sande. Groups in bulk synchronous parallel computing. In *Proc. 8th Euromicro Workshop on PDP'2000*, pages 244–251. IEEE, 2000.

[81] J.A. Gonzalez, C. Leon, F. Piccoli, M. Printista, J.L. Roda, C. Rodriguez, and F. Sande. Predicting the time of oblivious programs. In *Proc. Ninth Euromicro Workshop on Par. and Dist. Processing (PDP'01)*, pages 363–368. IEEE, Feb 2001.

[82] A. González-Escribano, V. Cardeñoso, and A.J.C. van Gemund. Conversion from NSP to SP graphs. Technical Report TR-DINFO-01-97, Universidad de Valladolid, Valladolid (Spain), Jan 1997.

[83] A. González-Escribano, V. Cardeñoso, and A.J.C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st Euro-PDS Int'l Conf. on Parallel and Distributed Systems*, pages 251–256, Barcelona, Jul 1997.

[84] A. González-Escribano, A.J.C. van Gemund, and V. Cardeñoso. A new algorithm for mapping DAGs to series-parallel form. Technical Report IT-DI-2002-2, Dpto. Informática, Univ. Valladolid, Apr 2002.

[85] A. González-Escribano, A.J.C. van Gemund, and V. Cardeñoso-Payo. Mapping unstructured applications into nested parallelism. In J.M.L.M. Palma, J. Dongarra, V. Hernández, and A.A. Sousa, editors, *High Performance Computing for Computational Science - VECPAR 2002*, number 2565 in LNCS, Porto (Portugal), 2003. Springer.

[86] A. González-Escribano, A.J.C. van Gemund, V. Cardeñoso-Payo, J. Alonso-López, D. Martín-García, and A. Pedrosa-Calvo. Measuring the performance impact of SP-restricted programming in shared-memory machines. In J.M.L.M. Palma, J. Dongarra, and V. Hernández, editors, *Vector and Parallel Processing - VECPAR 2000*, number 1981 in LNCS, pages 128–728, Porto (Portugal), 2001. Springer.

[87] J.A. González, C. León, F. Piccoli, M. Printista, J.L. Roda, C. Rodriguez, and F. Sande. Towards standard nested parallelism. In J. Dongarra et al., editor, *Recent Advances in PVM and MPI (EurPVM/MPI 2000)*, volume 1908 of *LNCS*, pages 96–103, Balatonfüred, Hungary, Sep 2000. Springer.

[88] S. Gorlatch. Toward formally-based design of message passing programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, Mar 2000.

[89] S. Gorlatch. Send-Recv considered harmful? myths and truths about parallel programming. In V. Malyshkin, editor, *PaCT'2001*, volume 2127 of *LNCS*, pages 243–257. Springer-Verlag, 2001.

[90] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *Proc. IPPS/SPDP'99*, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.

[91] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, and T. Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Tansactions on Computers*, 48(7):670–689, Jul 1999.

[92] M.W. Goudreau, K. Lang, S.B. Rao, and T. Tsantilas. The green BSP library. Technical Report TR-95-11, University of Central Florida, Orlando, 1995.

[93] J. Gross and J. Yellen. *Graph Theory and its Applications*. The CRC Press Series on Discrete Mathematics and its Applications. CRC Press, 1999.

[94] T. Gross, D.R. O'Hallaron, and J. Subhlok. Task parallelism in a high-performance Fortan framework. *IEEE Parallel & Distributed Technology*, pages 16–26, Fall 1994.

[95] E.J. Gumbel. *Statistical Theory of Extreme Values (Main Results)*, chapter 6, pages 56–93. Wiley Publications in Statistics. John Wiley & Sons, 1962.

[96] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. on Mathematical Software*, 28(3):301–324, Sep 2002.

[97] T.J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, Jun 1994.

[98] X. He and Y. Yesha. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 75:15–38, 1987.

[99] M.T. Heath. Parallel direct methods for sparse linear systems. In D.E. Keyes et al., editor, *Parallel Numerical Algorithms*, pages 55–90. Kluwer Academic Publishers, 1997.

[100] J.M.D. Hill, S.R. Donaldson, and D.B. Skillicorn. Portability of performance with the BSPlib communications library. In *Programming Models for Massively Parallel Computers, (MPPM'97)*, London (UK), Nov 1997. IEEE Computer Society Press.

[101] J.M.D. Hill, W. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, Nov 1998.

[102] J.M.D. Hill and D.B. Skillicorn. Practical barrier synchronisation. Tech. Rep. PRG-TR-16-96, Oxford Univ. Computing Laboratory, 1996.

[103] W.D. Hillis and L.W. Tucker. The cm-5 connection machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993.

[104] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD ditributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug 1992.

[105] C-W. Ho, S-Y. Hsieh, and G-H. Chen. Parallel decomposition of generalized series-parallel graphs. *Journal of Information Science and Engineering*, 15:407–417, 1999.

[106] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct 1974.

[107] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 28(1):666–677, Aug 1978.

[108] HPF: The High Performance Fortran Home Page. WWW. On http://www.crpc.rice.edu/HPFF/.

[109] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A parallel computational model for synchronization analysis. In *Proc. PPOPP'01*, pages 133–142, Snowbird, Utah USA, Jun 2001. ACM.

[110] P.G. Joisha and P. Banerjee. PARADIGM(version 2.0): A new HPF compilation system. In *Proc. IPPS/SPDP'99*, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.

[111] J.E. Jones. Parallel multigrid methods. In D.E.K̃eyes et al., editor, *Parallel Numerical Algorithms*, pages 203–224. Kluwer Academic Publishers, 1997.

[112] B. Juurlink and I. Rieping. Performance relevant issues for parallel computation models. In *Proc. PDPTA'2001*, 2001.

[113] B.H.H. Juurlink and A.G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Proc. Euro-Par'96*, volume 1124 of *LNCS*. Springer, 1996.

[114] B.H.H. Juurlink and H.A.G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318, Aug 1998.

[115] V. Karamcheti and A.A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Proc. SC'98*, Orlando, FL, 1998. ACM.

[116] R.M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. ACM-SPAA'93*, pages 142–153, Velen, Germany, 1993. ACM.

[117] G. Karypis. METIS: family of multilevel partitioning algorithms. WWW, 2002. On http://www-users.cs.umn.edu/˜karypis/metis/.

[118] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proc. 14th Int. Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, 2000. Computer Society, IEEE.

[119] C.W. Kessler. NestStep: nested parallelism and virtual shared memory for the BSP model. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas (USA), Jun-Jul 1999.

[120] D. Kim and I. Yoon. Performance analysis and experiments of sorts on a parallel computer with parallel computation methods. In *Proc. 1997 Int. Conf. on Parallel and Distributed Systems (ICPADS'97)*. IEEE, 1997.

[121] J-S. Kim, S. Ha, and C.S. Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *Proc. of IPPS/SPDP'98*. Computer Society, IEEE, 1998.

[122] L. Lamport and N. Lynch. *Handbook of Theoretical Computer Science*, chapter 18: Distributed Computing: Models and Methods. Elsevier Science Publishers, 1990.

[123] H.X. Lin. A general approach for parallelizing the FEM software package DIANA. In *Proc. High Performance Computing Conference'94*, pages 229–236. National Supercomputing Research Center. National University of Singapur, 1994.

[124] H.X. Lin, A.J.C. van Gemund, J. Meijdam, and P. Nauta. Tgex: a tool for portable parallel and distributed execution of unstructured problems. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *LNCS*, pages 467–474, Berlin, 1996. Springer.

[125] K. Lodaya and P. Weil. A kleene iteration for parallelism. In V. Arvind and R. Ramanujam, editors, *Proc. FST & TCS'98*, volume 1530 of *LNCS*, pages 355–366. Springer, 1998.

[126] K. Lodaya and P. Weil. Series-parallel posets: Algebra, automata, and languages. In *Proc. STACS'98*, volume 1373 of *LNCS*, pages 555–565, Paris, France, 1998. Springer.

[127] K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property. *Theoretical Comp. Science*, 237:347–380, 2000.

[128] W. Löwe and W. Zimmermann. Upper time bounds for executing PRAM-Programs on the LogP-Machine. In *Proc. ICS'95*, pages 41–50, Barcelona, Spain, 1995. ACM.

[129] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. 28th Hawaii Int'l Conf. System Sciences*, volume 2, pages 61–70, 1995.

[130] A.D. Malony, V. Mertsiotakis, and A. Quick. Automatic scalability analysis of parallel programs based on modeling techniques. In G. Haring and G. Kotsis, editors, *Comp. Perf. Eval.: Modeling Techniques and Tools* (LNCS 794), pages 139–158, Berlin, May 1994. Springer-Verlag.

[131] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *ICS'99*, pages 294–301, Rhodes, Greece, 1999.

[132] B.L. Massingill and K.M. Chandy. Parallel program archetypes. In *Proc. IPPS/SPDP'99*, San Juan, Puerto Rico, Apr 1999. Computer Society, IEEE.

[133] W.F. McColl. General purpose parallel computing. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, volume 4 of *Series on Parallel Computation*, chapter 13. Cambridge University Press, 1993.

[134] W.F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995.

[135] W.F. McColl. Universal computing. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. Euro-Par'96 Parallel Processing*, volume 1123 of *LNCS*, pages 25–26. Springer-Verlag, 1996.

[136] C. McCurdy and J. Mellor-Crummey. An evaluation of computing paradigms for N-body simulations on distributed memory architectures. In *Proc. PPoPP'99.* ACM, 1999.

[137] O. Melnikov, V. Sarvanov, R. Tyshkevich, V. Yemelichev, and I. Zverovich. *Exercises in Graph Theory*, volume 19 of *Kluwer Texts in the Mathematical Sciences.* Kluwer Academic Publishers, 1998.

[138] R. Miller. A library for bulk synchronous parallel programming. In *Proc. of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 100–108, 1993.

[139] C.A. Moritz and M.I. Frank. LoGPC: Modeling network contention in message-passing programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, Apr 2001.

[140] Message Passing Interface Forum. WWW. On http://www.mpi-forum.org/.

[141] A. Munier and C. Hanen. Using duplication for scheduling unitary tasks on m processors with unit communication delays. Technical Report LITP 95/47, Laboratoire Informatique Théorique et Programmation, Institut Blaise Pascal, Université Pierre et Marie Curie, 4, place jussieu, 75252 Paris cedex 05, 1995.

[142] G.J. Narlikar and G.E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems*, 21(1):138–173, Jan 1999.

[143] V. Naumann. Measuring the distance to series-parallelity by path expressions. In E.W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science* (LNCS), pages 269–281, Berlin, Jun 1994. Springer-Verlag.

[144] D.S. Nikopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in OpenMP? In *Proc. Supercomputing 2000*, page 47. ACM, 2000.

[145] L.S. Nyland, J.F. Prins, A. Goldberg, and P.H. Mills. A design methodology for data-parallel applications. *Trans. on Soft. Eng.*, 26(4):293–314, Apr 2000.

[146] National Institute of Standards and Technology (NIST). Matrix Market. WWW, 2002. On http://math.nist.gov/MatrixMarket/.

[147] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Proc. SC'99*, Portland, OR, 1999. ACM.

[148] A. Omicini. On the semantics of tuple-based coordination models. In *Proc. SAC'99*, pages 175–182, San Antonio, Texas (US), 1999. ACM.

[149] OpenMP organization.
WWW. On http://www.openmp.org.

[150] S. Orlando, P. Palmerini, and R. Perego. Coordinating HPF programs to mix task and data parallelism. In *Proc. SAC'00*, pages 240–247, Como, Italy, Mar 2000. ACM.

[151] G. Otero and R. Ferri. The Beowulf evolution. *Linux Journal*, 2002(100):5, 2002.

[152] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1998.

[153] R.B. Pelz. Parallel FFTs. In D.E.K̃eyes et al., editor, *Parallel Numerical Algorithms*, pages 245–266. Kluwer Academic Publishers, 1997.

[154] A. Pothen. *Parallel Numerical Algorithms*, chapter Graph Partitioning Algorithms with Applications to Scientific Computing, pages 323–368. Kluwer Academic Publishers, 1997.

[155] PVM: Parallel virtual machine (Home Page). WWW. On http://www.epm.ornl.gov/pvm/pvm_home.html.

[156] M.J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1993.

[157] A. Radulescu and A.J.C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory machines. In *Proc. 1999 ACM Int'l Conf. on Supercomputing*, pages 68–75, Rhodes, Jun 1999. ACM.

[158] A. Radulescu and A.J.C. van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):648–658, Jun 2002.

[159] A. Radulescu, A.J.C. van Gemund, and H-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. Int'l Parallel Processing Symp.*, pages 525–530, Puerto Rico, Apr 1999. IEEE.

[160] V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design. Technical Report TR98-22, CS dept., Univ. of Texas, Austin, Nov 1998.

[161] V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design. In *Proc. ACM-SIAM SODA'99*, pages 957–958, 1999.

[162] D.A. Reed, L.M. Adams, and M.L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, Jul 1987.

[163] J.L. Roda, F. Sande, C. León, J.A. González, and C. Rodríguez. The collective computing model. In *7th Euromicro Workshop on Parallel and Distributed Processing*, Funchal, Portugal, Feb 1999. IEEE.

[164] R.A. Sahner and K.S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Trans. on Software Eng.*, 13(10):1105–1114, Oct 1987.

[165] V. Sassone, M. Nielsen, and G. Winskel. A classification of models for concurrency (extended abstract). In Eike Best, editor, *CONCUR'93: 4th Int. Conf. on Concurrency Theory*, volume 715 of *LNCS*, pages 82–96, Hildesheim, Germany, Aug 1993. Springer-Verlag.

[166] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1–2):297–348, 1996.

[167] K. Schloegel, G. Karypis, and V. Kumar. *CRPC Parallel Computing Handbook*, chapter Graph Partitioning for High Performance Scientific Simulations. Morgan Kaufmann, 2000.

[168] L.A.M. Schoenmakers. A new algorithm for the recognition of series parallel graphs. Technical Report CS-R9504, CWI (Centrum voor Wiskunde en Informatica, 1995.

[169] H.J. Siegel. Panel: Top 10 most influential parallel and distributed processing concepts in the last millennium (M. Chandy position statement). In *Proc. 14th Int. Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, 2000.

[170] D.B. Skillicorn. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[171] D.B. Skillicorn. Building BSP programs using the refinement calculus. Technical Report ISSN-0836-0227-94-400, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Oct 1996.

[172] D.B. Skillicorn. miniBSP: A BSP language and transformation system. Technical report, Dept. of Computing and Information Sciences, Queen's University, Kingston, Canada, Oct 1996.

[173] D.B. Skillicorn, M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Oxfor Univ. Computing Laboratory, Oxford (UK), Nov 1996.

[174] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, Jun 1998.

[175] B.J. Smith. How shall we program high performance computers? ENACTS, The HPC Technology Roadmap, Seminar presented at NSC in Linkping, WWW, Jun 2001. On http://www.nsc.liu.se/rd/enacts/Smith/smith.html.

[176] T. Sterling. The scientific workstation of the future may be a pile of PCs. *Communications of the ACM*, 39(9):11–12, 1996.

[177] T.L. Sterling, J. Salmon, D.J. Becker, and D.F. Savarese. *How to Build a Beowulf.* Scientific and Engineering Computation. The MIT Press, 1999.

[178] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec 1990.

[179] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM*, 29(3):623–641, 1982.

[180] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of OpenMP applications with nested parallelism. In S. Dwarkadas, editor, *Proc. 5th Int. Workshop LCR'2000 (Selected Papers)*, number 1915 in LNCS. Springer, May 2000.

[181] T. Tobita and H. Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. In *ICS'99 Workshop*, pages 71–77, Jun 1999.

[182] V.D. Tran, L. Hluchy, and G.T. Nguyen. Parallel program model for distributed systems. In J. Dongarra et al., editor, *Recent Advances in PVM and MPI (EurPVM/MPI 2000)*, volume 1908 of *LNCS*, pages 96–103, Balatonfüred, Hungary, Sep 2000. Springer.

[183] A. Vaca-Díez. Tools and techniques to assess the loss of parallelism when imposing synchronization structure. Technical Report 1-68340-28(1999)02, TU Delft, Mar 1999.

[184] J. Valdés, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *SIAM Journal of Computing*, 11(2):298–313, May 1982.

[185] L.G. Valiant. A bridging model for parallel computation. *Comm.ACM*, 33(8):103–111, Aug 1990.

[186] H.A. van der Vorst. Linear system solvers: Sparse iterative methods. In D.E. Keyes et al., editor, *Parallel Numerical Algorithms*, pages 91–117. Kluwer Academic Publishers, 1997.

[187] R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. of PPoPP'01*, pages 34–43, Snowbird, Utah, USA, Jun 2001. ACM.

[188] S.A. Ward and S.H. Halstead. A syntactic theory of message passing. *Journal of the ACM*, 27(2):365–383, Apr 1980.

[189] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prantice Hall, 1999.

[190] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, chapter Models for Concurrency, pages 1–148. Oxford University Press, 1995.

[191] A. Zavanella and A. Milazzo. Predictability of bulk synchronous programs using MPI. In *Proc. 8th Euromicro Workshop on Par. and Dist. Processing*, Rhodes, Greece, Jan 2000. IEEE Computer.