Enhancing Performance on Combinatorial Optimization Algorithms

by

Francisco Cruz Mencía

A dissertation presented in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy in Computer Science

Advisors:
Dr. Antonio Espinosa
Dr. Juan Carlos Moure

Tutor:
Dr. Juan Carlos Moure

September 10, 2018

Universitat Autònoma de Barcelona
Departament d'Arquitectura de Computadors i Sistemes Operatius

To my grandfather
We will always remember you.

To my dear little Cloe
You are a blessing, I love you.

Abstract

Combinatorial Optimization is a specific type of mathematical optimization where variables' domain is discrete. This kind of optimization problems have an enormous applicability due to its ability to optimize over unitary and non-divisible objects. Beyond generic algorithms, the research community is very active proposing algorithms able to tackle specific combinatorial optimization problems.

The goal of this thesis is to investigate how to widen the applicability of Combinatorial Optimization algorithms that exploit the structure of the problems to solve. We do so from a computer's hardware perspective, pursuing the full exploitation of computational resources offered by today's hardware.

For the sake of generality we work on three different problems. First we tackle the Coalition Structure Generation Problem (CSGP). We find that the state-of-the-art algorithm is IDP. We propose an optimized and parallel algorithm able to solve the CSGP. We reach this by defining a novel method to perform the most critical operation –Splitting– as well as by defining a novel method to divide the the algorithm's operation in threads.

Then, we study the Winner Determination Problem (WDP) for Combinatorial Auctions (CA), which is very related to the CSGP. We find that state-of-the-art solvers' scalability is limited. More specifically we show how to improve results solving LP relaxations for the WDP in Large Scale Combinatorial Auctions by applying the AD^3 algorithm. Then we contribute with an optimized version of AD^3 which is also able to run in a shared-memory parallel scenario.

Finally we study the application of AD^3 to solve the LP relaxations of a more CPU demanding problem: The Side-Chain Prediction (SCP). We present an optimized way to solve the most critical operation which is solving a Quadratic Problem for an Arbitrary Factor.

In all the cases we propose optimized algorithms that are scalable in parallel and that improve significantly the state-of-the-art. Three orders of magnitude speedup on IDP, one order of magnitude speedup in AD^3.

The ultimate purpose of this work is to demonstrate how a hardware-aware algorithmic design can lead to significant speedups. We show strategies that are exportable to similar Combinatorial Optimization algorithms. Such strategies will help the algorithm designer to achieve more efficiency in modern CPUs.

Resum

L'optimització combinatòria és un tipus específic d'optimització matemàtica on el domini de les variables és discret. Aquest tipus de problemes d'optimització tenen una gran aplicabilitat degut a la seva capacitat d'optimització sobre objectes unitaris i no divisibles. Més enllà dels algoritmes genèrics, la comunitat investigadora és molt activa proposant algorismes capaços d'abordar problemes d'optimització combinatòria per a problemes específics.

L'objectiu d'aquesta tesi és investigar com ampliar l'aplicabilitat d'algorismes d'optimització combinatoria que exploten l'estructura dels problemes a resoldre. Ho fem des de la perspectiva del maquinari d'una computadora, perseguint l'explotació total dels recursos computacionals que ofereix el maquinari actual.

Per assolir generalitat treballem amb tres problemes diferents. Primer abordem el problema de generació d'estructures de la coalició (CSGP). Trobem que l'algorisme d'última generació és IDP. Proposem un algoritme optimitzat i paral·lel capaç de resoldre el CSGP. Aconseguim això definint un nou mètode per dur a terme l'operació més crítica -Splitting-, així com definint un nou mètode per dividir l'operació de l'algoritme en els diferent subprocessos.

A continuació, estudiem el problema de determinació del guanyador (WDP) per a les subhastes combinades (CA). Trobem que l'escalabilitat dels solucionadors d'avantguarda és limitada. Més concretament, mostrem com millorar la resolució de resultats de relaxació LP per al WDP en subhastes combinables de gran escala mitjançant l'aplicació de l'algoritme AD^3. A continuació, contribuïm amb una versió optimitzada d'AD^3 que també es pot executar en un escenari paral·lel de memòria compartida.

Finalment, estudiem l'aplicació de AD^3 per resoldre les relaxacions LP d'un problema més exigent de la computacionalment: El problema de la predició de cadenes laterals (SCP). Presentem una manera optimitzada de resoldre l'operació més crítica, la resol·lució d'un problema quadràtic per a un factor arbitrari.

En tots els casos proposem algoritmes optimitzats que es poden escalar de forma paral·lela i que milloren notablement l'estat de la tècnica. Tres ordres de magnitud a IDP, i un ordre de magnitud a AD^3.

L'objectiu final d'aquest treball és demostrar com un disseny algorisme conscient de maquinari pot conduir a millores de rendiment significatives. Mostrem estratègies exportables a algorismes d'optimització combinatòria similars. Aquestes estratègies ajudaran al dissenyador d'algorismes a assolir més eficiència en les CPU modernes.

Resumen

La optimización combinatoria es un tipo específico de optimización donde el dominio de las variables es discreto. Este tipo de problemas de optimización tienen una gran aplicabilidad debido a su capacidad de optimización sobre objetos unitarios y no divisibles. Más allá de los algoritmos genéricos, la comunidad investigadora es muy activa proponiendo algoritmos capaces de abordar problemas de optimización combinatoria para problemas específicos.

El objetivo de esta tesis es investigar cómo ampliar la aplicabilidad de algoritmos de optimización combinatoria que explotan la estructura de los problemas a resolver. Lo hacemos desde la perspectiva del hardware de una computadora, persiguiendo la explotación total de los recursos computacionales que ofrece el hardware actual.

Para alcanzar generalidad trabajamos con tres problemas diferentes. Primero abordamos el problema de generación de estructuras de la coalición (CSGP). Encontramos que el algoritmo de última generación es IDP. Proponemos un algoritmo optimizado y paralelo capaz de resolver el CSGP. Conseguimos esto definiendo un nuevo método para llevar a cabo la operación más crítica -Splitting-, así como definiendo un nuevo método para dividir la operación del algoritmo en los diferente subprocesos.

A continuación, estudiamos el problema de determinación del ganador (WDP) para las subastas combinatorias (CA). Encontramos que la escalabilidad de los solucionadores de vanguardia es limitada. Más concretamente, mostramos cómo mejorar la resolución de resultados de relajación LP para el WDP en subastas combinatorias de gran escala mediante la aplicación del algoritmo AD^3. A continuación, contribuimos con una versión optimizada de AD^3 que también se puede ejecutar en un escenario paralelo de memoria compartida.

Finalmente, estudiamos la aplicación de AD^3 para resolver las relajaciones LP de un problema más exigente de la computacionalmente: El problema de la predición de cadenas laterales (SCP). Presentamos una manera optimizada de resolver la operación más crítica, la resoución de un problema cuadrático para un factor arbitrario.

En todos los casos proponemos algoritmos optimizados que se pueden escalar de forma paralela y que mejoran notablemente el estado de la técnica. Tres órdenes de magnitud en IDP, y un orden de magnitud en AD^3.

El objetivo final de este trabajo es demostrar como un diseño algoritmo consciente de hardware puede conducir a mejoras de rendimiento significativas. Mostramos estrategias exportables a algoritmos de optimización combinatoria similares. Estas estrategias ayudarán al diseñador de algoritmos lograr más eficiencia en las CPU modernas.

Acknowledgements

First and foremost, I want to thank my advisors Juan Carlos Moure and Toni Espinosa, because this PhD would not have been possible without their support. Thanks for your patience, your always valuable advice, for pointing me in the right direction when I was lost, for sharing a beer with me in the right moment, for never giving up on me and for your understanding in all circumstances .

Also, I would like to thank Jesus Cerquides, who sat with me many times, helped me understand many concepts and offered me his help as well as his guidance during this long journey. His suggestions and ideas have always been useful and brilliant.

I also want to express my endless gratitude to Juan Antonio Rodriguez-Aguilar, JAR. He helped me to greatly increase the quality of my work; provided me with advice, motivation and feedback. He worked hard and behaved in a way that can only be expected from a best friend or a brother. Big thanks to JAR for everything!

I cannot forget the constant support of Pedro Meseguer. Not only Pedro has always been opent to talk, but he also knocked on my door to offer his help and discuss anything I needed to. Every time I asked him for help, he was there and provided me with valuable comments.

I am very lucky to work at the Artificial Intelligence Research Institute (IIIA-CSIC), where I have the opportunity to exchange views with students and researchers. Many people influenced me during my research, and it's impossible to mention all of them since my success is due in part to every student, every staff member, researcher. However I would like to highlight a few names that have been of fundamental importance to me. Firstly, some of the PhD students (now PhD holders) who motivated and gave me their advice: Andrea Giovanucci, Manu Atencia, Marc Pujol, Xavi Ferrer. Thank you, guys,you are a source of motivation! Then I thank the institutional support I received, thanks to Carles Sierra and Ramon López de Mántaras who always encouraged me to go ahead. Also, to Montse Calderón who has been a friend and a confidant and always tried to make everything easy. I would also like to thank my daily workmates, such as Anna Enciso, Ana Beltran, Montse Santmarti and Carmen, who made my day-to-day easier. But above all, I want to thank Bea Hernandez, who has been always covered for me in my daily duties and has selflessly helped me in everything she could with enormous patience. Bea, thank you very much for everything.

Morevoer, I have been involved in the Computer Architecture and Operating Systems Department, where I found some great people. I want to thank Alejandro for our conutless talks and conversations, it has been very rewarding to work close to him. Thanks also to Ania and Porfidio who always had good words for me and helped me with everything I needed with the utmost kindness.

I also have words of gratitude for my friends, who understood every time I was not able to join them, but they were still there, hoping that I managed to finish my PhD. Thank you for your support Xavi, Ariadna, Isaac, Adela, Marta, Anita, Izkor. Thanks also to my dear and crazy *vermut* friends for identical reasons: Mario, Núria, Berta, Jose, Nati and Joan. Soon we will party!

Furthermore, I want to thank my wonderful family who unconditionally supported me during this journey, thanks to my mother Pepi, my aunt Mercedes and my siblings Daniel and Mireia, as well as their respective partners Javi, Jesús, Ester and Manu. You are all extraordinary, I am very proud to be part of our family. I would also like to thank my new family: Paqui, Ramon, Alvaro, Judith, Elba and Pepe, who understood from the beginning that I was always busy and supported me and helped me enormously.

Finally, I will thank my wonderful wife, Paqui, for her love and constant support, for all the late nights and early mornings. Thank you, my love, for being there all the time.

– Francisco Cruz

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Combinatorial Optimization is a specific type of mathematical optimization where variables' domain is discrete. This kind of optimization problems have an enormous applicability due to its ability to optimize over unitary and non-divisible objects. Hence, they help to model many real world problems involving resources like people, tools, time-slots, etc. [Paschos, 2013]. However, *Combinatorial Optimization* problems are typically hard to solve [Ausiello et al., 2012]. Their complexity is commonly $\mathcal{NP}$-hard and $\mathcal{NP}$-complete, hence being computationally very expensive, specially when many variables are involved.

The standard method used for solving *Combinatorial Optimization* problems is *Integer Programming* (IP). Modern IP solvers are a "swiss knife" for *Combinatorial Optimization* problems. The most powerful solvers are commercial products such as IBM ILOG CPLEX Optimization Studio [Ibm, 2011] and GUROBI [Optimization, 2017]. Although there are also open source alternatives such as SCIP [Achterberg, 2009] or GLPK [Makhorin, 2000], commercial solvers are one step ahead of Open Source alternatives in terms of performance.

Beyond generic algorithms, the research community is very active proposing algorithms able to tackle specific *Combinatorial Optimization* problems. Examples of these approaches are classical problems like the traveling salesman problem, set bin packing or knapsack problems. These and other problems are discussed in [Bernhard and Vygen, 2008]. It is well-known that specific algorithms are able to exploit the structure of targeted problems and thanks to that, they are able to outperform generic algorithms.

The goal of this thesis is to investigate how to widen the applicability of *Combinatorial Optimization* algorithms that exploit the structure of the problems to solve. We do so from a computer's hardware perspective, pursuing the full exploitation of computational resources offered by today's hardware.

In this chapter we first introduce our motivation in section 1.1, then in section 1.2 we present a brief description of the problems that we target. Next, in section 1.3 we expose the challenges pursued in this work, and thereafter in section 1.4 we present our contributions. Finally, in section 1.5 we outline the structure of this dissertation.

1.1 Motivation

During the last decades, computers' computational capacities have enormously increased. This is mainly due to both the increment of processor clock frequencies, and the ability to integrate more transistors in a single chip. With more transistors, chip designers have exploited the possibility of implementing new strategies to increase processors' performance. Examples of these strategies are vectorized instructions, out-of-order execution or branch prediction.

In state-of-the-art research in artificial intelligence, and more specifically in *Combinatorial Optimization*, we find algorithmic proposals and implementations that exhibit a low exploitation of computational resources. This is particularly paradoxical when considering computationally-intensive algorithms whose running times are critical. The observed infra-utilization of resources is mainly due to a gap between programmer and hardware. We distinguish four factors conditioning this relationship:

- **Complex Hardware.** Hardware research has considerably advanced, giving rise to much more powerful yet complex computational architectures. Modern processors have been designed considering hardware techniques that improve the average program execution performance. For instance: nowadays we count on computer systems with a hierarchical heterogeneous memory system; systems with more than one processing unit; processors able to multiplex threads; also able to run instructions that run in parallel with the same core; among many other features. We argue that *achieving the maximum performance of a computer requires a precise understanding of its architecture.*

- **Increase of abstraction in programming languages.** At the same time, programming languages have also changed. Nowadays there are high-level languages that give programmers a higher degree of abstraction, thus facilitating rapid development. However, such benefits in terms of ease of development come at the price of losing control over the code that a machine actually runs.

- **Difficulty to grasp the parallel paradigm.** Parallel computers are a reality. The trend in computer's architecture is to include more computational units inside of the same chip [Sutter, 2005]. However, parallel programming follows a different paradigm than sequential programming. Moreover, current programming languages offer a very narrow support for parallel algorithms, hence hindering their exploitation by developers.

- **Outdated conceptual model.** Finally, algorithm designers use a conceptual machine model that has not changed for decades. However, as mentioned above, hardware has significantly changed in terms of complexity and capability of hosting parallelism. Common knowledge traditionally shared by developers time ago (e.g. "all instructions take approximately the same time" or "all memory accesses have the same cost") are no longer true. Most often development occurs in an architecturally-neutral manner, namely disregarding the full potential offered by the new conceptual model offered by modern computer architectures.

Against this background, in this thesis we propose to learn the architecturally-dependent algorithmic strategies that increase the performance of a particular family of algorithms: *Combinatorial Optimization* algorithms that exploit the structure of the problems to solve. Such strategies are to be founded on a better exploitation of hardware resources. Examples of such architecturally-dependent strategies are: data-oriented mechanisms for reducing memory footprint, the design of data structures that can be more efficiently accessed, or the design of a thread scheduling approach that maximizes the benefit of parallel resources.

1.2 Selected Problems

In this section we describe the *Combinatorial Optimization* problems that are the focus of our study throughout this thesis. For the sake of generalization of results, we analyze representative problems that vary in:

- size, from small problems (with low number of variables) that can be solved optimally to large problems (with a large number of variables) that can be only approximated;
- type of constraints composing a problem, being these either linear or declarative (able to implement first-order logic functions); and
- objective function, where the target of the algorithm will be to find either exact solutions or approximate ones.

Moreover, since the *Combinatorial Optimization* has applications in many fields, we choose *Combinatorial Optimization* algorithms from different domains (coalition formation, auctions and biology) with the aim of reaching results with a wide range of applicability.

In what follows, we characterize the structural features of the different problems selected in this thesis:

- **Coalition Structure Generation.** We first target a scenario where the *Combinatorial Optimization* problem has a low number of variables but a large density of restrictions. To this end, we work on those problems where there exists a constraint for every possible combination of variables.

  This problem has few decision variables subject to a large number of linear constraints.

- **Winner Determination Problem in Combinatorial Auctions.** We also study the case where the number of variables is larger than in the Coalition Structure Generation and the restrictions are structured according to known distributions. This distributions can be artificial or represent actual-world problems such as logistics or planning. We do such study facing the *Winner Determination Problem (WDP) in Combinatorial Auctions (CA).*

This problem has a large number of decision variables subject to a set of linear constraints.

- **Side-Chain Prediction.** Finally we will look at an industrial case where the number of variables is large and the restrictions are more expressive, implementing non-linear functions that can be characterized by declarative constraints: The *Side-Chain Prediction* (SCP) problem.

  This problem has a large number of decision variables subject to a set of declarative constraints.

Table 1.1 presents a summary of the qualitative features of the problems studied in this thesis. Table 1.1 shows that our selected problems allow us to cover a wide spectrum of problem structures in terms of size, type of constraints, and optimality.

Problem	Scale	Constraints	Objective
CSG	small	linear	optimal
WDP	medium-large	linear	approximate
SCP	large	declarative	approximate

Table 1.1: Qualitative features of the selected problems studied in this thesis

Coalition Structure Generation

In the multi-agent systems area [Wooldridge, 2009], coalition formation is one of the central types of collaboration. It involves the creation of typically disjoint groups of autonomous agents that collaborate in order to satisfy their individual or collective goals. One of the major research challenges in the field is the search for an effective set of coalitions that maximizes the global utility [Shehory and Kraus, 1998] of the agents.

According to [Sandholm and Lesser, 1997], the coalition formation process is divided into three activities.

- Coalition Structure Generation (CSG);
- solving the optimization problem of each coalition; and

- dividing the value of the generated solution among agents.

We focus on the first of these activities, namely Coalition Structure Generation, which is a particular type of *Combinatorial Optimization* problem. The Coalition Structure Generation problem considers a set of agents $A = \{a_1, a_2, \ldots, a_n\}$ such that every possible association between them has some revenue (or score). The objective is to find the set of associations (coalitions) with the largest aggregated revenue.

This is an interesting topic faced from different perspectives as summarized in a recent survey [Rahwan et al., 2015]. Our goal is to face problems where all the possible association between agents are defined and coalitional values do not follow any particular distribution. This is the hardest problem type in CSG, as shown by the low performance achieved by the state of the art. Indeed, IDP [Rahwan and Jennings, 2008b], a state-of-the dynamic programming algorithm takes around 2 months of running time to obtain a solution for a problem with only 27 agents, as reported in [Rahwan et al., 2009].

Winner Determination for Combinatorial Auctions

A combinatorial auction [Rothkopf et al., 1998] is a type of market mechanism where there is only one seller (auctioneer) and many participants (bidders). The auctioneer has a finite set of different objects to sell. The participants can place bids for single items or combinations of items. A combinatorial bid for a set of items I at a given price p indicates that the bidder is willing to pay p when provided with all the items in I. Thus, a bidder bids for all the items in I as a whole. There is a bidding phase during which all participants place their combinatorial bids. The winner determination problem is that of determining the combination of bids that maximizes the auctioneer's revenue.

In [Leyton-Brown et al., 2009], the authors observe that small and medium-sized problems (with dozens and hundreds of variables) are well addressed by Mixed Integer Programming (MIP) solvers. They also empirically determine the hardness of different known distributions. However, although there are application domains that claim to be large-scale, namely involving thousands and even millions of bids, current results indicate that the scale of winner determination problems for CAs that can be optimally solved is small [Leyton-Brown et al., 2009, Ramchurn et al., 2009]. For instance, CPLEX (a state-of-the-art commercial solver) requires a median of around 3 hours to solve the integer linear program encoding the Winner Determination Problem (WDP) of a hard distribution instance of a CA with only 1000 bids and 256 goods. This fact seriously hinders the practical applicability of current solvers to large-scale CAs.

Side-Chain Prediction

Proteins are macromolecules that play a fundamental role in every living being. In general terms, a protein is a large chain of 20 possible amino-acids sharing the same structure: an amine ($-NH_2$) and a carbooxyl ($-COOH$) functional groups along with a radical (or side-chain) specific to each amino-acid. A side-chain can take different configurations, each one being defined by a collection of angles. Side-chains have some preferred or stable configuration.

Amino-acids can catenate themselves in a linear structure. The angle formed between two amino-acids on each link will mostly depend of the side-chain configurations of the two amino-acids. However, not only the neighbor amino-acids occur in a particular link. Other sections of the protein that are close in the space also have their influence in the tridimensional composition. All in all, the sequence of amino-acids, together with their side-chain configuration determine both the structure of the protein and its properties.

An important research topic in biology is protein design [Krivov et al., 2009]. One approach is to first develop a 3D model of a protein that satisfies a particular function to subsequently compute the amino-acid sequence of the protein given its tridimensional structure. This is known as Side-Chain Prediction.

Side-Chain Prediction is a complex problem. It requires the model of the physics happening at molecular level, the computation of the energy needed by every possible side-chain using Van der Waals forces, the use of a library of stable configurations and the execution of a combinatorial optimization algorithm. The *Combinatorial Optimization* phase is the most time-consuming. Its input is an optimization problem with a variable for every radical position. Every variable has associated a list of candidate side-chains, each one with its energetic cost. The objective is to minimize the amount of molecular energy needed to conform a protein. The expected result is the assignment of side-chains to variables so that the overall energetic cost is minimized.

Key features of the *Combinatorial Optimization* problem are:

- Length. As many variables as amino-acids counts on. Therefore, the problem size is inherently big since proteins are built by chaining hundreds or thousands of amino-acids.
- Fixed arity. By the modelization method constraints are always pairwise.
- Declarative constraints. Constraints are *declarative*, hence they can't be approached using a linear optimization method.

1.3 Challenges

In this dissertation, we aim at improving the performance of representative state-of-the-art *Combinatorial Optimization* algorithms through the better exploitation of hardware resources. Our objective is to obtain a better performance thanks to an architecturally-aware algorithmic design.

After visiting the existing literature on state-of-the-art algorithms for *Combinatorial Optimization*, we observe that hardware considerations such as parallelism or memory organization are barely considered. We argue that this is due to the already discussed programmer-hardware gap in section 1.1. Such infra-utilization of hardware resources largely motivates our research.

The research challenges that we tackle in this thesis are framed in the context of maximizing hardware resources. We face such goal from a computers' architecture perspective by proposing mechanisms for enhancing algorithmic performance in a modern processor. Thus, in what follows we pose research challenges whose solutions will lead to improving algorithmic performance in a modern processor by:

1. reducing the amount of instructions to process,
2. increasing memory system throughput, and
3. increasing parallelism.

Henceforth, we shall refer to the above-mentioned architecturally-aware goals as *performance enhancement goals.*

In what follows, we identify the research challenges posed to achieve our performance enhancement goals by each of the selected problems described in section 1.2.

1.3.1 Coalition Structure Generation

The Coalition Structure Generation problem can be solved using a Dynamic Programming algorithm known as DP [Yun Yeh, 1986]. This algorithm was revised and updated years later, resulting into an improved version of DP, the so-called Improved DP (or IDP) [Rahwan and Jennings, 2008b]. IDP overcomes some of DP's drawbacks like the redundant evaluation of coalition structures.

Both DP and IDP algorithms run for a deterministic number of iterations, since they do not reduce work. Other algorithms like Branch-and-bound [Land and Doig, 1960] can skip computation during the search if they are able to prove that a path is not leading to a good solution. The computation of bounds depends of the data encoded in the problem. Hence, pruning techniques are highly dependent of the problem data distribution. DP and IDP worst-case complexity is the lowest of other known algorithms such as IP [Rahwan et al., 2007] or IDP-IP [Rahwan and Jennings, 2008a]. For these reasons DP and IDP are general-purpose CSG algorithms in the sense that they are well suited for problems whose distribution is unknown.

The first challenge of this thesis is to analyze the algorithmic features of DP and IDP that most affect their algorithmic performance. In this particular case this will amount to studying their: problem structures, memory access patterns, and most time-consuming operations.

Challenge 1 *Identify the algorithmic features of DP and IDP that most impact their performance in a modern processor*

The contributions of the previous challenge will allow us to determine the most time-consuming operations together with their memory access patterns. Furthermore, since neither DP nor IDP benefit from the parallel paradigm, leveraging on the results provided to the former challenge naturally leads to the next one:

Challenge 2 *How to enhance DP/IDP to benefit from the parallel paradigm and better exploit hardware resources?*

1.3.2 Combinatorial Auctions

Due to the the high complexity of *Combinatorial Optimization* problems, the limits of exact and exhaustive methods shown by coalition structure generation algorithms is early reached. If so, approximate algorithms stand as an alternative to optimal algorithms.

Branch-and-bound algorithms that employ good pruning strategies may reduce their solving time significantly. The standard method for solving *Combinatorial Optimization* problems with linear objective functions is Mixed Integer Linear Programming (MILP), which prunes subtrees of the search space based on linear programming relaxations to estimate bounds. In this scenario, the efficiency of the linear programming solver will condition the efficiency of the overall Branch-and-bound algorithm.

As stated in the literature, large scale *Combinatorial Optimization* problems (e.g Stereo Vision, Side-Chain Prediction, Protein Design [Yanover et al., 2006]) go beyond the capabilities of state-of-the-art solvers, according to [Yanover et al., 2006] they face problems that are so big that cannot to be solved by standard solvers: "*[..] We found that using powerful, off-the-shelf LP solvers, these problems cannot be solved using standard desktop hardware.*". The machine learning literature has recently contributed with a wealth of algorithms to obtain approximate solutions for *Combinatorial Optimization* problems, e.g [Gabay and Mercier, 1976, Ihler et al., 2005, Wainwright et al., 2005, Martins et al., 2014].

Despite the existence of such powerful approximate algorithms, to the best of our knowledge there is no approximate algorithm in the literature to solve large-scale winner determination problems for combinatorial auctions. This leads to our next research challenge:

Challenge 3 *How to solve a relaxation of a combinatorial auction winner determination problem?*

Fulfilling this challenge will require: (a) to find a state-of-the-art, representative approximate algorithm that is prone to parallelism; and (b) to show that such algorithm can indeed cope with large-scale combinatorial auction WDPs. Notice that solving a relaxation of the winner determination problem does not provide a feasible solution to the problem. However, this is a stepping stone towards building such feasible solution.

After finding a suitable algorithm satisfying our requirements, our next challenge will be:

Challenge 4 *How to design a shared-memory parallel algorithm to solve relaxations of the Winner Determination Problem?*

1.3.3 Side-Chain Prediction

In Side-Chain Prediction, the most time-consuming phase is the resolution of a *Combinatorial Optimization* problem. This is caused again by the structure of the problem. The existence of pairwise constraints of declarative nature has a major influence on the hardness of the *Combinatorial Optimization* problem.

In [Yanover et al., 2006, Sontag et al., 2012], convex optimization methods are used to efficiently solve the relaxation to the optimization problem posed by the Side-Chain Prediction.

Although the contributions to challenge 4 are expected to produce a shared-memory parallel algorithm to solve large-scale winner determination problems, we cannot benefit from it to solve the Side-Chain Prediction problem. This is because the structure of Side-Chain Prediction problems is different to that of winner determination problems. On the one hand, Side-Chain Prediction constraints are always pairwise. On the other hand, as mentioned in section 1.2, constraints in a side-prediction problem are declarative, and hence different to the linear constraints in winner determination problems. These two observations call for different algorithmic solutions to the side-prediction problem.

Both challenges 3 and 5 can be reached with the choice of an algorithm that is able to cope with the types of constraints handled by Combinatorial Auctions Winner Determination Problem and Side-Chain Prediction. In addition, as shown in [Martins, 2012b], AD^3 is able to solve LP relaxations for the Side-Chain Prediction. Nonetheless as already stated AD^3 is not optimized and it is not able to run in parallel. Our next challenge descent directly from this fact.

Challenge 5 *How to design a shared-memory parallel algorithm to solve relaxations of Side-Chain Prediction problem?*

The achievement of this challenge can be done by extending the contributions to challenge 4 to also cope with declarative constraints.

1.4 Contributions

Next, we present the contributions in this thesis in relation to the challenges posed in section 1.3. Likewise in section 1.3, we present our contributions around the selected problems described in section 1.2.

1.4.1 Coalition Structure Generation

In this section we present our contributions in reply to challenges 1 and 2. Recall that challenge 1 above focused on finding the algorithmic features of DP and IDP that most impact their performances on a modern processor. To tackle this challenge, we measured and characterized the execution of both DP and IDP. From our extensive analysis, we learned that the main factors affecting DP and IDP performances are:

- **The efficiency in splittings enumeration**. When following a dynamic programming strategy to solve a *Combinatorial Optimization* problem, the problem is divided in subproblems, which are solved by also dividing them into smaller subproblems. This process is repeated until the subproblems to be solved are trivial and can be easily solved by a simple evaluation. Then, the chosen dynamic programming method aggregates results from solving subproblems to build up a global solution. In the case of DP and IDP, the division of work is carried by the operation that enumerates splittings, which consists in enumerating all the partitions of size 2 of a given set, as shown in Table 1.2. The Table 1.2 shows all the splittings (partitions of size 2) of a set of objetcs. Our performance analysis found that more than 99% of the processor running time is spent in computing the enumeration of splittings, having a dramatic impact on algorithmic performance. However, the literature disregards how to implement such operation efficiently.

Set of Objects	Splittings
$\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$	$\{\spadesuit\}, \{\heartsuit, \clubsuit, \diamondsuit\}$
	$\{\heartsuit\}, \{\spadesuit, \clubsuit, \diamondsuit\}$
	$\{\clubsuit\}, \{\heartsuit, \spadesuit, \diamondsuit\}$
	$\{\diamondsuit\}, \{\heartsuit, \clubsuit, \spadesuit\}$
	$\{\spadesuit, \heartsuit\}, \{\clubsuit, \diamondsuit\}$
	$\{\spadesuit, \clubsuit\}, \{\heartsuit, \diamondsuit\}$
	$\{\spadesuit, \diamondsuit\}, \{\clubsuit, \heartsuit\}$

Table 1.2: Splittings enumeration example

- **The non-uniform memory access pattern**. Notice that in order to obtain a good memory system performance, consecutive memory access should request addresses that are near in memory, ideally consecutive [Guide, 2011]. In other words, memory system performance can be achieved by enforcing locality. Both DP and IDP algorithms enumerate splittings and evaluate the generated pairs within each splitting. In order to evaluate a pair, the computer has to perform two memory requests. If the two memory requests access addresses that are close in memory, the operation will be efficient. Otherwise, if the requests access distant memory positions, the operation will produce an inefficient memory access. The ideal situation will be to have only efficient memory accesses. However, since all the possible pairs are evaluated by DP and IDP, they unavoidably perform both efficient and inefficient splitting evaluations. Hence, there is no memory representation favoring a higher locality for DP/IDP.

- **The lack of parallelism**. In modern processors it is common to have more than one computational unit. However, DP and IDP use just one core and ignore the rest.

The findings of our analysis conclude that these three factors completely drive the performance of DP/IDP. However, the state of the art on coalition structure generation has not addressed any of them. We conclude that in order to increase DP/IDP performance, there is a need for defining strategies addressing the above-mentioned factors:

- how to efficiently perform the splitting enumeration operation,
- how to represent the problem in memory to minimize the effects of the non-local access patterns, and
- how to get profit from parallel architectures.

Our first contribution compiles the above-mentioned observations:

Contribution 1 *The three factors that limit the performance of both DP and IDP are: the inefficient splitting enumeration, the non-uniform memory access pattern, and their lack of parallelism.*

The second challenge inquires whether it is possible to obtain a hardware-efficient and parallel version of DP/IDP. We accomplish the second challenge by means of the following strategies: (i) a novel method for enumerating splittings, (ii) a compact representation of coalitions, and (iii) a novel method for parallelizing DP/IDP.

Although in the literature we find measures of IDP's running times, there is no reference implementation available to benchmark our contributions. And yet, in order to analyze our improvements we need a reference or baseline implementation to compare with. For that reason, we have implemented a sequential algorithm following the specifications of [Yun Yeh, 1986] for DP, and [Rahwan and Jennings, 2008b] for IDP. Henceforth, we shall refer to those algorithms as base-DP and base-IDP respectively, or simply as base algorithms if there is no risk of misunderstanding.

Next, we detail the above-mentioned strategies that target the performance goals defined in section 1.3:

- **Efficient enumeration of splittings.** We propose a novel method for enumerating splittings, the so-called Fast-split method. Thanks to the use of logic arithmetic, Fast-split is able to perform the splitting operation –the most critical operation– using only 12 machine-code instructions per splitting. The result is a fast mechanism able to boost the execution of DP/IDP. Fast-split yields very significant speedups: One order of magnitude with respect to our base-IDP version, and two orders of magnitude with respect to the times reported by the state-of-the-art IDP implementation [1].

We observe from this strategy that:

An optimal way to enumerate splitting fulfills the the performance enhancement goal of reducing the number of instructions.

- **Compact representation of coalitions.** We define a compact way of representing coalitions. In our proposal, we use a binary encoding. Thanks to the compactness provided by such encoding, we manage to significantly reduce the memory footprint. At the same time, our compact representation allow us to significantly increase the amount of information sent by data transfers between memory and CPU. This representation helps to cope with the non-uniform memory access pattern exhibited by DP/IDP.

  We observe from this strategy that:

  A compact memory representation satisfies the performance enhancement goal of increasing memory system throughput.

- **Parallelization.** We design the first parallel version of DP/IDP. Our contribution is twofold:

  - We define an even way of dividing the work among different threads with the objective of maximizing the workload per thread and minimizing communication and synchronization between them.

  - We endow each thread with an algorithm to find its assigned tasks, without requiring communication with other threads.

  Our parallel algorithm obtains a 5X speedup with respect to base-IDP on a six-core computer.

  We observe from this strategy that:

  Our thread level parallelization of DP/IDP accomplishes the performance enhancement goal of increasing parallelism.

Combining the above-mentioned strategies, we reach a 60X speedup with respect to our base implementation in a 6-core computer. Since IDP is reported to solve a 27-agent problem in around 2 months [Rahwan et al., 2007], our

[1] Our base-IDP is able to solve a problem of 27 agents in 2.9 days in an Intel Xeon E5645. As already stated, there is no state-of-the-art public implementation of IDP available, but authors report in [Rahwan et al., 2007] that IDP solves a problem of 27 agents in more than 2 months using an unspecified computer.

result is by large the fastest Dynamic Programming algorithm, able to solve the same problem in 1.2 hours. We can announce our contribution as:

Contribution 2 *We contribute with the first parallel implementation of DP/IDP, which employs an efficient method for performing enumerations and a compact data representation. Our approach reaches a speedup of 60X in a six-core computer with respect to our base implementation.*

The above-mentioned contributions have been reported in the following publications:

1. Cruz-Mencía, F., Cerquides, J., Espinosa, T., Moure, J. C., Ramchurn, S. D., and Rodriguez-Aguilar., J. A. (2013). Coalition structure generation problems: optimization and parallelization of the IDP algorithm. In *Proceedints of the 6th International Workshop on Optimisation in Multi-Agent Systems, OPTMAS 2013, workshop affiliated with AAMAS 2013.*.
2. Cruz-Mencía, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2013). Optimizing performance for coalition structure generation problems' IDP algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 706. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), July 22-25, las Vegas, Nevada.
3. Cruz-Mencia, F., Espinosa, A., Moure, J. C., Cerquides, J., Rodriguez-Aguilar, J. A., Svensson, K., and Ramchurn, S. D. (2017). Coalition structure generation problems: optimization and parallelization of the IDP algorithm in multicore systems. *Concurrency and Computation: Practice and Experience*, 29(5).

Furthermore, the source code of our parallel implementations of DP an IDP is publicly available at:

`https://github.com/CoalitionStructureGeneration/DPIDP`.

1.4.2 Winner Determination Problem in Combinatorial Auctions

We start focusing on challenge 3, which considered how to solve a relaxation of the WDP.

With the raise of big data and machine learning, the limits of applicability of classic linear programming techniques have been reached [Sheffi, 2004]. To cover such limitation, convex optimization algorithms are gaining significance. Recent studies, demonstrate that convex optimization algorithms are well suited to approximate LP relaxations for large problems very efficiently

[Boyd and Vandenberghe, 2004].

The Alternate Direction Dual Decomposition (AD^3) [Martins et al., 2015] is an ADMM-based [Gabay and Mercier, 1976], dual-decomposition, message-passing[2] algorithm. It is a state-of-the art convex optimization algorithm that has been successfully applied to solve LP relaxations in different application domains, such as frame semantic parsers or protein prediction [Martins, 2012b]. AD^3 has advantages over other convex optimization algorithms that can be summarized as follows:

- it exhibits fast convergence to quality solutions;
- it also counts with a defined library of factors that are able to perform different optimization functions very efficiently;
- it allows the decomposition of the overall problems into local subproblems (encoded thanks to the library of factors), which can be solved independently.

We observe that in the state-of-the-art WDP research there is no specific approach to cope with large-scale problems relaxations. We also observe that, up to our knowledge, there is no proposal for solving WDP relaxations using a convex optimization technique. Finally, we observe that AD^3 is prone to be parallelized.

Because of the above-mentioned observations, we choose AD^3 as the algorithm to develop a solver for WDP LP relaxations. Our contribution starts by proposing an encoding of the WDP into a Factor Graph [Loeliger, 2004] that can be solved by means of AD^3. Our encoding only employs the *atMostOne* factor, namely one of the already bundled factors in the AD^3 factor library.

We also study the scope of applicability of AD^3 as a method to solve relaxations for large-scale WDPs. To this end, we use the WDP distributions defined by [Leyton-Brown et al., 2009], and we compare the solving times of AD^3 with those obtained by CPLEX, an off-the-shelf commercial solver, when solving WDP LP relaxations. We empirically find that while problems of small and medium size[3] are well addressed by CPLEX, larger and harder problems are better suited for AD^3.

Therefore, our contribution in reply to challenge 3 can be summarized as follows:

Contribution 3 *We provide a general encoding of a WDP into a Factor Graph that can be employed by the AD^3 algorithm to solve LP relaxations of WDPs. We observe that AD^3 is the algorithm of choice to solve large-scale*

[2] In this case, message-passing refers to a term in the field of convex optimization and it is related to the order that the operations are performed according to a graphical modelization of the optimization problem. This is distinct to message-passing in computer architecture, where it stands for a parallelization paradigm where threads communicate themselves through send and receive primitives.

[3] We use the term simpler/harder to refer to those problems that are easy/hard to solve to [Leyton-Brown et al., 2009]. We also refer to moderate size for problems involving hundreds of variables and constraints. By large scale we refer to problems involving thousands of variables and constraints. The sizes and distributions that we consider in our study are discussed in detail in Chapter 4.

WDP relaxations.

Next, to tackle challenge 4, we address the design of a parallel and high performance AD^3 algorithm. Such parallel version is constrained to one type of factors, *atMostOne*, since these are the only factors required to solve WDP relaxations. To design our parallel version with the aim of satisfying the performance goals described en section 1.3, we follow the next strategies: address specificity, increase memory system performance, embrace shared-memory parallelization, and promote vectorization.

Next, we detail the implementation of such strategies.

- **Address specificity.** We reach specificity by considering only the *atMostOne* factor. For AD^3, every factor is a "black box" able to solve a local optimization problem. Our approach is to build a specific algorithm able to work only with *atMostOne* factors. This yields some advantages that lead to increase performance:
  - on the one hand, having the same factor in all the computation nodes of the Factor Graph favors homogeneity, which benefits task balancing, hence favors parallelization;
  - on the other hand, it also allows us to blend the computation happening at factors together with the rest of computation happening at AD^3.

  This strategy satisfies the performance goal concerned with reducing the number of executed instructions.

- **AoS to SoA.** At present, the publicly available implementation of AD^3 [Martins, 2012a] follows an array of structures (AoS) paradigm for memory organization. Instead, we propose an alternative memory organization based on a structure of arrays (SoA) as suggested in [Guide, 2011] to increase locality and promote sequential memory access. Thus, in our alternative way of organizing memory helps improve memory system throughput.

  This new memory organization achieves the performance enhancement goal of increasing memory throughput.

- **Embrace edge-centric parallelization.** We reach shared-memory parallelism by distributing AD^3 computations among different execution threads. In AD^3, computations occur at two levels: computation at factors, and computation at variables –factors and variables are the two types of nodes of a Factor Graph. Typically, the machine learning literature claims that algorithms running over Factor Graphs can be readily parallelized because each factor independently solves a local subproblem [Martins et al., 2014]. Thus, the parallelization approach suggested by the machine learning literature can be considered as *factor-centered*. Instead, we propose a parallelization paradigm where edges are promoted to first-class citizens, hence following an *edge-centric* paradigm, as suggested by [Roy et al., 2013]. In an *edge-centric* paradigm, the computation is performed from an edge perspective. According to this paradigm, computation is moved as much as possible to the edge, with the objective of promoting edge-only dependent computations. Then the algorithm iterates also through the collection of edges, regardless which factors are edges attached to. Such parallelization design offers some important advantages with respect to a *factor-centered* design in terms of performance, namely:

- *Simple memory organization.* Edges are simpler than factors.
  An edge always connects two nodes, while a factor may have a variable number of nodes it is attached to. Therefore, the memory representation of a set of edges can be performed with simpler data structures, for instance using vectors, which are very efficient: (i) vectors are convenient for sequential memory access, leading to a high memory system performance; (ii) vectors can be easily divided in even blocks, making them suitable for workload distribution in a parallel execution; and (iii) vectors also ease instruction vectorization (SIMD).
  On the contrary, factors are more complex and need the definition of complex structures like linked lists, which are versatile but inefficient from a performance engineering point of view.
- *Flexibility.* In a Factor Graph, the number of edges is typically larger than the number of factors. An *edge-centric* design offers more granularity than a *factor-centric* approach leading to a fine-grained parallelism. Notice that obtaining a higher granularity benefits the dynamic parallelization, since thread workload is adapted dynamically.

Both advantages above lead to an improvement of algorithmic performance. On the one hand, the simpler memory organization enforces memory locality. On the other hand, more flexibility helps to reach a better balance between parallel threads.

This strategy fulfills the performance enhancement goal of increasing parallelism and also increase the memory throughput.

- **Promote vectorization.** We reach vectorization by decomposing every calculation in elemental operations. Hence, elemental operations can be grouped together and carried out at once. This fact, with a more convenient memory organization, aids the compiler to reach a vectorized code able to perform several operations using one single instruction.

This optimization satisfies two performance enhancement goals: reducing the number of executed instructions and increasing parallelism.

By combining the above-mentioned performance enhancement strategies, we develop the first shared-memory parallel AD^3-based algorithm able to efficiently solve LP relaxations for the WDP. Our work can be outlined as:

Contribution 4 *We develop a novel shared-memory parallel version of AD^3 capable of solving WDP relaxations. Our approach reaches a speedup of 14.4X in a 6-core computer compared to the publicly available implementation of AD^3.*

The above-mentioned contributions have been reported in the following publications:

1. Cruz-Mencia, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2015b). Paving the way for large-scale combinatorial auctions. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1855–1856. International Foundation for Autonomous Agents and Multiagent Systems.

2. Cruz-Mencia, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2015a). Parallelisation and application of AD^3 as a method for solving large scale combinatorial auctions. In *International Conference on Coordination Languages and Models*, pages 153–168. Springer.

Furthermore, the source code of our parallel implementation of AD^3, together with datasets, is publicly available at:

https://github.com/parad3-wdp/ca

1.4.3 Side-Chain Prediction

In this section we present our contributions giving answer first for challenge 5.

AD^3 has already proposed to be used as a method to solve the *Maximum A Posteriori (MAP)* in the problem of Side-Chain Prediction. This proposal requires the encoding of the Side-Chain Prediction problem as a Factor Graph where all the factors are pairwise and encode local subproblems with the use of declarative constraint.

As in the WDP for CA, we show how to encode a Side-Chain Prediction as a Factor Graph to be solved by means of AD^3. This time our codification relies on the representation of every amino-acid as a multi-variable and every possible side-chain configuration as a binary choice associated the specific multi-variable.

Interactions between different amino-acids are modeled as pairwise forces and we use one factor for every pair of interacting amino-acids.

We measure the performance of AD^3 and with previous related works [Yanover et al., 2006, Sontag et al., 2012], being the fastest algorithm on this class. We also discover that the problems in [Yanover et al., 2006] dataset are solved very efficiently by AD^3 and they does not exhibit enough computational work to allow us to stress AD^3. For this reason, we propose an extension of the dataset thanks to the use of a modern proprietary tool able to model Side-Chain Prediction problems (SCWRL4 [Krivov et al., 2009]).

We show then how to design an efficient algorithm able to solve LP relaxations for the Side-Chain Prediction problem. We contribute by extending the shared-memory parallel AD^3 algorithm developed in challenge 4, with the support for declarative constraints. Hence widening its applicability to a wide scope of problems.

Since the parallelization happens in outer levels, the fulfillment of challenge 5 is accomplished through an performance analysis and optimization of the computation happening at subproblem level in only one thread. Then our parallel mechanism designed in challenge 4 will be able to solve every subproblem in the parallel resources.

Nonetheless, the *performance enhancement goals* remain intact. We increase the single thread performance following the next actions.

- **Move towards a less instruction scenario**. In our work, we analyze the process of resolution of an *Arbitrary* factor, which is the one implementing the declarative constraint. As a result of our analysis we find that we are able to encode an equivalent algorithm leading to a more efficient execution. The specific actions we do to get this results are.
  - *Make data structures simpler.* We do that reducing encapsulation so functions that manipulate data are specialized.
  - *Promoting vectorization.* We increase the instructions throughput by obtaining vectorized code.
  - *Making use of external libraries.* We use state-of-the art optimized linear algebra libraries to compute critical operations.

  This optimization satisfies two performance enhancement goals: reducing the number of executed instructions and increasing parallelism.

- **Redesign memory representation and management**. We redesign how the data is stored in memory, data structures are expressed in a way that we enforce sequential access and high locality. We also define per-thread scratch areas, where threads are able to store partial results. These scratch areas are reused by threads, having a exclusive and close-to-cpu access. The fact that it is exclusive help with cache-coherence policies, while the fact that is close-to-cpu helps with the latency of the data. In addition, we radically reduce the dynamic memory allocation requests (a reduction of around the 98% of the memory allocation calls). In order to get this reduction, our optimized version makes an estimation of what will be the memory required and allocate it in advance. Then, it is able to track whether some extra memory is required to solve a particular expensive subproblem. This reduction of the memory allocation calls comes with benefits in performance.

This optimization satisfies the performance goal of increase the memory system performance.

The design and realization of these strategies achieves our last contribution, announced as:

Contribution 5 *We propose the designing of an efficient algorithm to solve the quadratic problems for Arbitrary factors. Our approach delivers an average speedup of 10.8X in a 6-core computer with respect to the publicly available implementation of AD^3.*

This contribution gives answer to the question

How to design a shared-memory parallel algorithm to solve relaxations of Side-Chain Prediction?

We make the source code of this work available at:
https://github.com/parad3-scp/source

1.5 Structure of the Thesis

In this chapter we have presented an overview of this dissertation, where we have identified our objectives and what are the solutions we propose, we also present what are our contributions.

Our work is in between of two disciplines, the *Combinatorial Optimization* field where we find applications for our contributions and the computer architecture field which provide us the means to reach them. In the next chapter we present a brief introduction to fundamental concepts that frame our work, both in *Combinatorial Optimization* and computer's architecture fields.

The next Chapters (3 to 5) are focusing on each one of the applications we are targeting in this thesis.

In Chapter 3, we describe the work we have done optimizing the IDP algorithm as well as proposing the first parallel version of IDP for the problem of the Coalition Structure Generation, after a brief introduction to the problem we present what are our contributions and our what are results.

Chapters 4 and 5 are closely linked since they present a workaround of the same algorithm: AD^3. However there are differences both in the application domain as well as in which aspect of AD^3 has been studied.

In Chapter 4 we propose to use AD^3 as a method to solve LP relaxations for the Winner Determination Problem for Large Scale Combinatorial Auctions. Beyond our contributions in the domain of the Combinatorial Auctions, in this chapter we show to effectively parallelize AD^3, giving rise to $PAR\text{-}AD^3$.

In Chapter 5 we apply $PAR\text{-}AD^3$ to solve LP relaxations to Side-Chain Prediction problems. In this chapter we also show how to optimize the execution inside a thread, since this time –as opposed to the previous chapter– most of the CPU load falls into the computation of the local subproblems.

We end the dissertation in Chapter 6, where we draw conclusions and present pointers to possible future lines.

Chapter 2

Background

In this chapter we present key concepts that are basic to understand the arguments and explanations contained in this document, both regarding computer's architecture and combinatorial optimization. The expert to any of this topics can freely skip parts of this background.

First we introduce relevant concepts of modern computer architecture. Such concepts, like parallelism, locality and others, will be recurrent during the rest of the dissertation and, in consequence, we draw an brief overview of them. Next, we present an outline of combinatorial optimization strategies that constitute a frame for the algorithms we work on.

2.1 Overview of Modern Computer Architecture

Computer design has vastly evolved from its beginnings. As Gordon Moore[1] predicted 50 years ago enunciating his very acclaimed Law, the chip industry has been able to improve enough in one year to fit as twice of the transistors than the previous year. Moore's Law was adjusted in 1975 to the pace to double the number of transistors every two years. Nowadays processors are able to integrate orders of 10^9 transistors in a single chip. They are extremely complex systems that implement many features in order to improve the system performance.

2.1.1 Parallelism

The use of parallelism is one of the most recurrent techniques that are implemented in current processors in order to speedup the execution of programs. We understand parallelism as any method that allows the processor to perform two or more tasks, either fine-grained or coarse-grained, at the same time.

[1] Co-founder and chairman emeritus of Intel Corporation

In 1972 Michael Flynn proposed a taxonomy of parallel architectures that has become very popular [Flynn, 1972]. According to Flynn, parallel systems can be categorized as:

- **SISD:** *Single Instruction Stream, Single Data Stream.*
  A processor executes a sequence or stream of instructions, each operating with a single data. It happens in a regular uni-processor machine.

- **SIMD:** *Single Instructions Stream, Multiple Data Stream.*
  A single sequence of instructions, with each single instruction operating on vectors containing a sequence of data. This kind of processing strategy happens in the vector operations (also called multimedia or SIMD operations) implemented in a modern computer.

- **MISD:** *Multiple Instructions Stream, Single Data Stream.*
  There is no machine implementing MISD.

- **MIMD:** *Multiple Instructions Stream, Multiple Data Stream.*
  The processor executed multiple Instruction Streams simultaneously, with each stream operating with a subset of a large set of data. This execution model happens in a multithreaded and/or multicore computer.

Flynn's taxonomy is the classical approach to classify parallel systems. Nowadays this taxonomy is still widely used although is not expressive enough to reflect current parallel hardware strategies. Let us examine different levels of parallelism present in modern processors.

Bit Level Parallelism (BLP)

BLP represents the most fine-grained use of parallelism, and takes part at the registers and logic-arithmetical units. Processors do not operate bit by bit, instead, they operate by blocks of bits (words). As for today, a regular desktop computer processor has words of 64 bits; this number has been increasing as the semiconductor technology has been improving. Increasing the number of bits of the processor words implies a big investment in the number of transistors and datapaths, but in reward, increments the processor capabilities, specially when working with large precision numbers.

Instruction Level Parallelism (ILP)

Processors increment the processor execution throughput by making use of the pipelining technique in order to execute multiple instructions at the same time. Pipelining is a strategy that consists in dividing the execution of each single instruction in different stages. The processor has different units specialized for the processing of every one of those stages. The main idea behind this approach is to overlap the execution of multiple instructions at the same time, and then maintain a full usage of the processor capabilities.

With this objective all the pipelined execution has to be synchronized to the speed of the slowest stage. This implies a slow down of the total execution process of each single instruction but in reward the system achieves a higher instruction throughput. Since there are instructions more complex than others, the processor pipeline contains different execution paths, with different resources, that make the total execution times of instructions to be different.

The ILP[2] of the program is a characteristic that is exploited automatically by the processor H/W, which means that it is something transparent to the programmer. In order to do that, the processor has to analyze the instruction dependencies to guarantee the correct result of the execution. Such dependencies are due to different factors and are categorized in three types: i) data dependencies, where one operand of an instruction is the result of a previous instruction, ii) name dependencies, where the same processor registers are used by different instructions containing different instances of data, and iii) control dependencies, where there are conditional jumps that determine the execution flow. The processor analyzes any kind of dependence and acts accordingly.

Modern compilers help to produce code that will be efficiently executed in parallel at the instruction Level. The expert programmer can also adapt the code to exhibit a higher degree of ILP. Techniques that help improving ILP are the reordering of instructions, the renaming of variables and the explicit unroll of loops.

For the programmers' eye, the instructions are virtually executed in parallel. Unfortunately, the programmer cannot see what is exactly happening inside the processor, but can obtain some feedback through profiling. A metric that allows the programmer to estimate how well the ILP is performing is the rate of Instructions Per Cycle (IPC), which can be retrieved from internal processor counters after executing the program.

Thread Level Parallelism (TLP)

It is possible to execute different threads running at the same time and sharing the same hardware and the same memory address space. Thread Level Parallelism, or TLP, is commonly understood as Parallelism; at the same time, algorithms that get benefit of TLP are known as Parallel Algorithms. Computer architectures having TLP capabilities are also known as Parallel Architectures.

From a hardware perspective, Thread Level Parallelism can be exploited in two different ways. The first usage of TLP is by multiplexing the same hardware resources to allow several threads to run concurrently (using H/W multithreading, or just multithreading). A second usage of TLP is replicating the different processor units, or processing cores, and executing one thread in each core. Today, a common situation is to have both multiple cores with multithreading

[2] ILP is also an acronym that stands for Integer Linear Programming, which is a topic studied in this dissertation. However in this section, ILP stands for Instruction Level Parallelism. Both acronyms are widely used in their respective communities.

capabilities.

- Hardware Multithreading requires a lower investment in terms of transistors, since functional units are shared by different threads. The most advanced multithreading mechanisms allow several threads to share the same processor core simultaneously. This mechanism improves the usage of the computation units, since some threads can issue work to those units while other threads are waiting due to some dependence situation. It is a good mechanism to hide the waiting latencies of single threads and keep busy the CPU with productive work when some threads are waiting.
- Multicore processors are based on the replication of the functional units, and integrate several complete cores inside the same chip. This implies that the computational power is multiplied, ideally providing a linear improvement in the performance as the number of cores running threads is increased.

Current processors do not divide the execution of a sequential program into separate execution of independent threads, and future processors are not expected to do so in the near future. Compilers can automatically parallelize the execution of a program only on very simple situations and for some code portions. Therefore, parallelization relies most of the time on the programmer, who has to identify opportunities for exploiting TLP and to actively design and write programs using explicit operations for thread creation, synchronization, and communication.

It is not realistic to expect an increase of performance directly proportional to the number of cores offered by the hardware. There are three main problems for achieving optimal performance from the parallel execution:

- **Amdhal's law**. Every algorithm has a sequential part, i.e. which cannot be arranged for parallel execution, and according to Amdhal's law the sequential part of the algorithm constrains the maximum overall speedup, following this pattern:

  $$Speedup = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

  where p is the fraction of parallel code and s the speedup obtained in this parallelization.

- **Unbalanced parallel work load.** Sometimes the amount of work that must be done by each parallel thread is unbalanced. In this situation, the execution time of the program will be limited by the execution time of the slowest thread, which will likely be the thread that is assigned more work to perform.

- **False cache-line sharing**. This problem arises when a core writes to a cached memory area in other thread's cache, producing an invalidation of that data and a performance degradation.

With all of that, trends in Computer Architecture are to integrate more cores in processors; for instance the Intel Xeon E5-2698V4 is a general-purpose processor with 20 cores and able to run up to 40 threads simultaneously. One architecture that deserves mention is the GPU architecture, which was initially conceived for graphics operations and nowadays includes thousands of independent cores. TLP (or parallelization) has become a very interesting paradigm for many applications and a challenging way to generate new solving approaches for already known problems.

Data Level Parallelism (DLP)

DLP is exploited by MIMD and SIMD processors, but is commonly referred as SIMD (Single Instruction Multiple Data) or also vectorization. In a vectorized operation, a single instruction is able to perform the same operation to different elements of a vector. This support for vectorized operations is frequently offered by means of specialized units. SIMD multiplies the throughput of the operations, hence speeding up the computation. SIMD capabilities were introduced as extensions to the Instruction Set Architecture (ISA) of popular processor architectures, like MMX and SSE for Intel, and AliVec for PowerPC.

The paradigmatic example for SIMD is the vector addition, where elements of two vectors are added. For instance, if our processor allows 512 bit vector operations and we require to sum two vectors of 64bit integers, a vectorized code will end executing 8 times less operations than a regular code.

The vectorization of instructions is responsibility of the programmer with the assistance of the compiler. Vectorization is usually achieved by decorators to the language like the use of openMP #pragmas in C or the vector syntax of Cilk Plus.

2.1.2 Memory Subsystem

The memory subsystem is comparatively slow compared to the processing capabilities. This performance difference has been widening through the last decades where industry has been able to improve processor performance at a higher rate than memory performance. This fact is known as the processor-memory performance gap. Figure 2.1 shows the growth of the processor-memory gap during the last decades.

A modern computer has different volatile memory types. From registers, which are very fast (around 0.25ns-0.5ns access time) but very limited in storage capacity at a high cost, to the main memory system which is slower (around 50-250 ns access time) but with a very large capacity and much cheaper.

The memory subsystem is organized following a hierarchic structure where there exist several levels of intermediate memories (memory caches). Caches can be specialized, (i.e. instruction cache and data cache) or general purpose. The number of caches depends on every design, but the most typical approach for

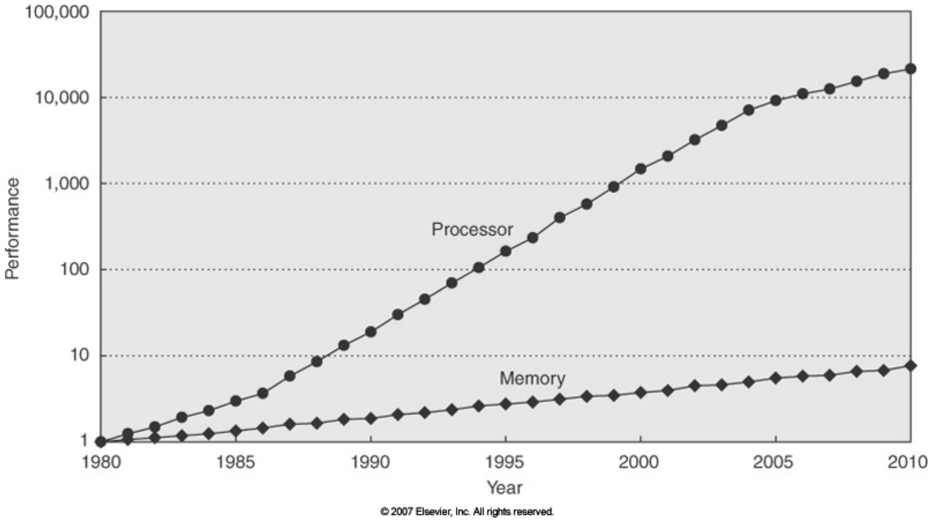


Figure 2.1: Processor-memory Gap. Image from [Hennessy and Patterson, 2011]

a current CPU is to implement two or three levels of cache. As long as more close to the processor a cache memory is, it has the advantages of less latency and higher bandwidth, but the drawback of a more limited capacity and higher price per storage unit.

The cache system is based on the exploitation of the principle of locality, given the fact that most of the programs manifest some patterns when accessing data. We distinguish two types of locality:

- Temporal locality: Data fetched from memory is very likely to be accessed again in the near future.
- Spatial locality: Data which is close in memory is frequently accessed close in time.

The information is transferred between the different memory levels in blocks named cache lines. Cache lines are the minimum unit to be read or written from/to memory and it is a few bytes long (nowadays the standard is 64 bytes).

From a hardware perspective, caching data for reading is quite easy. The cache system knows which blocks of data are in every specific cache and when they are going to be replaced they are simply released and overwritten. By the contrary, write operations are more problematic since they introduce a discrepancy of the data stored in the cache and the data stored in higher levels of memory. In a single-core processor, the problem is not very dramatic, since the update in the cache system after a write operation can be consolidated to the main memory either using a write-through policy (store operation is sent to cache and to the main memory at the same time) or a write-back policy (the

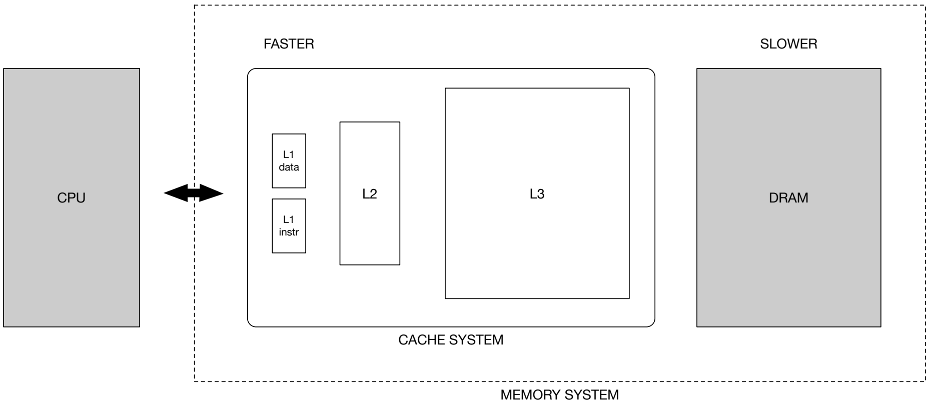


Figure 2.2: A typical mono-processor cache system

actual store to main memory is done when the block is released from cache).

In multicore systems, every core has its own private L1 and/or L2 cache. Data can be cached at different core's private cache at the same time, therefore there is a need to establish mechanisms able to ensure the cache coherence in all the system. There are different hardware techniques to keep the coherence between caches: one option is to invalidate every updated cache line in every foreign cache; another option is to notify other cores' caches about the change.

TLP has other major implications in the memory system. For the programmer, the most relevant consideration is that threads can be executed in any order, and there is no hardware synchronization between threads. Hence if some data is used by different threads, it is the programmer's responsibility to ensure that the result will be correct. This can be done either by restructuring the code to avoid conflicting situations, or by using primitives offered by the operating system to establish atomic operations or mutual exclusion zones, with the corresponding impact in the performance.

2.1.3 What Can a Software Architect Do?

Both the use of different levels of parallelism by of a program and the design of the data structures that are stored in memory and the access pattern to these data have a substantial impact in the execution performance. The software architect obtains benefits from many of the improvements in computer's architecture for free.

Nonetheless, programming languages and compilers do not offer the complete support to squeeze hardware capabilities. It requires a knowledge of the actual hardware executing and a right characterization of the executed code. Software architects carry out performance engineering thanks to the adoption of a methodology. We provide a brief description of key aspects of such methodology.

Code profiling. A profiler is a tool that generates execution reports. It can be very useful since it helps to discover which are the portions of code that are being executed the most and underline which are the critical parts of the code to optimize. For example, if 90% of the execution time is devoted inside a single function, it is worth to concentrate efforts in optimizing the execution of that particular function.

Use Thread Level Parallelism. TLP should be used at the higher level possible, parallelizing big chunks of execution. Depending on the problem, this parallelism could be hard to exploit. The ideal situation is to make every thread work with a disjoint set of data. In the most complex scenario, where using disjoint data is not possible, threads should be as much self-dependent as they can.

There exist two programming models that are able to express parallelism:

- **Message Passing**: Every thread has a private memory and threads share information using explicit messages. The main advantage of this paradigm is that it can be used both in multicore environments and also in distributed systems with separate memory address spaces. It can be useful when different threads do not need to share a large amount of data. Typically message passing application threads work with disjoint sets of data. There exists a standardization of message primitives [Forum, 1993].

- **Shared memory**: All the threads share the same memory space. Threads have access to the same memory positions and the programmer must deal with dependencies, data conflicts, race conditions and deadlocks. The main advantages of this paradigm are the higher performance achieved by very fast communication and the possibility to include parallelism in the code in an incremental way. A standard API for parallelism is OpenMP [Dagum and Menon, 1998].

Analyze memory access pattern and choose the right data representation. Data access is critical for applications, and the software designer should design the data representation so it can be accessed satisfying as much as possible the principles of spacial and temporal locality. Thus, maximizing the data re-use and promoting sequential access. A useful resource provided by the processor is the set of performance counters, providing information about Cache Hits and Misses.

Check the output of the compiler. Modern compilers provide information about code optimization. Examining the output of the compiler can provide valuable information of the optimization techniques applied to the code. Information such as vectorization, loop unrolling or code transformations are shown to the developer. A useful information is the problems preventing from performing certain optimizations; this can help the developer to adjust or improve the code in order to allow the compiler to do a better job.

Choose a compact representation. Using a compact representation in programs with high memory footprint helps to increase data access bandwidth in certain circumstances. The case of sparse matrix calculations is paradigmatic.

Identify bottlenecks. Thanks to profiling tools such as Linux kernel perf tools or Intel® VTune ™it is possible to inspect hardware counters and identify the bottlenecks of an application.

Use less instructions. This might look obvious, but a program that executes less instructions is -as a general rule- faster than a program that executes more instructions.

2.2 Combinatorial Optimization

Combinatorial optimization is a topic that consists of finding a set of objects from a finite set subject to some constraints and where a certain objective function is optimized.

A generic Combinatorial Optimization problem has:

- A finite set A.
$$A = \{1, 2, \ldots, n\}$$
- Weights (costs) defined as c_j for all $j \in A$.
- A set F of feasible subsets of A.
- an optimization function
$$min_{F \subseteq A} \left\{ \sum_{j \in S} c_j : S \in F \right\}.$$

This topic is a subset of mathematical optimization where the *combinatorial* term indicates that the optimization is applied in the discrete domain.

Combinatorial Optimization is also refereed in the literature as *discrete* or *integer* optimization [Nemhauser and Wolsey, 1988]. Although other classifications consider both *Combinatorial Optimization* and *Integer Programming* a subset of *Discrete Optimization* [Hammer et al., 2000].

Essentially, the three topics are very interleaved and the different nomenclatures come from facing the optimization problem from different perspectives. While *Combinatorial Optimization* emphasizes either the combinatorial origin, formulation or solution algorithm of a problem; *Discrete Optimization* emphasizes the difference to continuous optimization. *Integer Programming* emphasizes the usage of integer valued variables in the formulation or the solution.

Combinatorial Optimization is a wide topic with a large and diverse applicability to real world problems. The most classical and exemplified applications belong to the domains of scheduling, planning, or packing. A collection of relevant and paradigmatic examples of applications are discussed in [Paschos, 2013]. Nonetheless, optimization problems involving non divisible entities (i.e. people, objects, spaces, resources and more besides) are the objects of applicability of *Combinatorial Optimization* techniques, hence everyday widening applicability. As a little example, only in the area of Artificial Intelligence, *Combinatorial Optimization* is applied to solve problems in different research lines including agents, machine learning, auction theory or natural language processing.

One of the main characteristics of the *Combinatorial Optimization* instances is its hardness. The combinatorial behaviour of this family of problems comes with an enormous space of solutions. In many cases the combinations are too high that it is not assumable to process all the space of solutions along with it is not realistic to target the optimal solution. There is a large body of research presenting techniques to obtain reasonable solutions dodging the fact that in many cases it is not assumable to reach the optimal. Instead of finding the optimal, different methods do “the best they can do” given some computational resources and time.

We find three different approaches to tackle *Combinatorial Optimization* problems:

- **Find the optimal solution.** Where an algorithm runs until it finds a solution, providing or not intermediate results.
- **Find a sub-optimal solution.** Where an algorithm run-time is usually limited by some constraint (typically the run-time). Algorithms of this class provide a solution with or without measure of quality. They also may find the optimal even the may not be able to prove it.
- **Find an approximate solution.** Approximate algorithms are applied

to find a quick approximation. In general it is done by solving the Linear Relaxations of the problem. The approximate solution is useful as a bound or as an heuristic.

2.2.1 Methods Solving Combinatorial Optimization Problems

The complexity of *Combinatorial Optimization* Problems is variable. This is because the problem's structure of constraint varies depending on the problem itself.

As long as an algorithm is able to take profit of the particularities of a particular structure, the complexity varies. Recall that the complexity of a problem is defined by the lowest complexity of the set of algorithms solving that problem.

In [Ausiello et al., 2012] there is an extensive study of the different categorizations of complexity. This study is complemented with an online resource [Kann, 2009] where one can track algorithmic advances for each problem type. In practice, many of the different *Combinatorial Optimization* problems fall in the class of $\mathcal{NP}$-hard or $\mathcal{NP}$-complete.

The most naive approach to solve a *Combinatorial Optimization* problems is the enumeration of solutions. Since the solutions belong to the discrete domain –hence they are enumerable– and we have already an evaluation function by definition, a simple enumeration and evaluation of all the solutions leads to find the optimal. This strategy is obviously very expensive since the number of possible solutions trends to be very large.

According to [Paschos, 2012], there are two classical algorithmic approaches to solve the *Combinatorial Optimization* problems: Dynamic Programming [Bellman, 1954] and Branch and Bound [Land and Doig, 1960].

Dynamic Programming

Dynamic Programming is a special type of enumeration that avoids work by breaking down the problem into a sequence of problems, then establishing a recurrence link between their optimal solutions to build up the solution. This quite general strategy was first proposed by Bellman back in the 1950's [Bellman, 1954]. It is guaranteed to find the optimal solution but needs the complete execution of the algorithm to provide it.

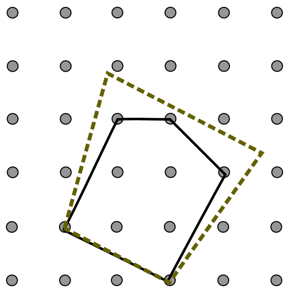


Figure 2.3: Linear Programming relaxation.

Branch-and-bound

Branch-and-bound explores the space of solutions. It uses bounds to cut the exploration of some paths and also uses an heuristic to guide the search more efficiently. Branch-and-bound algorithms are usually faster than Dynamic Programming as long as they have a good heuristic and they are able to compute good bounds. The search space is reduced as long as good is the bound we are able to compute. There is an enormous corpus of research regarding the three structural components of Branch-and-bound: different ways to compute bounds; a very large number of proposed heuristics and meta-heuristics; new ways of branch scheduling.

With regards to our work, we focus our attention on the bounding mechanism.

2.2.2 Bounding

In search, bounding is done with the aim of skipping work. The bounding is carried out thanks to the ability to over-estimate partial solutions quickly. And solutions can be over-estimated by transforming the *Combinatorial Optimization* problem into an regular less-constrained optimization problem, which is simpler to solve.

The classical yet powerful standard method to relax the problem is the known as Linear Programming (LP) relaxation, which consists in removing the integrality constraint.

Figure 2.3 shows the space of solutions for a given *Combinatorial Optimization* problem, where all the solutions are the dots inside the polygon with solid lines. The LP relaxation defines a new problem where the space of solutions is any point contained in the area delimited by the dotted line. Now solutions are no longer discrete points. Instead, solution is any point of the new defined area and it can be approximated using poligonometrics calculations.

In practice the industry has very powerful tools able to solve LP relaxations such as IBM ILOG CPLEX Optimization suite [Ibm, 2011] or GUROBI [Optimization, 2017]. They provide efficient implementations of classical algorithms such as Simplex and Barrier. They conform the state-of-the-art in optimization and carry a large history of success being the reference to benchmark new algorithmic proposals. Although being fairly general purpose algorithms, both commercial solvers perform really well on many domains, becoming a standard solution for any kind of optimization problems. The drawbacks of such programs is that they are quite general and do not exploit the particularities of specific problems.

Convex Optimization

When the linear problem is big enough, i.e. having thousands of variables and tens of thousands of constraints, finding the result of an LP relaxation using classic methods is computationally expensive.

In the last decades, there is a growing interest in convex optimization problems, which is a super class of Linear Programming. And more recently there is a family of algorithms known as "Proximal algorithms" that have been proven to be very successful when solving large-scale optimization problems.

> *(..) convex optimization problems (beyond least-squares and linear programs) are more prevalent in practice than was previously thought. Since 1990 many applications have been discovered in areas such as automatic control systems, estimation and signal processing, communications and networks, electronic q circuit design, data analysis and modeling, statistics, and finance. Convex optimization has also found wide application in combinatorial optimization and global optimization, where it is used to find bounds on the optimal value, as well as approximate solutions. We believe that many other applications of convex optimization are still waiting to be discovered.*
>
> *[Boyd and Vandenberghe, 2004]*

The key concept of the proximal algorithms is precisely their ability to face very big problems, they are able to do so by splitting the problem in many small pieces that can be solved using generic optimization methods.

The Alternate Direction Method of Multipliers (ADMM) is a proximal algorithm developed in the 1970s [Gabay and Mercier, 1976], and is closely related to many other algorithms, such as belief propagation, dual decomposition, the method of multipliers, and others.

ADMM was developed so early that there were hardly use cases for it. Nowadays, with the needs of the Big Data era, this algorithm have been proved useful to many applications, including data processing [Boyd et al., 2011], natural language processing [Rush and Collins, 2012], vision [Chan et al., 2017], and others.

ADMM is an algorithm that solves a problem of the form:

$$\begin{array}{ll} \text{minimize} & f(x) + g(z) \\ \text{subject to} & Ax + Bz = c \end{array}$$

with variables $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$, where $A \in \mathbb{R}^{p\times n}$, $B \in \mathbb{R}^{p\times m}$, $c \in \mathbb{R}^p$, and the functions f and g are convex. The main difference of this formulation with respect to a generic Optimization Problem is that the set of variables is divided in two, i.e. x and z, with the objective function separable across this splitting.

ADMM is an iterative algorithm that in every iteration performs a x-minimization step and a z-minimization step, followed by dual variable update according the Augmented Lagrange Method of Multipliers [Hestenes, 1969]. Updates on x and z are performed alternatively, hence the name of Alternate Directions.

Very recently, a variation of ADMM known as AD^3 [Martins et al., 2014] has been proposed as an approximation algorithm with good theoretical properties and advantages over ADMM:

- It is formally proved that it can run in parallel .
- It has a library of specific factors (functions), that can represent different types of constraints.
- It reaches consensus faster because the global state is notified to each sub-problem.

2.3 Analysis

Against this background we observe that *Combinatorial Optimization* is a large topic with high applicability to real world problems. Above the different techniques, commercial of-the-shelf products such as IBM CPLEX and GUROBI are very efficient and they are the industry standard. Although both IBM CPLEX and GUROBI target general-purpose optimization problems, they are very efficient and used widely in the industry.

Recent research in combinatorial optimization has shown how new techniques are very useful when targeting selected problems. At the same time, today computers are everyday more powerful but also more complex. However performance engineering considerations are largely ignored by the combinatorial optimization community.

We show how a better use of the computational resources can move one step forward in the performance execution of selected instances of combinatorial optimization problems.

Chapter 3

Building an efficient and parallel DP/IDP

3.1 Chapter Overview

In this chapter we tackle the optimization of the algorithms able to solve the Coalition Structure Generation using a Dynamic Programming strategy. Dynamic Programming strategy follows a brute-force approach, where all the possible solutions are evaluated.

This strategy has advantages and drawbacks. The main advantage is that this algorithm running time is not sensible to data input structure, therefore when there is not a defined pattern or distribution in the data, this algorithm is a safest choice. By the contrary other algorithms like IP [Rahwan et al., 2007], or D-IP [Michalak et al., 2010] are able to exploit data structure to reduce execution times, but in worst-case scenario their performance is dramatically worst.

As proposed in D-IP [Michalak et al., 2010], where a distributed anytime algorithm is presented, in this chapter we present an algorithm able to exploit the power of distribution but using a different paradigm. Our proposal is building a IDP based algorithm able to run in a shared memory scenario, which is common in nowadays computers [Sutter, 2005]. Using a shared memory paradigm simplifies the communication between computation nodes, since there is no need to send messages between them, but it requires a data dependence study, because of possible synchronization.

As far as we are concerned, no reference implementation neither of DP nor IDP algorithms has been published. Also there is not any version able to run in parallel. We have found, though, non-negligible issues on the algorithmic details that have a considerable impact on the overall performance.

The contributions of this work can be summarized as:

- We analyze and evaluate fast methods for generating splittings, the most critical operation, establishing that a bad choice can degrade performance by $10x$ or more.
- We parallelize the generation of splittings and execute the problem on a shared-memory, multi-core, multi-thread and multi-processor system.
- We identify the main performance bottleneck: both the sequential and parallel execution are limited by the lack of temporal and spatial locality of the memory access pattern, and by the weak support for irregular and scattered accesses provided by current memory hierarchies.
- We find out that the performance advantage of IDP versus DP is only realized for large problems, when reducing memory bandwidth requirements pay off.
- We make our code publicly available at the following URL: https://github.com/CoalitionStructureGeneration/DPIDP.

The rest of the chapter is organized as follows. First we present an introduce the background that frames the Coalition Structure Generation Problem. Then we show how this problem is tackled by Dynamic Programming algorithms. Thereafter we show details of our optimizations in sequential and propose a method to parallelize DP/IDP. Finally we end with an experimentation section where we characterize the algorithm performance.

3.2 Coalition Formation

The multi-agent systems research field belongs to the artificial intelligence area where agents are autonomous software entities coexisting in a closed system. Agents have an individual vision of their environment and they have their our goals, preferences, beliefs and capabilities [Shoham and Leyton-Brown, 2008].

Agents are able to interact with other agents in order to satisfy its own goals. In fact, one of the key aspects of the multi-agents research field is the ability of the agents to communicate between themselves in order to satisfy their individual or collective objectives. Collaboration can be useful if the agents are *cooperative* where they pursue a shared goal but also if they are *selfish*, since their own goals may exceed agent sole capabilities.

Agents collaborate following different paradigms, every one has its strengths and weaknesses [Horling and Lesser, 2004]. One of the paradigms of major success is the formation of *coalitions* where a group of agents establish a collaboration in order to satisfy a common objective for a given period of time.

Coalition Formation is a potential method for face a substantial scope of applications including distributed vehicle routing [Sandholm and Lesser, 1997], cooperative transmission in wireless networks [Han and Poor, 2009], task allocation [Shehory and Kraus, 1998], among many others.

According to [Sandholm and Lesser, 1997] the coalition formation process involves three main activities.

- *The Coalition Structure Generation.* All the agents are grouped in teams forming a structure of exhaustive and disjoints groups where all the members of a group collaborate to reach a given goal. Some coalitions are preferred than others and they are scored in order to evaluate them. Scores can be defined by a designer of the system or by the agents themselves. An optimal Coalition Structure is a structure that maximizes the global score of the every all the possible coalitions.
- *Solving the optimization problem in every coalition.* The agents members of a coalition start to collaborate to solve a specific problem. There is no collaboration with agents outside the coalition.
- *Dividing the reward of every coalition.* Members of a coalition get benefit of the coalition gain reached thanks to the coalition. This gain can be evenly distributed among agents or with an agreed distribution in advance.

The Coalition Structure Generation is a computational-intense process that can be modeled as a cooperative game. If the valuations of every coalitions is influenced by the rest of coalitions, then the model is based on *partition function games* [Thrall and Lucas, 1963] if by the contrary, valuations depends only on the members of each coalition, then this situation is modeled as a *characteristic function game.*

We tackle the Coalition Structure Generation Problem as characteristic function game. In particular we are interested in optimizing exact algorithms able to solve the Coalition Structure Generation. This problem is equivalent to the Complete Set Partitioning [Lin, 1975], it is also identical to the Winner Determination Problem in Combinatorial Auctions [Lehmann et al., 2006].

3.3 Key Definitions

Let us define some role-playing entities involved in the Coalition Structure Generation process for multi-agent systems.

- **Agent** (a_x): A single agent. E.g. Ann or Bob.

- **Agent Set** (A): The set of all available agents. $A = \{a_1, a_2, \dots, a_n\}$.
E.g. A= {Ann, Bob, Chris, Dave}.
- **Coalition** (C): $C \subseteq A$. C is a subset of A that contains the agents participating in a coalition.
E.g. C= {Ann, Chris, Dave}.
- **Split** : Is the operation performing a binary partition of a coalition.
E.g. {Ann,Chris,Dave} → ({Ann},{Chris,Dave}).
- **Splitting** : Is the result of the split operation. A splitting is a 2-tuple represented by (C_1, C_2). C=$C_1 \cup C_2$ where $|C_1|$,$|C_2|$ >0, $C_1 \cap C_2$=$\emptyset$.
E.g. ({Ann},{Chris,Dave}) or ({Ann,Chris},{Dave}).
- **Coalition Structure** (CS): Is a collection of disjoint Coalitions such that their union constitute the Agent Set.
E.g. ({Ann},{Bob},{Chris,Dave}).
- **The evaluation function of a single coalition** (v(C)), or simply the value of a coalition: Is a function returning a value representing the reward of this coalition. We will assume that this function reads values from a table.
- **The evaluation fucntion of a CS** (V(CS)), or the value of a CS. Is a fucntion returning a value representing the reward of a coalition structure. It is computed by adding the values of all the CS member coalitions.
- **The set of all possible CSs** (Π^A): Is a set that contains all the possible coalition structures over the Agent Set.

The Coalition Structure Generation problem is formally defined as the problem of finding a Coalition Structure (CS) over the Agents (A) with maximal value. This maximal CS is named optimal and it is denoted by CS*.

$$CS^* = arg\,max_{CS \in \Pi^A} V(CS) \tag{3.1}$$

the value of a given CS is determined by the value of all included coalitions.

$$V(CS) = \sum_{C \in CS} v(C) \tag{3.2}$$

As an example, consider an agent set of size 4 whose agents are A=\{a_1,a_2,..., a_n\}. Agents can establish coalitions with other agents in order to perform a task. Every possible coalition has a particular value associated, those values are shown in the table 3.1. The goal of the CSG problem is to find the coalition structure providing maximum global satisfaction. From Table 3.1 one can notice that the coalition formed by \{a_2,a_3\} has lower value than the sum of $v(\{a_2\})$ and $v(\{a_3\})$, meaning that agents a_2 and a_3 prefer to work alone rather than collaborate. Just looking at the values on the table one can realise the difficulty of the problem, even for this moderated size.

C	$v(C)$
$\{a_1\}$	33
$\{a_2\}$	39
$\{a_3\}$	13
$\{a_4\}$	40
$\{a_1,a_2\}$	87
$\{a_1,a_3\}$	87
$\{a_1,a_4\}$	70
$\{a_2,a_3\}$	36
$\{a_2,a_4\}$	52
$\{a_3,a_4\}$	67
$\{a_1,a_2,a_3\}$	97
$\{a_1,a_2,a_4\}$	111
$\{a_1,a_3,a_4\}$	100
$\{a_2,a_3,a_4\}$	132
$\{a_1,a_2,a_3,a_4\}$	151

Table 3.1: Coalition values for a CSG problem of size 4.

3.4 Algorithms to Solve the CSG

Combinatorial Optimization problems contain solutions belonging to the discrete space. This imply that every possible solution can be enumerated and evaluated. Therefore the most simple -and inefficient- approach to solve the problem is a brute-force algorithm that explores all the possible partitions of the Agent Set in Coalition Structures. Unfortunately the number of possible CS is a very large number known as bell number and denoted by B_n, satisfying the inequation:

$$\alpha n^{n/2} \leq B_n \leq n^n \tag{3.3}$$

where n is the number of agents and α is a positive constant [Sandholm et al., 1999].

3.4.1 Dynamic Programming Algorithms

Dynamic Programming algorithms are able to reach the optimal solution performing an exhaustive search of the space of solutions. Several algorithms following a Dynamic Algorithms strategy has been proposed so far.

The first Dynamic Programming algorithm able to solve the Coalition Structure Generation problem was the DP algorithm [Yun Yeh, 1986].

3.4.2 DP Algorithm

The DP [Yun Yeh, 1986] algorithm uses Dynamic Programming to find the optimal solution of the problem. For a given input data, DP first evaluates all the possible coalitions of size 2. For each possible pair of agents a_x and a_y, DP evaluates if it is better to form a coalition or not. This is done by comparing $value[\{a_x\ ,a_y\}]$ with $value[\{a_x\}]+\ value[\{a_y\}]$. The maximum value represents the preferred formation and substitutes the previous $value[\{a_x,\ a_y\}]$.

After evaluating all coalitions of size 2, DP starts evaluating all possible coalitions of size 3, saving the maximum between $value[\{a_x,\ a_y,\ a_z\}]$ and all its possible splittings. There are three ways to split the coalition: $\{a_x,\ a_y\}+\{a_z\}$, $\{a_x,\ a_z\}+\{a_y\}$ and $\{a_x\}+\{a_y,\ a_z\}$. Note that all the splittings for coalitions of size 3 have at most 2 elements. Since DP evaluates the coalitions of size 3 after evaluating and finding optimal values for coalitions of size 2, the new coalition values computed for size 3 will also be optimal. This process is repeated increasing the size of the coalitions (m).

Algorithm 1 Pseudo-code of the DP Algorithm

```
 1: for $m = 2 \rightarrow n$ do
 2:     for $C \leftarrow coalitionsOfSize(m)$ do                 ▷ $\binom{n}{m}$ iterations
 3:         $max\_value \leftarrow value[C]$
 4:         $C_1 \leftarrow getFirstSplit(C)$
 5:         while ($C_1$) do                                     ▷ $2^{n-1} - 1$ iterations
 6:             $C_2 \leftarrow C - C_1$
 7:             if ($max\_value < value[C_1] + value[C_2]$ then
 8:                 $max\_value \leftarrow value[C_1] + value[C_2]$
 9:             end if
10:             $C_1 \leftarrow getNextSplit(C_1)$
11:         end while
12:         $value[C] \leftarrow max\_value$
13:     end for
14: end for
```

The DP algorithm (see Algorithm 1) is composed of three nested loops: (i) the outer loop (line 1), where coalition size (m) grows from 2 to the total number of agents (n), (ii) the intermediate loop (line 2), where all coalitions of size m are generated, a total of $\binom{n}{m}$, and (iii) the inner loop (line 5), where each coalition is split and evaluated, a total of $2^{m-1} - 1$ splittings. The temporal complexity of the DP algorithm is determined by these three loops: $\Theta(3^n)$.

Table 3.2 shows a trace of the DP algorithm for a problem of size 4. Where

Coalition size selection	*Coalition* Generation (A)	*Coalition* Value ($v[A]$)	Splittings Generation	Value of splittings (vs)	Stored value $max(v[A], vs)$
m=2	$\{a_1,a_2\}$	87	$\{a_1\},\{a_2\}$	v[$\{a_1\}$]+v[$\{a_2\}$]=72	87
	$\{a_1,a_3\}$	87	$\{a_1\},\{a_3\}$	v[$\{a_1\}$]+v[$\{a_3\}$]=46	87
	$\{a_1,a_4\}$	73	$\{a_1\},\{a_4\}$	v[$\{a_1\}$]+v[$\{a_4\}$]=73	73
	$\{a_2,a_3\}$	36	$\{a_2\},\{a_3\}$	v[$\{a_2\}$]+v[$\{a_3\}$]=52	52
	$\{a_2,a_4\}$	52	$\{a_2\},\{a_4\}$	v[$\{a_2\}$]+v[$\{a_4\}$]=53	53
	$\{a_3,a_4\}$	67	$\{a_3\},\{a_4\}$	v[$\{a_3\}$]+v[$\{a_4\}$]=53	67
m=3	$\{a_1,a_2,a_3\}$	97	$\{a_1\},\{a_2,a_3\}$	v[$\{a_1\}$]+v[$\{a_2,a_3\}$]=85	97
			$\{a_2\},\{a_1,a_3\}$	v[$\{a_2\}$]+v[$\{a_1,a_3\}$]=126	126
			$\{a_3\},\{a_1,a_2\}$	v[$\{a_3\}$]+v[$\{a_1,a_2\}$]=100	126
	$\{a_1,a_2,a_4\}$	111	$\{a_1\},\{a_2,a_4\}$	v[$\{a_1\}$]+v[$\{a_2,a_4\}$]=98	111
			$\{a_2\},\{a_1,a_4\}$	v[$\{a_2\}$]+v[$\{a_1,a_4\}$]=112	112
			$\{a_4\},\{a_1,a_2\}$	v[$\{a_4\}$]+v[$\{a_1,a_2\}$]=127	127
	$\{a_1,a_3,a_4\}$	100	$\{a_1\},\{a_3,a_4\}$	v[$\{a_1\}$]+v[$\{a_3,a_4\}$]=100	100
			$\{a_3\},\{a_1,a_4\}$	v[$\{a_3\}$]+v[$\{a_1,a_4\}$]=112	112
			$\{a_4\},\{a_1,a_3\}$	v[$\{a_4\}$]+v[$\{a_1,a_3\}$]=127	127
	$\{a_2,a_3,a_4\}$	132	$\{a_2\},\{a_3,a_4\}$	v[$\{a_2\}$]+v[$\{a_3,a_4\}$]=106	132
			$\{a_3\},\{a_2,a_4\}$	v[$\{a_3\}$]+v[$\{a_2,a_4\}$]=78	132
			$\{a_4\},\{a_2,a_3\}$	v[$\{a_4\}$]+v[$\{a_2,a_3\}$]=92	132
m=4	$\{a_1,a_2,a_3,a_4\}$	151	$\{a_1\},\{a_2,a_3,a_4\}$	v[$\{a_1\}$]+v[$\{a_2,a_3,a_4\}$]=165	165
			$\{a_2\},\{a_1,a_3,a_4\}$	v[$\{a_2\}$]+v[$\{a_1,a_3,a_4\}$]=166	166
			$\{a_3\},\{a_1,a_2,a_4\}$	v[$\{a_3\}$]+v[$\{a_1,a_2,a_4\}$]=140	166
			$\{a_4\},\{a_1,a_2,a_3\}$	v[$\{a_4\}$]+v[$\{a_1,a_2,a_3\}$]=166	166
			$\{a_1,a_2\},\{a_3,a_4\}$	v[$\{a_1,a_2\}$]+v[$\{a_3,a_4\}$]=154	166
			$\{a_1,a_3\},\{a_2,a_4\}$	v[$\{a_1,a_3\}$]+v[$\{a_2,a_4\}$]=152	166
			$\{a_1,a_4\},\{a_2,a_3\}$	v[$\{a_1,a_4\}$]+v[$\{a_2,a_3\}$]=125	166

Table 3.2: Trace of execution of a problem of size 4.

the optimal coalition structure have a value of 166 units. Finding what are the members of the coalition structure can be retrieved navigating backwards. Starting for the bigger size split, we know that best splitting of the last level is {{a_2},{a_1,a_3,a_4}}. From there we inspect the two members of the splitting. i.e. {a_2} and {a_1,a_3,a_4}. The first member is a size-one coalition, thus is already optimal meanwhile the second one has three members. So we will need to look at the splittings of this particular coalition. Looking at the table we see that the best splitting {a_1,a_3,a_4} is {{a_4},{a_1,a_3}}. Now we have again two members of the splitting, but both are optimal. So the best coalition structure will be {{a_2},{a_4},{a_1,a_3}}

3.4.3 IDP Algorithm

While DP generates all the possible splittings of each coalition, IDP [Rahwan and Jennings, 2008b] introduces conditions to avoid the generation and evaluation of a large amount of splittings. The performance advantage of IDP is a reduction in the total number of operations and memory accesses. Overall, IDP explores only between 38% and 40% of the splittings explored by DP for problems from 22 to 28 agents. Algorithm 2 presents the pseudo-code of IDP, where the main changes are the filters introduced on lines 4 and 6.

Algorithm 2 Pseudo-code of the IDP Algorithm

```
 1: for $m = 2 \rightarrow n$ do
 2:     for $C \leftarrow coalitionsOfSize(m)$ do                      ▷ $\binom{n}{m}$ iterations
 3:         $max\_value \leftarrow value[C]$
 4:         $(lower\_bound, high\_bound) \leftarrow IDPBounds(n, m)$
 5:         $C_1 \leftarrow getFirstSplit(C, lower\_bound)$
 6:         while $(sizeOf(C_1) \leq high\_bound$ do
 7:             $C_2 \leftarrow C - C_1$
 8:             if $(max\_value < value[C_1] + value[C_2]$ then
 9:                 $max\_value \leftarrow value[C_1] + value[C_2]$
10:             end if
11:             $C_1 \leftarrow getNextSplit(C_1, C)$
12:         end while
13:         $value[C] \leftarrow max\_value$
14:     end for
15: end for
```

3.5 Single-Thread Implementation

In this section we analyze the operations of generating and evaluating splittings inside the inner loop, which consumes ≈ 99 % of the execution time. We compare two suitable options and analyze their performance and the impact of the memory access pattern.

3.5.1 Data Representation

The coalitions and their associated values are stored in a vector. A coalition is represented using an integer index where the bit at position x of the index indicates that agent x is a member of the coalition. The index determines the vector element containing the coalition value. Using this representation, the input of the CSG problem fits into a vector of $2^n - 1$ positions. With coalitions represented by 4-byte words, we can run problems up to 32 agents.

3.5.2 Splitting Generation

The splitting generation problem can be reduced to the subset enumeration problem, since each coalition splitting is composed by a subset, C_1, and its complementary, C_2. Generating all the subsets C_1 from a coalition C and then calculating the complementary $C_2 = C - C_1$, though, would produce the same splitting twice: once for each of the splitting subsets. We remove one element from the coalition (the agent with the highest rank) when performing the subset enumeration, so that the removed element is never part of the enumerated subset and always belongs to its complementary.

There exist several ways of enumerating subsets [Loughry et al., 2000], like banker's sequence, lexicographical order, and gray codes. The banker's sequence seems a suitable option for IDP, since it generates the splittings in growing order of $|C_1|$, and then simplifies the filtering of splittings by its size. Figure 3.1a shows a scheme of the banker's sequence operation for C=$\{a_1,a_4,a_5,a_6,a_7\}$, and assuming that only coalitions with $|C_1|$=2 need to be evaluated. Note that element a_7 is always assigned to the complementary subset (lighted colour). The generation starts directly from the first splitting of size $|C_1| = 2$, follows with the remaining $\binom{4}{2} - 1$ subsets of the same size, and stops before generating the first subset of size 3. The code does not waste instructions generating useless subsets.

When generating splittings in lexicographical order (see Fig. 3.1b), some filtering code is required to check that the size of the splitting ranges between a given pair of bounds. Execution resources are wasted to generate splittings that are then discarded, and to perform the filter check. In Fig. 3.1, only 6 out of 14 splittings are actually needed (note the check and discard crossed signs).

Both methods were implemented using recurrent functions that calculate the next splitting from the previous one. The lexicographical order was implemented

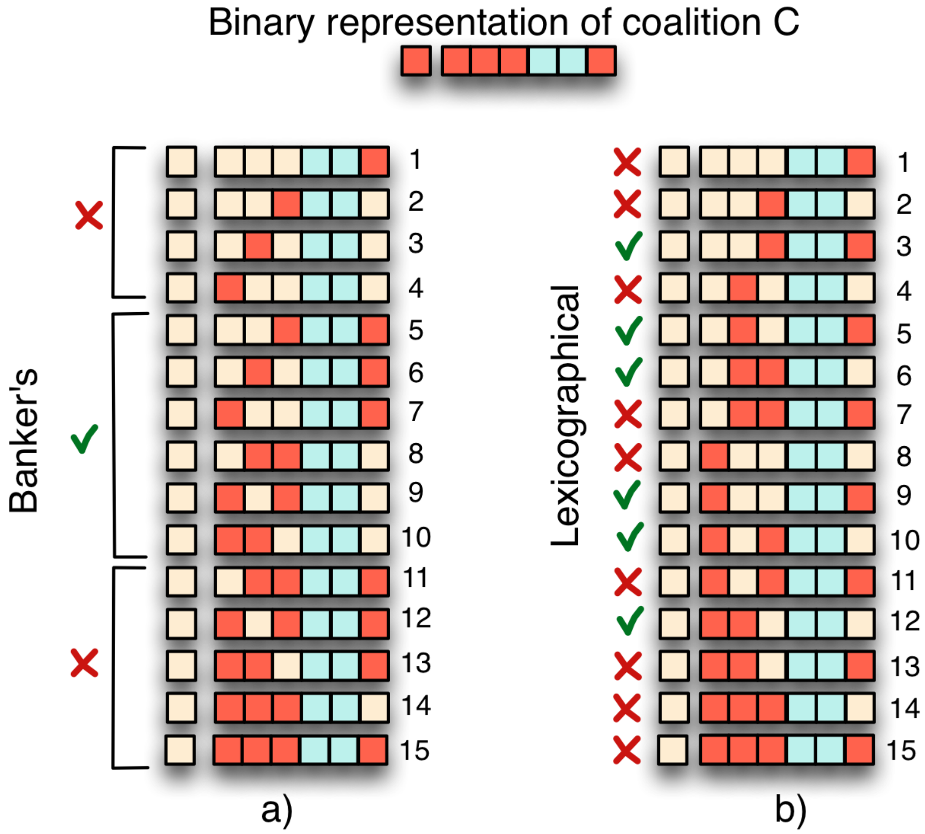


Figure 3.1: a) Banker's sequence versus b) lexicographical order.

with a few number of very simple operations: $C_1 \leftarrow (C_1 + C^{**})$ AND C, where C^{**} is the two's complement of C, that can be precalculated for all the splittings of a given coalition. The whole splitting code requires only few machine code instructions in a current x86 ISA. On the other hand, our implementation of banker's sequence, an improved version of the algorithm published in [Loughry et al., 2000], required, on average, 6 times more instructions. More details about the implementation, like the usage of a special population count instruction for computing $|C_1|$, can be found in the published code.

```
.L72:
  addl       %r10d, %edx
  movl       %eax, %ecx
  andl       %eax, %edx
  subl       %edx, %ecx
  movslq     %edx, %r9
  movslq     %ecx, %rcx
  movl       (%r8,%rcx,4), %ecx
  addl       (%r8,%r9,4), %ecx
  cmpl       %ecx, %esi
  cmovl      %ecx, %esi
  addl       $1, %edi
  cmpl       %r13d, %edi
  jbe        .L72
```

3.5.3 Memory Accesses

All memory accesses correspond to reads from the vector of coalition values performed in the inner loop of the algorithm, and a few writes on the intermediate loop. The total number of data read operations done by the DP algorithm is around $2{\times}3^n$. As explained above, IDP evaluates only a subset of the splittings, corresponding to 38%-40% of the read operations performed by DP.

The memory-level parallelism of the algorithm is moderate. The inner loop recurrence can generate multiple independent read requests, without having to wait for data, subject to storage availability for pending requests and for the window of instructions blocked on those data.

The data-reuse degree of the algorithm is high. There are 2^n elements in the *value* vector, and so the average number of reads to the same data item is $\approx 2{\times}(3/2)^n$ ($\approx$100,000 for $n = 27$). However, accesses to the same item are scattered in time, specially when the algorithm analyzes medium- or large-size coalitions. The combinatorial nature of the problem involves a pseudo-random read access pattern, where reads that are consecutive in time refer to data from distant positions in memory.

The bad performance behavior of the memory access pattern arises for vectors that do not fit into the processor's cache. The vector size is 2^{n+2} bytes, which is 16 MBytes for n=22. For larger n's an important amount of vector accesses will miss the cache and will request a full 64-Byte cache block to DRAM. This creates both latency and bandwidth problems. The moderate memory-level parallelism helps hiding part of the DRAM latency but, as we will show later, an important amount of this latency is exposed in the execution time. Also, given the lack of spatial locality, most of the 64-Byte block read from DRAM will be unused. In the worst situation, only 4 Bytes out of 64 will be used, giving a bandwidth efficiency of 1/16= 0.0625.

3.6 Multi-thread Implementation

This section analyzes the algorithm's data workflow in order to find its potential thread-level parallelism (TLP). Exploiting concurrency efficiently is not straightforward, and a new method to generate coalitions is devised. Finally, potential performance problems are described.

3.6.1 Identifying Sources of TLP

The simplest and most efficient approach is always to parallelize the outer loop of a program. DP and IDP, though, exhibit loop-carried dependencies on the outer loop: the optimal values for coalitions of size m must be generated before using them for generating the optimal values for coalitions of size $m + 1$.

The intermediate loop generates all the coalitions of a given size, and for each coalition it analyzes all the splittings of certain sizes. Tasks corresponding to coalitions are independent: they only modify the value associated to the coalition, and only read values corresponding to coalitions of lower size. Therefore, there cannot exist read-after-write (RAW) dependencies nor any other false data dependence among the tasks. However, the single-thread code was designed to accelerate coalition generation by using an inherently sequential algorithm that uses the previous coalition to generate the next one in lexicographical order. The next subsection describes a method for breaking this artificial dependence.

3.6.2 Speeding up Work Distribution Among Threads

Assume we have t threads and we want each thread to evaluate a disjoint set of coalitions. We must distribute work to assure good load balance, and do it in a fast and efficient way. Table 3.3 illustrates the generation of all the possible coalitions of size m=3 from a set of n=6 agents. The single-thread code implements a sequential algorithm to generate in lexicographical order all $\binom{6}{3}$=20 coalitions, represented as bitmaps in the binary encoding columns of Table 3.3. In practice, we must calculate cnt=$\binom{n}{m}$ and then assign cnt/t coalitions to each thread. Once a thread obtains its starting position in the coalition series, say k, it can generate the whole range with the fast sequential method. But we need an efficient strategy to generate the k^{th} coalition without having to compute all the previous coalitions from the beginning.

Order	Encoding		Coalitions	Order	Encoding		Coalitions
(k)	Bin	Dec		(k)	Bin	Dec	
1	...111	7	$\{a_1, a_2, a_3\}$	11	..111.	14	$\{a_2, a_3, a_4\}$
2	..1.11	11	$\{a_1, a_2, a_4\}$	12	.1.11.	22	$\{a_2, a_3, a_5\}$
3	.1..11	19	$\{a_1, a_2, a_5\}$	13	1..11.	38	$\{a_2, a_3, a_6\}$
4	1...11	35	$\{a_1, a_2, a_6\}$	14	.11.1.	26	$\{a_2, a_4, a_5\}$
5	..11.1	13	$\{a_1, a_3, a_4\}$	15	1.1.1.	42	$\{a_2, a_4, a_6\}$
6	.1.1.1	21	$\{a_1, a_3, a_5\}$	16	11..1.	50	$\{a_2, a_5, a_6\}$
7	1..1.1	37	$\{a_1, a_3, a_6\}$	17	.111..	28	$\{a_3, a_4, a_5\}$
8	.11..1	25	$\{a_1, a_4, a_5\}$	18	1.11..	44	$\{a_3, a_4, a_6\}$
9	1.1..1	41	$\{a_1, a_4, a_6\}$	19	11.1..	52	$\{a_3, a_5, a_6\}$
10	11...1	49	$\{a_1, a_5, a_6\}$	20	111...	56	$\{a_4, a_5, a_6\}$

Table 3.3: Coalitions generated using lexicographical order.

Algorithm 3 describes $getCoalition(n, m, k)$, a function that generates the k^{th} coalition in lexicographical order of m elements from a set of n. The description is done recursively to help understand how it works, although the actual implementation is iterative in order to improve its performance. The coalition is created recursively, bit by bit, starting from the least significant bit and considering $\binom{n}{m}$ possibilities. The first half of the possible coalitions have the less significant bit set to 1. If the requested rank, k, is lower than or equal to h=$1/2\times\binom{n}{m}$, then the bit is set to 1, and m is decremented by one. Otherwise, the bit is set to zero, and the rank k is reduced to $k - h$. Each recursive call decrements the number of bits to consider to $(n - 1)$.

Algorithm 3 pseudocode of $getCoalition(n, m, k)$

```
if (($m == 0$) OR ($k == 0$)) then
    return 0
end if
$h \leftarrow \binom{n-1}{m-1}$
if ($k \leq h$) then
    return 1 + 2×$getCoalition(n-1, m-1, k)$
end if
return 2×$getCoalition(n-1, m, k-h)$
```

3.6.3 Potential Parallel Performance Hazards

The first and last iterations of the outer loop exhibit few TLP, compromising the efficiency of the parallel execution. We tuned the implementation so that threads are launched in parallel only for iterations that have a minimum amount of work. A minor problem is the need for a few number of synchronization barriers at the end of every iteration of the outer loop. They can be neglected, except for very small problem sizes.

An important performance issue is the occurrence of false cache sharing misses. They occur when different threads update different positions in the vector of values that happen to be mapped to the same cache line.

Finally, there is also the issue of true cache sharing. Threads generate values for coalitions of size m that are stored into local caches. When all the threads need to access those values for handling larger coalitions, data has to be moved from local storage to all the execution cores.

3.7 Experimental Results

The computer system used in our experiments is a dual-socket Intel Xeon E5645, each socket containing 6 Westmere cores at 2.4 GHz, and each core executing up to 2 H/W threads using hyperthreading (it can simultaneously execute up to 24 threads by H/W). The Last Level Cache (LLC) provides 12 MiB of shared storage for all the cores in the same socket. 96 GiB of 1333-MHz DDR3 RAM is shared by the 2 sockets, providing a total bandwidth of 2×32 GB/sec. The Quickpath interconnection (QPI) between the two sockets provides a peak bandwidth of 11.72 GB/sec per link direction.

Input data was created using a uniform distribution as described by [Larson and Sandholm, 2000] for problem sizes $n = 18 \ldots 27$.

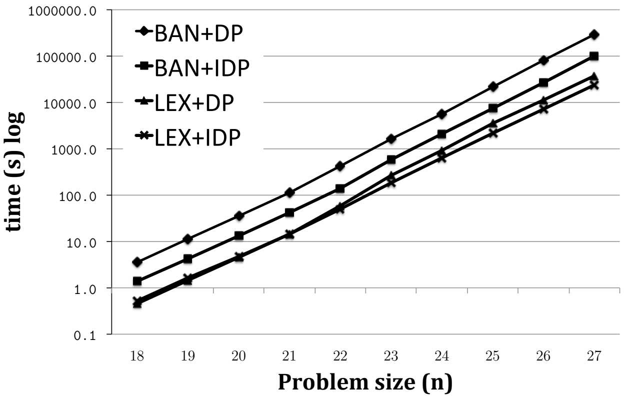


Figure 3.2: Execution time (log).

3.7.1 Single-Thread Execution

DP and IDP were executed using both the banker's and lexicographical splitting generation methods. Figure 3.2 plots the execution time in logarithmic scale for the four algorithmic variants. Lexicographic order is around $7x$ to $11x$ faster than banker's and, therefore, in what remains of chapter we will use the first splitting method.

Figure 3.3 represents the execution time of DP and IDP divided by 3^n (algorithmic complexity). This metric evaluates the average time taken by the program to execute a basic algorithmic operation, in this case a splitting evaluation. It is similar to the CPI (Cycles Per Instruction) metric, but at a higher level. The metric helps identifying performance problems at the architecture level. Figure 3.3 shows two different problem size regions: those that fit into the LLC (n<22), and those that do not. A small problem size determines a computation-bound scenario, where DP slightly outperforms IDP, even when it executes around 20% more instructions. The reason is that IDP is penalized by a moderate number of branch mispredictions.

Large problem sizes determine a memory-bound scenario, where IDP amortizes its effort on saving expensive memory accesses to outperform DP by 40-50%. Figure 3.4 shows the effective memory bandwidth consumption seen by the programs. The shape of the curves can be deduced from Figure 3.3, but we are interested on the actual values. The effective bandwidth ranges between 0.5 and 1.0 GB/sec. A small fraction of this bandwidth comes from the LLC and lower-level caches, and the remaining fraction comes from DRAM. Even considering the worst case described in section 3.3, that only 4 bytes out of the 64-Byte cache block are effectively used, it is still a very small value compared

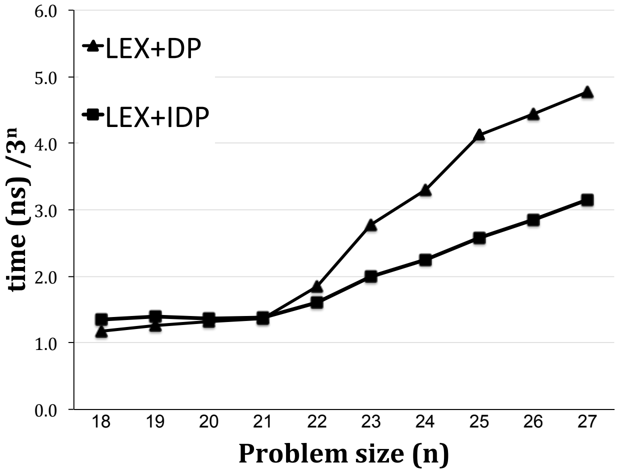


Figure 3.3: Time / Complexity $\Theta(3^n)$.

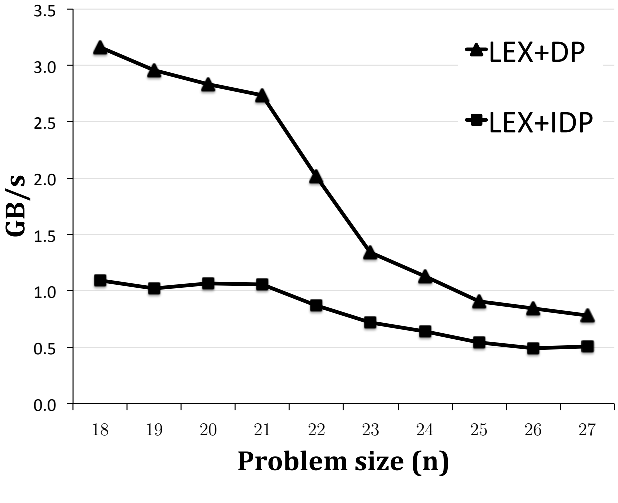


Figure 3.4: Effective Memory Bandwidth (GB/s).

to the peak 32 GB/sec. The conclusion is that DRAM latency is the primary performance limiter. Results on the next subsection corroborate this conclusion.

3.7.2 Multi-Thread Execution

We focus our multi-thread analysis on IDP, which outperforms DP for interesting problem sizes. We run IDP using t= 6, 12, and 24 threads. The case t=6 corresponds with using a single processor socket. The case t=12 uses only one socket but also exploits its hyperthreading capability. Finally, t= 24 is an scenario where all 2 sockets have their 6 cores running 2 threads each, using hyperthreading. Figure 3.5 shows the speedup compared to the single-thread execution. Again, distinguishing between small and big problem sizes is useful.

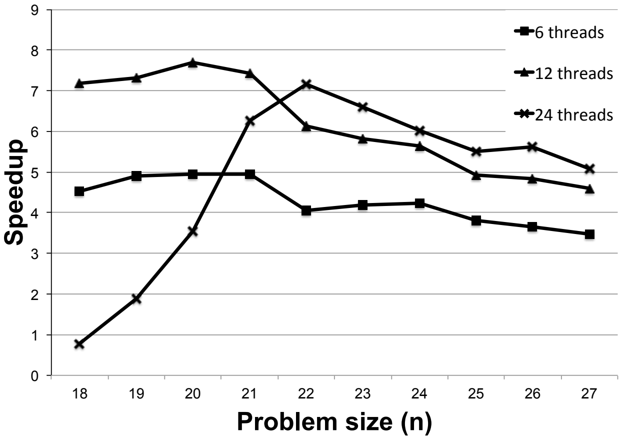


Figure 3.5: Single-thread IDP versus 6-, 12- and 24-thread IDP execution

The t=6 configuration provides a speedup of 5 for small problems, and lower than 4 for large problems. The t=12 configuration further increases performance around 60% for small problems, and 30% for bigger problems. The fact that executing two threads per core do improves performance corroborates previous latency limitations, since hyperthreading is a latency-hiding mechanism. It also indicates that 6 threads do not generate enough LLC and DRAM requests to fully exploit the available LLC and DRAM bandwidth.

The effective memory bandwidth achieved with 12 threads is around 2.5 GB/sec for the bigger problem sizes, or around 13 times lower than the peak achievable bandwidth. Given the lack of spatial locality of DRAM accesses, we are probably reaching the maximum bandwidth available for the pseudo-random memory access pattern of the problem.

The t=24 configuration checks the benefit of using a second socket. Performance is highly penalized for small problems, due to the overhead of communication traffic along the QPI links for both false and true cache sharing coherence. On average, half of the data accessed by a thread is fetched from the other socket. Compared to the single-socket scenario, where all data is provided from local caches, performance drops up to 7 times for very small problems.

Large problems benefit very little from a second socket, with improvements near to 10%. The advantage of the 2-socket configuration is that the available DRAM bandwidth is duplicated, and the overhead due to coherence traffic is not so important, given that most of the data is obtained from DRAM. Anyway, the small performance gain does not justify using a second socket. Again, the symmetric, scattered memory access pattern does not fit well with the NUMA hierarchy. We are currently working on a way to partition data that reduces communication between sockets.

3.8 Conclusions

In this chapter we present an optimized implementation of the DP and IDP algorithm and a novel contribution describing the first parallel version of DP and IDP.

Our implementations clearly outperform the results found in the literature. According to [Rahwan et al., 2007], they need around 2 months to solve a CSG problem with 27 agents using IDP (2.5 days in selected distributions with IP algorithm), in some unspecified computer, and using an IDP implementation which code is not provided. Our best single-thread implementation solves a same sized CSG problem in 5.8 hours. The multi-core implementation reduces execution time to 1.2 hours. Therefore, we claim that our implementation is the fast implementation of IDP published so far. We have made available to the community our source code.

We have analyzed the bottlenecks of DP and IDP. The pseudo-random memory access pattern lacks locality, and exploits the memory system capabilities very inefficiently. The latency tolerance ability of multi-threading improves performance on a multi-core processor. However, a dual-socket NUMA system is not appropriate for solving neither small nor big problems. The use of GPUs or accelerators with massive thread parallelism can obtain benefits from some of the core operations that have been described here, in fact our work has set the grounds for a GPU implementation described at [Svensson et al., 2013].

Chapter 4

Large Scale Combinatorial Auctions

4.1 Chapter Overview

In this chapter we study a variation of the Coalition Structure Generation: The Winner Determination Problem (WDP) in Combinatorial Auctions (CA). Both problems are identical in terms of their formulation, but in practice they differ in terms of scale and distribution.

While in the Coalition Structure Generation we faced a problem where all the possible combinations were evaluated, in the case of the WDP CA's there are -in the general case- only a subset of the possible combinations defined.

A standard and well-known method to solve WDP for CAs is to use search algorithms. A common practice is to improve search algorithms by the use of a heuristic and a bounding mechanism. It is also a common practice to use Linear Programming (LP) relaxations as a method to compute those bounds as well as offer support to the heuristic. Classic and yet very used algorithms to compute LP relaxations are Simplex based methods and interior-points methods (also known as barrier) [Nesterov and Nemirovskii, 1994].

We show that off-the-shelf commercial solvers implementing simplex and barrier such as IBM ILOG CPLEX [Ibm, 2011] or Gurobi [Optimization, 2017] are well suited for solving WDP for CA when the problem size is moderated (i.e. few hundreds of variables and constraints).

However, as long as the problem becomes bigger they lose potential in front of other methods. More specifically, we find convex optimization methods that are able to deliver a better response than CPLEX or GUROBI when computing an LP relaxation on a large-scale *Combinatorial Optimization* problem.

AD^3 [Martins et al., 2014] is a Convex Optimization Graph-based Algorithm that has successfully applied in the domain of Natural Language Processing. it has interesting properties, such as fast convergence to a solution and a novel way of organize the computation. It's source code is freely accessible.

The main purpose of this chapter is to demonstrate that the optimization and parallelization of AD^3 can deliver enormous benefits when solving relaxations of large-scale *Combinatorial Optimization* problems, and in particular WDPs in large-scale CAs. For solving this problem we use a simplified version of AD^3 that assumes that all the factors are factors implementing the same function. The *AtMostOne* function.

We make the following contributions:

- We show how to encode the WDP for CAs so that it can be approximated by AD^3. For this endeavour we employ the computationally-efficient factors provided by AD^3 to handle hard constraints.
- We propose an optimized, parallel implementation of AD^3, the so-called PAR-AD^3. Our implementation is based on a mechanism for distributing the computations required by AD^3 as well as on a data structure organization that together favor parallelism. PAR-AD^3 delivers an average speedup of 3X using one thread and an additional 4.8X in a 6-core computer, delivering an average speedup of 14.4X in the mentioned 6-core computer.
- We show that while AD^3 is up to 12.4 times faster than CPLEX in a single-thread execution, PAR-AD^3 is up to 23 times faster than parallel CPLEX in an 8-core architecture[1]. Therefore PAR-AD^3 becomes the algorithm of choice to solve large-scale WDP LP relaxations.
- We make our code publicly available at the following URL: `https://github.com/parad3-wdp/ca`.

To summarize, our results indicate that PAR-AD^3 obtains significant speedups on multi-core environments, hence increasing AD^3's scalability and showing its potential for application to large-scale *Combinatorial Optimization* problems in particular and for large-scale coordination problems that can be cast as *Combinatorial Optimization* problems.

The rest of the chapter is organized as follows. First, we introduce some background on AD^3. Next, we detail how to encode the WDP for CAs by

[1] The use of different architectures –6-core or 8-cores– to perform different experiment responds solely to our computational resources availability.

means of AD^3. Thereafter, we thoroughly describe $PAR\text{-}AD^3$ and afterwards we present empirical results. Finally, we draw some conclusions and set questions that will find answers in the next chapter.

4.2 Introduction

Auctions are a standard technique to solve coordination problems that has been successfully employed in a wide range of application domains [Parsons et al., 2011]. Combinatorial auctions (CAs) [Cramton et al., 2006] are a particular type of auctions that allow to allocate entire bundles of items in a single transaction. Although computationally very complex, auctioning bundles has the great advantage of eliminating the risk for a bidder of not being able to obtain complementary items at a reasonable price in a follow-up auction (think of a CA for a pair of shoes, as opposed to two consecutive single-item auctions for each of the individual shoes). CAs are expected to deliver more *efficient* allocations than non-combinatorial auctions complementarities between items hold.

CAs have been also employed to solve a variety of coordination problems (e.g. transportation [Sheffi, 2004], emergency resource coordination in disaster management [Ramchurn et al., 2008], or agent coordination in agent-driven robot navigation [Sierra et al., 2001]). However, although such application domains claim to be large-scale, namely involving thousands and even millions of bids, current results indicate that the scale of the CAs that can be optimally solved is small [Leyton-Brown et al., 2009, Ramchurn et al., 2009]. For instance, CPLEX (a state-of-the-art commercial solver) requires a median of around 3 hours to solve the integer linear program encoding the Winner Determination Problem (WDP) of a hard instance of a CA with only 1000 bids and 256 goods. This fact seriously hinders the practical applicability of current solvers to large-scale CAs.

Linear Programming (LP) relaxations are a standard method for approximating *Combinatorial Optimization* problems in computer science [Bertsimas and Tsitsiklis, 1997]. Yanover et al. [Yanover et al., 2006] report that realistic problems with a large number of variables cannot be solved by off-the-shelf, commercial LP solvers (such as CPLEX). Instead, they propose the usage of TRBP, a message-passing, dual-decomposition algorithm, to solve LP relaxations, and show that TRBP significantly outperforms CPLEX. Since then, many other message-passing and dual decomposition algorithms have been proposed to address this very same problem [Kolmogorov, 2006, Komodakis et al., 2007, Globerson and Jaakkola, 2008, Rush et al., 2010]. The advantage over other approximate algorithms is that the underlying optimization problem is well-understood and the algorithms are convergent and provide certain guarantees. Moreover, there are ways of tightening the relaxation toward the exact solution [Sontag et al., 2012].

In order to solve LP relaxations, there has been a recent upsurge of interest in the Alternating Direction Method of Multipliers (ADMM), which was invented in the 1970s by Glowinski and Marroco [Glowinski and Marroco, 1975] and Gabay and Mercier [Gabay and Mercier, 1976]. As discussed in [Boyd et al., 2011], ADMM is specially well suited for application in a wide variety of large-scale distributed modern problems. Along this line, Martins has proposed AD^3 [Martins et al., 2014], a novel algorithm based on ADMM, which proves to outperform off-the-shelf, commercial LP solvers for problems including declarative constraints. AD^3 has the same modular architecture of previous dual decomposition algorithms, but it is faster to reach consensus, and it is suitable for embedding in a Branch-and-bound procedure toward the optimal solution. Martins derives efficient procedures for handling logic factors and a general procedure for dealing with dense, large, or combinatorial factors. Notice that until [Martins, 2012b], the handling of declarative constraints by message-passing algorithms was barely addressed, and not well understood. This hindered their application to combinatorial auction WDPs, which typically require this type of constraints. Therefore, AD^3 constitutes a promising tool to solve WDPs in CAs.

As discussed in [Martins, 2012b] (see section 7.5), AD^3 is largely amenable to parallelization, since AD^3 separates an optimization problem into sub-problems that can be solved in parallel. Nonetheless, to the best of our knowledge there is no parallel implementation of AD^3. Therefore, the potential speedups that AD^3 may obtain when running on multi-core environments remain unexplored. And yet, this path of research is encouraged by recent experiences in parallelization of ADMM applied to solve an unconstrained optimization problem [Miksik et al., 2014]. Indeed, Miksik et al. show that a parallel implementation of ADMM delivers large speedups for large-scale problems. Notice though that the work in [Miksik et al., 2014] cannot be employed to solve the WDP for CAs because it cannot handle hard constraints.

4.3 Background

Graphical models are widely used in computer vision, natural language processing and computational biology, where a fundamental problem is to find the maximum a posteriori probability (MAP) given a Factor Graph. Since finding the exact MAP is frequently an intractable problem, significant research has been carried out to develop algorithms that approximate the MAP.

Linear Programming (LP) relaxations have been extensively applied to approximate the MAP for graphical models since [Santos Jr, 1991]. Typically, such application domains lead to sparse problems with a large number of variables and constraints (i.e beyond 10^4). As shown in [Yanover et al., 2006], message passing algorithms have been proved to outperform state-of-the-art commercial LP solvers (such as e.g. CPLEX) when approximating the MAP for large-scale problems. This advantage stems from the fact that message-passing algorithms better exploit the underlying graph structure representing the problem.

Along this direction, several message passing algorithms have been proposed in the literature: ADMM [Eckstein and Bertsekas, 1992], TRBP [Wainwright et al., 2003], MPLP [Globerson and Jaakkola, 2008], PSDD [Komodakis et al., 2007], Norm-Product BP [Hazan and Shashua, 2010], and more recently Alternate Direction Dual Decomposition (AD^3) [Aguiar et al., 2011].

As discussed in [Martins et al., 2014], the recently-proposed AD^3 has some very interesting features in front of other message passing algorithms: it reaches consensus faster than other algorithms such as ADMM, TRBP and PSDD; it does have neither the convergence problems of MPLP nor the instability problems of Norm-Product BP; and its anytime design allows to stop the optimization process whenever a pre-specified accuracy is reached. Furthermore, as reported in [Martins et al., 2014], AD^3 has been empirically shown to outperform state-of-the-art message passing algorithms on large-scale problems.

Besides these features, AD^3 also provides a library of computationally-efficient factors that allow to handle declarative constraints within an optimization problem. This opens the possibility of employing AD^3 to approximate constrained optimization problems.

Algorithm 4 outlines the main operations performed by AD^3 on a Factor Graph G with a set of factors F, a set of variables V, and a set of edges $E \subseteq F \times V$. AD^3 receives a set of parameters θ that encode variable coefficients and a penalty constant η able to regulate the update step size. We use the function $\partial(x)$ to denote all the neighbours (i.e. connected nodes) of a given graph node. The primal variables q and p, the dual λ as well as the unary log-potentials ξ are vectors which are updated during the execution .We refer the reader to [Martins et al., 2014] for a detailed description of the algorithm. AD^3 is an iterative three-step algorithm designed to approximate an objective function encoded as a Factor Graph. A key aspect of AD^3 is that it separates the optimization problem into independent subproblems that progress to reach consensus on the values to assign to primal and dual variables. Thus, during the first step, *broadcast*, the optimization problem is split into separate subproblems, each one being distributed to a factor. Thereafter, each factor locally solves its local subproblem. In AD^3, this computations is carried on solving a quadratic problem. During the second step, *gather*, each variable gathers the subproblems' solutions of the factors it is linked to. Finally, during the third step, *Lagrange updates*, the Lagrange multipliers for each subproblem are updated.

Algorithm 4 Alternating Directions Dual Decomposition(AD^3)

```
input: Factor Graph $G$, parameters $\theta$, penalty constant $\eta$
initialize $p$ (i.e. $p_i = 0.5 \forall i \in 1 \ldots |V|$), initialize $\lambda = 0$
repeat                                                          ▷ Broadcast
    for each factor $\alpha \in F$ do
        for each $i \in \partial(\alpha)$ do
            set unary log-potentials $\xi_{i\alpha} := \theta_{i\alpha} + \lambda_{i\alpha}$
        end for
        $\hat{q}_\alpha :=$ SolveQP$(\theta_\alpha + \xi_\alpha, (\mathbf{p}_i)_{i \in \partial(\alpha)})$
    end for
    for each variable $i \in V$ do                              ▷ Gather
        compute avg $p_i := |\partial(i)|^{-1} \sum_{\alpha \in \partial(i)} \hat{q}_{i\alpha}$
        for each $\alpha \in \partial(i)$ do                    ▷ Lagrange updates
            $\lambda_{i\alpha} := \lambda_{i\alpha} - \eta(\hat{q}_{i\alpha} - p_i)$
        end for
    end for
until convergence
```

A distinguishing feature of AD^3 is that both the broadcast and update steps can be safely run in parallel. Indeed, notice that, since subproblems are independent, they can be safely distributed in different factors so that each one independently computes a local solution. AD^3 provides a collection of factors for which their quadratic problems are defined. As an example we present how the quadratic problem for the XOR factor is solved in Algorithm 5, where the input of the algorithm are the potentials $Z_\alpha : z_0, \ldots, z_K$ relative to the factor α. Note that in Algorithm 4 the call to the SolveQP method has two parameters, the second parameter is omitted here since it is not needed to solve the XOR. Algorithm 5 proceeds as follows. Lines 11-13 are responsible of checking if the constraint XOR is already satisfied. Then, if not satisfied, the Z_α vector is transformed using the projection onto simplex method described by [Duchi et al., 2008]. This method navigates through Z_α in decreasing order, to find the pivot element y_i and the value of τ. Afterwards this τ is used to perform the actual projection. To this end, two auxiliary vectors Z'_α and Y_α are used: the former will contain the algorithm output and the latter is used to contain a sorted copy of Z_α. Although there are ways to obtain the pivot without the need of sorting the vector Z_α (described in [Duchi et al., 2008]), in AD^3 is preferable to have a persistent sorted vector since order of elements is commonly preserved or barely altered across the iterations. Therefore efficient sorting methods on nearly-ordered sequences can be applied. An important feature of the XOR factor is that its quadratic problem can be solved in $O(K \cdot logK)$, where K stands for the number of variables connected to the factor.

As to gather, the step in which the subproblems communicate their local results, each variable can independently (from the rest of variables) gather and aggregate the results computed by the factors it is linked to. Despite being highly prone to parallelisation, to the best of our knowledge there is only one public

Algorithm 5 SolveQP for an XOR factor

```
input: $Z_\alpha : z_0, \ldots, z_K$, vector with $\alpha$ log-potentials
function FindTau($Y_\alpha$)
    $\tau = 0.0$;
    $sum := \sum_{y_i \in Y_\alpha} y_i$
    for each $y_i \in Y_\alpha$ do
        $\tau := \frac{sum-1}{K-i}$
        if $y_i > \tau$ then break
        update $sum := sum - y_i$
    end for
    return $\tau$
end function
$z'_i := max(0, z_i)$, for each $z_i \in Z_\alpha$
$sum := \sum_{z'_i \in Z'_\alpha} z'_i$
if $sum > 1.0$ then                                   ▷ Projection onto simplex
    sort $Z_\alpha$ into $Y_\alpha$: $y_0 \leq \ldots \leq y_K$
    $\tau :=$ FindTau($Y_\alpha$)
    $z'_i :=$ max($z_i - \tau$,0) , for each $z_i \in Z_\alpha$
end if
output: $Z'_\alpha$
```

implementation of AD^3 and cannot run in parallel [2]. The recent contributions to the parallelisation of ADMM to solve unconstrained optimisation problems [Miksik et al., 2014] are very encouraging because they show that it is possible to obtain very significant speedups by exploiting nowadays parallel hardware. This finding spurs and motivates the need for a parallel implementation of AD^3.

But before that, in the next section we show that the WDP for CAs can be solved by means of AD^3.

4.4 Solving Combinatorial Auctions with AD^3

A Combinatorial Auction (CA) is an auction in which bidders can place bids for a combination of items instead of individual ones. In this scenario, one of the fundamental problems is the Winner Determination Problem (WDP), which consists in finding the set of bids that maximise the auctioneer's benefit. Notice that the WDP is an $\mathcal{NP}$-complete problem.

Although special-purpose algorithms have addressed the WDP (e.g. [Fujishima et al., 1999, Sandholm et al., 2001]), the state-of-the-art method for solving a WDP is to encode it as an integer linear program (ILP) and solve it

[2] Available at http://www.ark.cs.cmu.edu/AD3/

using an off-the-shelf commercial solver (such as CPLEX [Ibm, 2011] or Gurobi [Optimization, 2017]). Nonetheless, this approach fails to scale to large CA instances. Indeed, as noticed in [Sheffi, 2004], real problems may involve up to millions of bids. Therefore, such real problems are out of reach for state-of-the-art optimal solvers, and hence the need for heuristic approaches arise.

As observed in [Ball, 2011], "The simplest and, perhaps most tempting approach, to an optimization-based heuristic is to round the solution to a linear programming relaxation". Furthermore, solutions to an LP relaxation can provide a very effective start to finding a good feasible solution to the non-relaxed optimisation problem. Hereafter we focus on solving the LP relaxation of the WDP by means of AD^3. Since AD^3 requires a Factor Graph to operate, we first show how to encode the WDP as a Factor Graph. Then we show how AD^3 can run on top of this Factor Graph. We shall start by showing such encoding by means of an example to finally derive a general procedure.

Consider an auctioneer puts on sale a pair of goods g_1, g_2. Say that the auctioneer receives the following bids: b_1 offering \$20 for g_1 ; b_2 offering \$10 for g_2; and finally b_3 offering \$35 for goods g_1 and g_2 together. The WDP for this CA can be encoded as the following ILP:

$$\begin{array}{lll}
\text{maximise} & 20 \cdot x_1 + 10 \cdot x_2 + 35 \cdot x_3 & \\
\text{subject to} & x_1 + x_3 \leq 1 & [\textit{constraint } c_1] \\
 & x_2 + x_3 \leq 1 & [\textit{constraint } c_2] \\
 & x_1, x_2, x_3 \in \{0, 1\} &
\end{array}$$

where x_1, x_2, and x_3 stand for binary decision variables that indicate whether each bid is selected or not; constraint c_1 expresses that good g_1 can only be allocated to either bid b_1 or bid b_3 and constraint c_2 encodes that good g_2 can only be allocated to either bid b_2 or bid b_3.

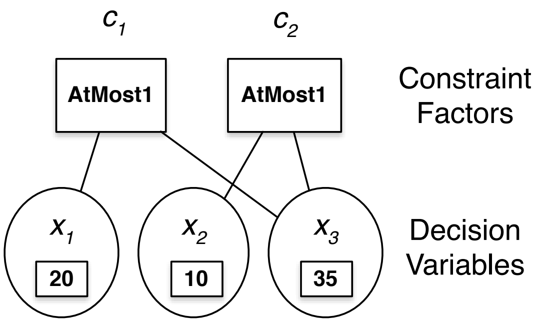


Figure 4.1: Factor Graph encoding of our CA example.

Now we can encode the optimisation problem above into a Factor Graph as illustrated in Figure 4.1. First, we create a variable node for each bid. Each variable contains its bid's offer (indicates the value that the auctioneer obtains when the variable is active). For instance, variable x_1 for bid b_1 contains value 20. Then we create a factor node per good, connecting the bids that compete for the good, and which are therefore incompatible. For instance, factor c_1 is linked to the variables corresponding to bids b_1 and b_3.

We observe that each factor representing a constraint in the Factor Graph in figure 4.1 corresponds to the "AtMost1" function introduced by Smith and Eisner [Smith and Eisner, 2008], which is satisfied if there is at most one active input. Although AD^3 does not directly support "AtMost1" constraints, as seen in [Martins, 2012b], an XOR factor can be used to define it by adding a slack variable to the factor. The XOR factor complexity is $O(K \cdot logK)$, where K stands for the number of variables connected to the XOR factor. Notice that the operation of AD^3 when solving the WDP only involves computationally-efficient factors.

4.5 Parallel Realization of AD^3

The AD^3 algorithm is amenable to general, architecture-level optimization and parallelization [Martins, 2012b]. We propose an efficient realization of the message-passing[3] algorithmic pattern using shared variables and targeting multi-core computer architectures. The so-called *PAR-*AD^3, that exploits the inherent parallelism at two dimensions: thread-level and data-level. For that, we reorganize both the data structures layout and the order of operations. The approach is generalizable to other similar graph processing algorithms. The key insights of our design are:

- An edge-centric representation of the shared variables that improves memory access performance.
- A reorganization of the operations that promotes parallel scaling (thread parallelism) and vectorizing (data parallelism).

4.5.1 Edge-centric Shared Data Layout

AD^3 is a message passing algorithm that iterates on three steps: *broadcast*, *gather* and *Lagrange multiplier update*. The message passing pattern isolates the operations applied to the different elements of the graph (factors, variables and edges), so that multiple operations can be performed concurrently on the

[3] Recall that message-passing algorithm here stands for a classification of the algorithm in graphical models and not for the use of passing messages between CPU threads or processes.

graph data. These operations and data can then be physically distributed along different computation and storage elements.

The memory requirements of AD^3 are approximately proportional to the number of edges, and, for the problem sizes considered, they are fulfilled by most current shared-memory computer systems. In this situation, the fastest and most efficient mechanism for communication and synchronization between processing cores is using shared variables (instead of explicit messages). The different processing cores of the computer will operate concurrently on the different elements of the graph (factors, variables or edges), both reading input data and generating new results stored in the shared memory. Execution performance is improved with a careful selection of synchronization operations at the right point and an appropriate data structures layout.

Memory access performance is very sensitive to the data layout and data access pattern. When a loop has to iterate along a large regular data structure, the best performance is achieved when the next elements of the structure are naturally fetched from the next memory positions at each step of the iteration. Since AD^3 demands more computation work operating in edge data than in vertex or factor data, we adopt an edge-centric data representation, as reported in [Roy et al., 2013]. We want all information related to edges, such as unary log-potentials or lagrangian components, to be stored in consecutive memory positions. With this purpose, we apply a memory layout transformation that converts data structures originally designed in an Array of Structures (AoS) representation to a Structure of Arrays (SoA) representation.

Figure 4.2 illustrates how data was stored in memory in AD^3 and how the data layout is modified in $PAR\text{-}AD^3$. For the sake of clarity, we present data regarding 2 variables and 4 edges. AD^3 encodes the information following an AoS representation, where all properties related to each variable or edge are stored consecutively (see figure 4.2a).

As the AD^3 design is variable-centric, iterating on all the edges in the graph requires an indirect and scattered access to the variables (edges are accessed using the pointers associated to each variable). In contrast, the $PAR\text{-}AD^3$ SoA memory layout (figure 4.2b) stores the properties of variables and edges sequentially, thus resulting in a different array for each edge or variable property. Now, iterating on all edges of the graph requires consecutive accesses to array elements. The AOS memory representation of AD^3 benefits from memory access patterns where all the variable properties are used together, meanwhile the SoA memory representation of $PAR\text{-}AD^3$ benefits from the access of any property traversing all variables or edges.

To summarize, the $PAR\text{-}AD^3$ data representation transforms many scattered memory accesses into sequential, improving the memory access throughput. A derived advantage of the simplified edge access pattern is to foster better parallel

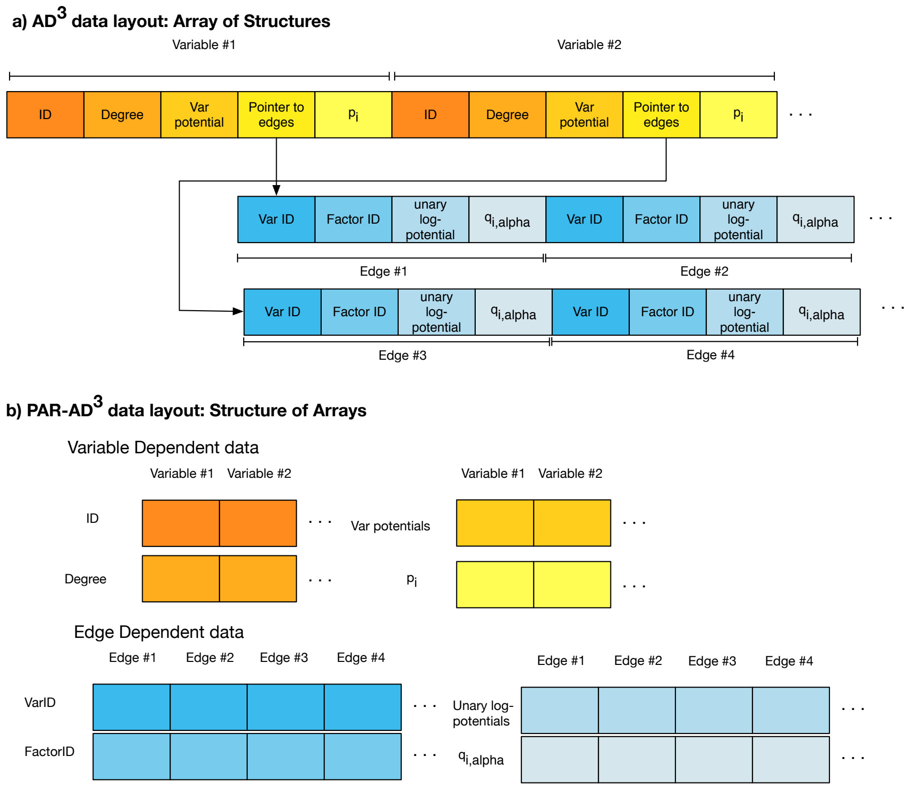


Figure 4.2: a) AoS data representation of AD^3, compared to b) SoA data representation of $PAR\text{-}AD^3$.

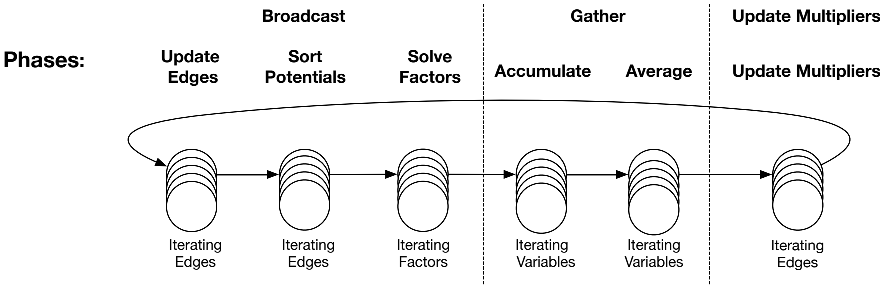


Figure 4.3: Processing phases and parallelism in $PAR\text{-}AD^3$.

scaling and vectorization, but we need additional algorithmic transformations that are described in the next section.

4.5.2 Reordering Operations

Parallel scaling means distributing compute operations on large chunks of data along different computational units sharing the same memory space. Vectorizing applies data parallelism strategies inside the same computational unit, and consists in using instructions that operate simultaneously on a small vector of consecutive data elements. Both parallel scaling and vectorizing are usually applied to simple loop iterations with clearly separated inputs and outputs, no recurrent dependencies, and sequential accesses to vector elements.

Our proposal reshapes the way the algorithm defines the graph operations towards a new structure of many simple consecutive loops, outlined in figure 4.3. The original *Broadcast* step is now split in three phases: *update edge*, *sort potential* and *solve factors*. Also, the original *Gather* step is now split in two phases: *accumulate* and *average*. Note that we iterate on factors twice and also iterate on variables twice: this makes the loops simpler and provides more data locality. As a result, all phases are now parallelized for concurrent execution (thread parallelism) and four out of six are vectorized: *update edge*, *accumulate*, *average* and *update multiplier*.

Algorithm 6 shows a pseudo-code of $PAR\text{-}AD^3$ as a result of the optimizations applied. A pool of parallel threads is created outside of the main loop (line 2). Whenever a parallel loop inside the main loop is reached (lines 4, 9, 12, 19, 24, 27), the loop iterations are distributed to the threads for parallel execution. There is an implicit synchronization after each loop, so that all threads wait for the generation of the results in one loop before starting the execution of the next.

As thoroughly described in the next section, these contributions have a significant impact in the sequential execution as well as allow good parallel scalability when an increasingly large number of threads are used.

It is important to underline that this code reorganization takes benefit from considering that all the factors are solving the same Quadratic Problem (QP), hence some computations that belong to the local subproblem are interleaved in other phases of $PAR - AD^3$. In other words, we are sacrificing encapsulation to in exchange gain extra performance. We can see examples of this technique looking at Algorithm 5 where lines 11 and 14 (accumulate and sorting) are no longer execued inside every factor but they are precalculated in advance (see Algorithm 6 lines 7, 9-11). This way we en having a lightweight QP that most of the time performs an addition and exit. The precalculated operations are run in a bunch from a parallelized graph-wise perspective.

Algorithm 6 *PAR-AD*^{3} pseudo-code

```
input: Factor Graph $G$, parameters $\theta$, penalty constant $\eta$
initialize $p$ (i.e. $p_i = 0.5 \forall i \in 1 \ldots |V|$), initialize $\lambda = 0$
create threads
repeat
    parallel for $i\alpha \in E$ do                                  ▷ Update edges
        Update log-potentials $\xi_{i\alpha} := \theta_{i\alpha} + \lambda_{i\alpha}$
        compute $\hat{q}_{i\alpha} = \theta_{i\alpha} + \xi_{i\alpha}$
        compute $\hat{q}'_{i\alpha} = max(0, \hat{q}_{i\alpha})$
    end for
    parallel for factor $\alpha \in F$ do                            ▷ Sort potentials
        $\hat{q}\_sorted_{\alpha}$ := sort($\hat{q}_{\alpha}$)
    end for
    parallel for factor $\alpha \in F$ do                            ▷ Solve factors
        $sum = \sum_{i \in \partial(\alpha)} (\hat{q}'_{i\alpha})$
        if $sum > 1.0$ then
            $\tau$ := FindTau($\hat{q}\_sorted_{\alpha}$)
            $q'_{i\alpha}$ := max($q_{i\alpha} - \tau$,0), for each $q_{i\alpha} \in q_{\alpha}$
        end if
    end for
    parallel for  variable $i \in V$ do                              ▷ Accumulate
        for $i \in \partial(\alpha)$ do
            $\tilde{p}_i := \tilde{p}_i + \hat{q}_{i\alpha}$
        end for
    end for
    parallel for variable $i \in V$ do                               ▷ Average
        $p_i := \tilde{p}_i / |\partial(i)|$
    end for
    parallel for $i\alpha \in E$ do
        $\lambda_{i\alpha} := \lambda_{i\alpha} - \eta(\hat{q}_{i\alpha} - p_i)$         ▷ Update multipliers
    end for
    update $\eta$
until convergence
output: primal variables $p$ and $q$, dual variable $\lambda$
```

Since a clear trend in computer architecture is an increase of parallelism both at instruction and thread level, (for example, the intel Xeon Phi accelerator operates with 512-bit vector registers and contains more than 60 execution cores) the methodology applied to $PAR\text{-}AD^3$ makes it ready to benefit from upcoming improvements.

4.6 Empirical Evaluation

In this section, we assess $PAR\text{-}AD^3$ performance against the original AD^3 algorithm to evaluate our performance improvements as well as to the state-of-the-art optimisation software CPLEX with the aim of determining the scenarios for which $PAR\text{-}AD^3$ is the algorithm of choice. We also quantify its current gains, both in sequential and parallel executions. To this end, we first find the data distributions and range of problems that are best suited for $PAR\text{-}AD^3$. Thereafter, we briefly analyse two algorithmic key features: convergence and solution quality. Afterwards, we quantify the speedups of $PAR\text{-}AD^3$ with respect to AD^3 and CPLEX in sequential and parallel executions. From this analysis we conclude that $PAR\text{-}AD^3$ significant speedup with respect to AD^3 (14.4X in a 6-core computer). It also obtains larger benefits from parallelisation than CPLEX. Indeed, $PAR\text{-}AD^3$ achieves a peak speedup of 23X above CPLEX barrier, the state-of-the-art solver for sparse problems.

4.6.1 Experiment Setup

In order to generate CA WDP instances, we employ CATS, the CA generator suite described in [Leyton-Brown et al., 2000]. Each instance is generated out of the following list of distributions thoroughly described in [Leyton-Brown et al., 2009]: arbitrary, matching, paths, regions, scheduling, L1, L3, L4, L5, L6 and L7. We discarded to employ the L2 distribution, because the CATS generator is not capable of generating large instances. While the first five distributions were designed to generate realistic CA WDP instances, the latter ones generate artificial instances. The main difference between the two distribution categories is the use of dummy goods that add structure to the problem inspired in some real life scenarios. i.e. Paths models the transportation links between cities; Regions models an auction of real estate or an auction where the basis of complementarity is the two-dimensional adjacency of goods; Arbitrary extends regions by removing the two-dimensional adjacency assumption, and it can be applied to model electronic parts design or procurement; Matching models airline take-off and landing rights auctions; and Scheduling models a distributed job-shop scheduling domain. Artificial (or Legacy) distributions have been often criticized [Andersson et al., 2000, Leyton-Brown et al., 2000, De Vries and Vohra, 2003] mainly due to their poor applicability, specially in the economic field. However they are interesting in order to study the algorithm performance in different situations. Both AD^3 and $PAR\text{-}AD^3$ are well suited for large-scale hard problems. For this reason, we first determine which of these distributions are hard to solve, putting special attention to the realistic ones. For our experimentation, we considered a number of goods within $[10^3, 10^4]$ in steps of 10^3 goods. Furthermore, the number of bids ranged within $[10^4, 4 \cdot 10^4]$ in steps of 10^4 bids. Each problem scenario is characterized by a combination of distribution, number of goods, and number of bids. Our experiments consider 5 different instances for each problem scenario and we analyze their mean value. Experiments are executed in two different hardware architectures.

- Experiments comparing AD^3 against $PAR\text{-}AD^3$ were run in a computer with a 6-core Intel(R) Core(TM) i7-7500U processor with 32 GB RAM and the hyper-threading mechanism enabled.
- Experiments comparing $PAR\text{-}AD^3$ against CPLEX were run in a computer with two four-core Intel (R) Xeon(TM) L5520 processors with 32 GB RAM and the hyper-threading mechanism disabled.

While the former experiments comparing AD^3 against $PAR\text{-}AD^3$ were run in a local environment, the latter experiments were run in a cluster licensed with CPLEX. Unfortunately we were not able to reproduce the former experimentation of AD^3 against $PAR\text{-}AD^3$ in the cluster due to hardware availability issues.

4.6.2 Different Distributions Hardness

We empirically determine the hardness of the relaxation for our experimental data by solving the LP using CPLEX simplex (simplex henceforth), CPLEX barrier (barrier henceforth), the state-of-the art algorithms. Results are plot in figures 4.4a and 4.4b. According to the results, scheduling and matching from the realistic distributions and L1, L4 from the legacy ones are very well addressed by simplex, where solving time is, in general, less than one second. Both AD^3 and $PAR\text{-}AD^3$ are not competitive in this scenario. Applicability of $PAR\text{-}AD^3$ will be shown to be effective to the rest of distributions, especially in hard instances. Barrier is also doing a good job when the problems are hard, particularly in the arbitrary and regions distributions, where the problem representation matrix is more sparse.

4.6.3 Single-Thread Analysis

After comparing the publicly-available version of AD^3 against sequential $PAR\text{-}AD^3$, we observed that $PAR\text{-}AD^3$ outperformed AD^3 even in sequential execu-

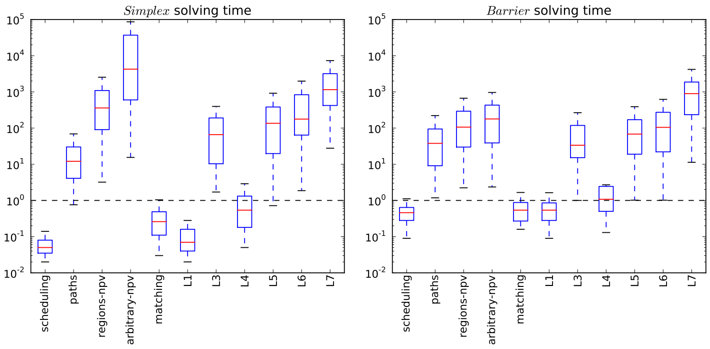


Figure 4.4: Solving time in seconds for different distributions, single thread. a) Simplex b) Barrier

tion, reaching an average speedup of 3X and a peak speedup of 6.4X. Moreover, we observed that the harder the instances, the larger the speedups of $PAR\text{-}AD^3$ with respect to AD^3. Since both algorithms are well suited for hard instances, this is particularly noticeable. Next, we compared the single-thread average performance of $PAR\text{-}AD^3$ against simplex and barrier. The results are plot in Figure 4.5,where we display the best algorithm for the different distributions and problem sizes. $PAR\text{-}AD^3$ is shown to be well suited for larger problems (the upper-right corner) in almost all the distributions. In general, barrier is the best algorithm in the mid-sized problems, while simplex applicability is limited to a small number of cases. Distribution paths presents a different behaviour, where adding goods increases the average bid arity and this is beneficial for simplex, which runs better in dense problems.

In general, the larger the WDP instances, the larger the $PAR\text{-}AD^3$ benefits. Single-threaded $PAR\text{-}AD^3$ reaches a peak speedup of 12.4 for the hardest distribution when compared to barrier, the best of the two state-of-the-art solvers.

4.6.4 Convergence and Solution Quality

Figure 4.7 shows a trace of an execution that illustrates the way the different solvers approximate the solution over time (using a regions distribution, 5×10^3 goods, and 10^4 bids). We chose this run because the similar performance of the three algorithms made them comparable. Note that $PAR\text{-}AD^3$ converges to the solution in 29 sec. , while barrier requires 102 sec. and simplex 202 sec. (not visible in the figure).Furthermore, notice that $PAR\text{-}AD^3$ quickly reaches a high-quality bound, hence promptly guaranteeing close-to-the-solution anytime approximations. In general, our experimental data indicate that the initial solution provided by $PAR\text{-}AD^3$ is always significantly better than the one assessed by both simplex and barrier. Finally, upon convergence, there is a maximum

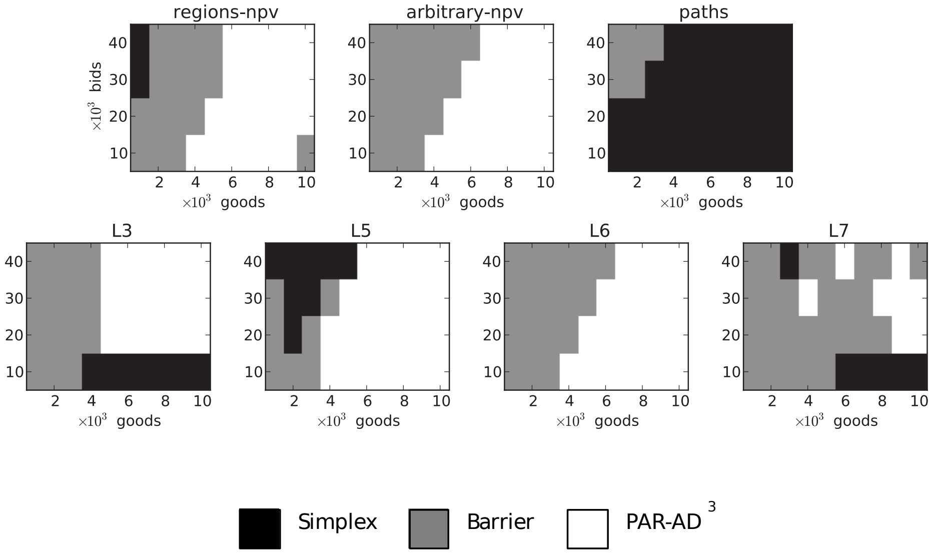


Figure 4.5: Fastest single-thread algorithm solving different distributions and problem sizes.

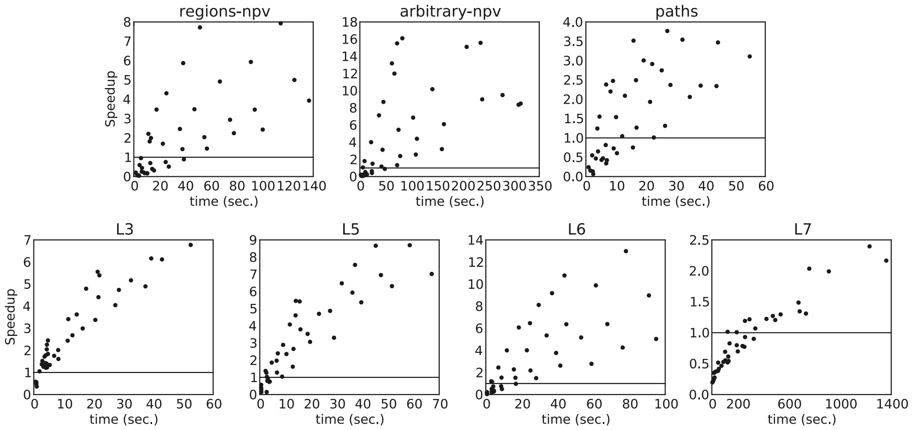


Figure 4.6: Speedup of $PAR\text{-}AD^3$ for different distributions against barrier in a multi-thread execution

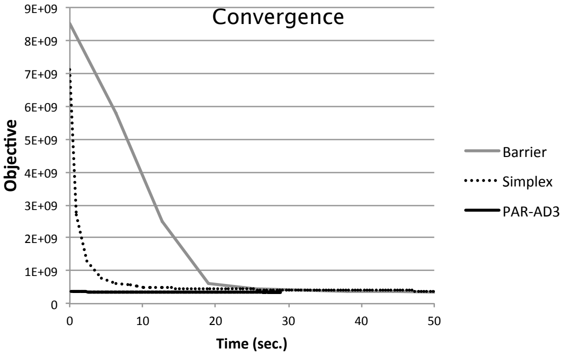


Figure 4.7: Convergence of simplex, barrier and $PAR\text{-}AD^3$

deviation of 0.02% between $PAR\text{-}AD^3$ solutions and those assessed by CPLEX. Note that we run CPLEX with default parameters, has the feasibility tolerance set to 10^{-6}. This means that CPLEX solutions may be infeasible up to a range of 10^{-6} per variable. In the same sense, $PAR\text{-}AD^3$ feasibility tolerance is set to 10^{-12}. This good initial solution is a nice property that makes $PAR\text{-}AD^3$ suitable to be used as a method able to obtain quick bounds, either to be embedded in a MIP solver or also to provide a fast solution able to be used towards an approximate solution.

4.6.5 Multi-Thread Analysis

$PAR\text{-}AD^3$ obtained an average parallelization speedup of 4.8X in a 6-core computer, delivering an aggregated speedup of 14.4X with respect to AD^3.

We also have run $PAR\text{-}AD^3$, simplex and barrier with 8 parallel threads each, hence using the full parallelism offered by our computer. The results are displayed in figure 4.8. When comparing with figure 4.5 (corresponding to the single-thread execution), we observe that $PAR\text{-}AD^3$ outperforms simplex and barrier in many more scenarios, and in general $PAR\text{-}AD^3$ applicability grows in concert with the parallel resources in all cases. Hence, we infer that $PAR\text{-}AD^3$ better benefits from parallelisation than simplex and barrier. The case of the paths distribution is especially remarkable since simplex is faster than other algorithms when running in a single-thread scenario. Nonetheless, as $PAR\text{-}AD^3$ better exploits parallelism, it revealed to be the most suitable algorithm for hard distributions when running in multi-threaded executions, including paths. In accordance with those results, it is expected that increasing the number of computational units will widen the range of applicability of $PAR\text{-}AD^3$.

Finally, we compared $PAR\text{-}AD^3$ performance against barrier using 8 threads. We only compare $PAR\text{-}AD^3$ to barrier since it is the best suited algorithm for the selected distributions (i.e in some executions $PAR\text{-}AD^3$ can be up to three orders

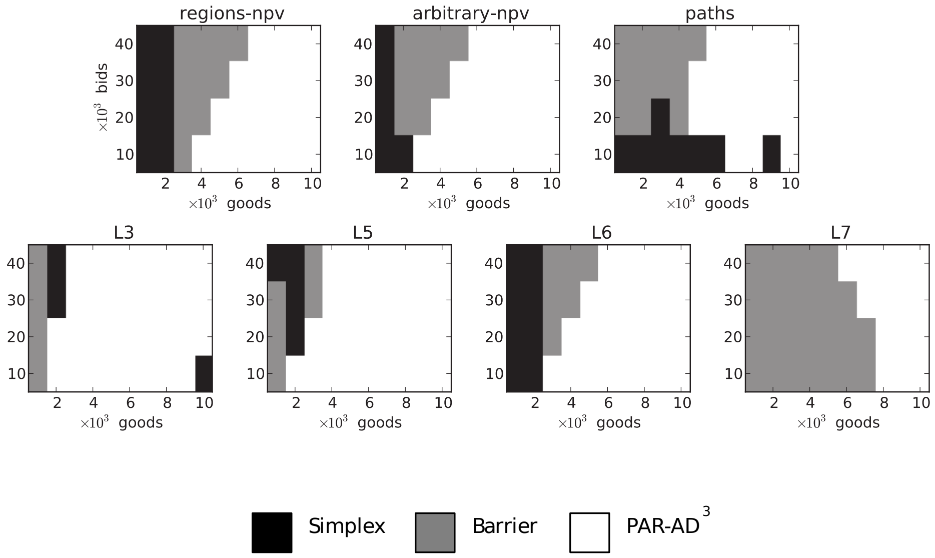


Figure 4.8: Fastest algorithm solving different distributions and problem sizes using multiple threads.

of magnitude faster than simplex). Figure 4.6 shows the average performance speedup of *PAR-AD*^{3} versus barrier as a function of the total running time of the execution of barrier (shown in the X-axis). We observe a clear trend in all scenarios: the harder the problem becomes for barrier, the larger the speedups obtained by *PAR-AD*^{3}. Our peak speedup is 23X (16X when taking the mean execution time of the different instances). The best results are achieved in the arbitrary distribution, which in addition was significantly better solved by barrier than by simplex according to figure 4.4. We recall that arbitrary is a distribution that can be applied to the design of electronic parts or procurement since it removes the two-dimensional adjacency of regions. In arbitrary, larger speedups correspond to the more sparse scenario, i.e. the bottom-right corner in figure 4.8.

4.7 Conclusions

In this chapter we have opened up a path towards solving large-scale CAs. We have proposed a novel approach to solve the LP relaxation for the WDP. Our approach encodes the optimization problem as a Factor Graph and uses AD^3, a dual-decomposition message-passing algorithm, to efficiently find the solution.

In order to achieve higher efficiency, we rearranged the operations performed by AD^3 providing a new algorithm, the so-called *PAR-AD*^{3}, which is an optimized and parallel version of AD^3. *PAR-*AD^3 delivers an average speedup of 14.4X in a 6-core computer with respect to AD^3.

Our experimental results validate *PAR-*AD^3 efficiency gains in large scale scenarios. We have shown that *PAR-*AD^3 performs better than CPLEX for large-scale CAs in the computationally hardest distributions, both in single- and multi-threaded scenarios, with a peak speedup of 23X. Furthermore, the speedup is larger in multi-threaded scenarios, showing that *PAR-*AD^3 scales better with hardware than CPLEX. Therefore, *PAR-*AD^3 has much potential to solve large-scale coordination problems that can be cast as optimization problems.

However the described version of *PAR-*AD^3 for WDPs for CA is limited to solve the LP relaxations for those problems whose constraints can be represented by constraints of the type *AtMostOne*, i.e. all the problems that can be modeled as a 0-1-knapsack problem. In the next chapter we show how to obtain a bigger applicability by extending the kind of Quadratic Problems that the algorithm will be able to compute.

Chapter 5

Side Chain Prediction

5.1 Chapter Overview

In the previous chapter we presented a parallel and efficient design of AD^3, the so-called PAR-AD^3 and applied it to approximate LP relaxations for the Winner Determination Problem (WDP) in Combinatorial Auctions (CA). In the case of the WDP for CA, it was possible to encode the problem considering only one type of constraint –or factor– known as *AtMostOne*. Implementing a method to solve the *AtMostOne* factor was enough to solve the problem by means of PAR-AD^3. Moreover, *AtMostOne* was shown to carry very lightweight computation. Hence, in the previous chapter, the computational load was centered in the computation happening outside the factor. Our work was focused in the design of a parallel paradigm and we studied how to represent the data and the organization of the computation scheduling. In other words, we were designing how to orchestrate in parallel a set of thousands factors of a Factor Graph to be solved by PAR-AD^3. In the present chapter we still work with PAR-AD^3, but this time we tackle a problem that requires a computational-intensive constraint to be solved. While we get benefit of the parallelization as was designed in the previous chapter, in this chapter we will focus in the optimization of computationally intense factor: The *Arbitrary* factor. This new factor allow the representation of any discrete function, widening the applicability of PAR-AD^3 substantially.

This chapter presents how to approximate results for LP relaxations for a problem belonging to the domain of Biology: the Side-Chain Prediction problem. Our work follow the steps of the contributions of Chen Yanover et al in their studies with the belief propagation algorithm [Yanover et al., 2006], where they propose Tree Reweighted Belief Propagation to approximate LP relaxations for the Side-Chain Prediction problem. They also build a dataset optimization problems for Side-Chain Prediction that is freely downloadable.

We make the following contributions:

- We propose an optimized version of the method to solve *Arbitrary* factors. This optimized version can be embedded into *PAR-AD*^{3}. Our *PAR-AD*^{3} reaches an average speedup of 2.7X in a single-threaded execution and 10.8X using our 6-core computer with respect to AD^3.
- We contribute with the generation of a *Combinatorial Optimization* dataset based on Side-Chain Prediction problems.
- We present a detailed performance analysis where the we show the performance limiters of our *PAR-AD*^{3} both in sequential and parallel.
- We release our code and materials to be freely accessible at `https://github.com/parad3-scp/source`.

The chapter is organized as follows:

The chapter starts with an introduction to the Side-Chain Prediction problem. We provide an overview of the key concepts needed to understand the rationale behind this Problem.

Then we show how to encode a problem of Side-Chain Prediction as a Factor Graph in order to be solved by means of AD^3.

Next we use a Side-Chain Prediction-based dataset proposed by [Yanover et al., 2006] in order to measure the performance of AD^3 algorithm when solving the Side-Chain Prediction. We show that more than the 90% of the problems in the dataset do not suppose a big challenge for AD^3. Most of them are solved in less than one second and also in the majority of cases AD^3 is able to find the optimal solution. In order to have a highest computational challenge, we build a new Side-Chain Prediction-based dataset. The new dataset is based in the Side-Chain Prediction problem for larger proteins than the ones present in the Yanover dataset.

Then we study how to optimize the aforementioned *Arbitrary* factors from a computational performance perspective, and present our optimized and parallel algorithm with support to *Arbitrary* factors.

The chapter ends with a description of the techniques we used to improve algorithm performance in a single-thread execution. Finally we present results in both in single core and multicore and examine in detail the limitations of our new proposed algorithm.

5.2 The Biological Perspective

5.2.1 Background

Proteins are large biomolecules consisting of one or more chains of amino-acids. Proteins are a fundamental component for every living organism. They are specially interesting because they can perform a broad range of biological functions. They are also very diverse yet structurally very simple. Molecular properties such as shape, polarity, hydro-affinity of a protein are determined in a great extent by the sequence of amino-acids conforming it.

Amino-acids are small and simple molecules. All the amino-acids share the same structure, shown in figure 5.1. They are formed by an amina group (-NH3), a carbon with a residue (R) attached, and a carboxyl group (-COOH). The residue (R) also known as side-chain, is what characterizes an amino-acid. There exist up to 21 possible residues or side-chains giving rise to the 21 existent amino-acids. This 21 amino-acids constitute the bricks able to build proteins.

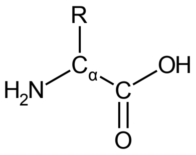


Figure 5.1: Structure of an amino-acid.

A protein has two structural elements: The backbone and the side-chains. The backbone is the large chain of amino-acids connected that define the structure of the protein, and the side-chains are the small chains decorating and giving properties to the backbone, as shown in figure 5.2.

The link between two amino-acids is characterized per two angles in the two planes of the 3-dimensional space. This angles receive the name of rotamers.

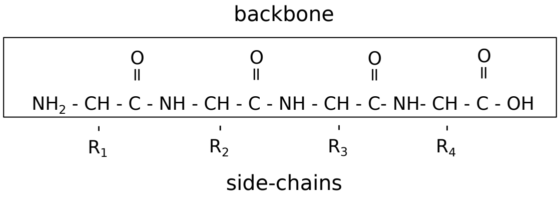

backbone
O
O
O
O
NH_2 - CH - C - NH - CH - C - NH - CH - C- NH- CH - C - OH
R_1
R_2
R_3
R_4
side-chains

Figure 5.2: Structure of a protein. Backbone and side-chains.

Amino-acids can adopt different forms or conformations since side-chains can be folded in different ways. A particular conformation of a side-chain is defined by the collection of angles between the different atoms constituting the side-chain. However, not all the possible angle configurations are equiprobable. In fact, it is known that there are some configurations that appear repetitively in the nature. They are known as stable side-chain configurations. The biology research community has studied the characteristics of such stable configurations, fruit of this research, there are different approaches that approximate the side-chain configuration given a particular amino-acid context. Some relevant libraries that were used in the last decades are [Lovell et al., 2000, Tuffery et al., 1991]. Nowadays the most relevant rotamer library is [Dunbrack, 2002].

As said, the particular sequence of amino-acids together with their side-chain conformations is what determines the properties of the protein. In particular, there is one property that has a capital relevance to our problem and no less relevance to the protein functional purpose: the protein tridimensional structure.

5.2.2 The Side-Chain Prediction Problem

Roughly speaking, the conceptual idea catching the essence of the problem is "*Draw a protein first, then let's find how to build it*". The second part of the sentence is what the Side-Chain Prediction is about. Let us give you a more detailed view of the problem.

In the Side-Chain Prediction problem, the starting point is the 3D structure of a protein and the objective is to find which are the sequence of amino-acids and their corresponding side-chains. This problem is a classic but still a difficult challenge in computational biology that has been tackled from different perspectives, i.e. [Summers and Karplus, 1989, Holm and Sander, 1991, Lee and Subbiah, 1991, Tuffery et al., 1991, Desmet et al., 1992, Dunbrack Jr and Karplus, 1993, Wilson et al., 1993, Kono et al., 1994, Laughton, 1994, Hwang and Liao, 1995, Koehl and Delarue, 1995, Bower et al., 1997, Samudrala and Moult, 1998, Dunbrack Jr, 1999, Mendes et al., 1999, Xiang and Honig, 2001, Liang and Grishin, 2002].

Typically, solving methods define an energy function over a discretization of the side-chain angles, then the global minimum can be found using search algorithms. Even when the energy function contains only pairwise interactions, the configuration space grows exponentially and it can be shown that the Side-Chain Prediction problem is NP-complete [Fraenkel, 1997, Pierce and Winfree, 2002].

The Model

A common model is to express the Side-Chain Prediction problem as a *Combinatorial Optimization* problem where the optimization function minimizes the energy in terms of pairwise interactions among nearby residues and interactions between a residue and the backbone.

More formally,

$$E(r) = \sum_{<ij>} E_{ij}\left(r_i, r_j\right) + \sum_{i} E_i\left(r_i, backbone\right) \tag{5.1}$$

where $r = (r_1, ..., r_N)$ denotes an assignment of rotamers for all residues, and E the result of applying an Energy function.

Library of Rotamers

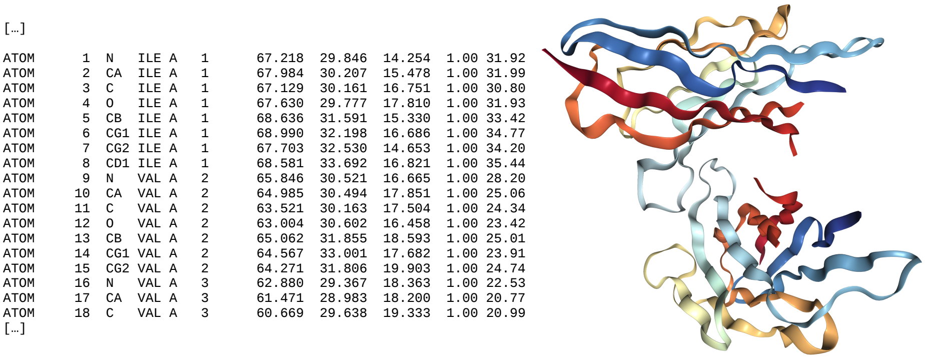
All the known methods require the use of a library of rotamers to work. The library of rotamers provides a description of which are the stable configurations of the residues as they have been observed in the nature. Thanks to the use of a library of rotamers we can discretize the space of solutions considering only the stable configurations of the side-chains as possible solutions, in our notation $r_i \in LR(P, i)$ where LR represent the call to the rotamer's library, P is the protein and i the side-chain we are evaluating. The rotamer's library will return the possible formations of the side-chain i.

The Energy Function

As stated before, the optimization function minimizes over the energy needed to conform the protein. There exist different energy functions that try to estimate the energy consumption of a protein [Canutescu et al., 2003, Rohl et al., 2004]. Different energy functions generate different graphical models to be solved. The election of the energy function has impact in the quality of the solution but also in the amount of computation that has to be done.

5.2.3 PDBs and SCWRL4

The Protein Data Bank (PDB) was established in the 1970's as the first open-access digital resource in the biological sciences. It is today a leading global resource for experimental data central to scientific discovery. Through an online information portal and downloadable data archive, the PDB provides access to 3D structure data for large biological molecules (proteins, DNA, and RNA). It was established by Walter Hamilton at Brookhaven National Laboratory. The PDB contained at that time 7 structures. Nowadays the PDB is coordinated by the Research Collaboratory for Structural Bioinformatics (RCSB) and at



```
[…]

ATOM      1  N   ILE A   1      67.218  29.846  14.254  1.00 31.92
ATOM      2  CA  ILE A   1      67.984  30.207  15.478  1.00 31.99
ATOM      3  C   ILE A   1      67.129  30.161  16.751  1.00 30.80
ATOM      4  O   ILE A   1      67.630  29.777  17.810  1.00 31.93
ATOM      5  CB  ILE A   1      68.636  31.591  15.330  1.00 33.42
ATOM      6  CG1 ILE A   1      68.990  32.198  16.686  1.00 34.77
ATOM      7  CG2 ILE A   1      67.703  32.530  14.653  1.00 34.20
ATOM      8  CD1 ILE A   1      68.581  33.692  16.821  1.00 35.44
ATOM      9  N   VAL A   2      65.846  30.521  16.665  1.00 28.20
ATOM     10  CA  VAL A   2      64.985  30.494  17.851  1.00 25.06
ATOM     11  C   VAL A   2      63.521  30.163  17.504  1.00 24.34
ATOM     12  O   VAL A   2      63.004  30.602  16.458  1.00 23.42
ATOM     13  CB  VAL A   2      65.062  31.855  18.593  1.00 25.01
ATOM     14  CG1 VAL A   2      64.567  33.001  17.682  1.00 23.91
ATOM     15  CG2 VAL A   2      64.271  31.806  19.903  1.00 24.74
ATOM     16  N   VAL A   3      62.880  29.367  18.363  1.00 22.53
ATOM     17  CA  VAL A   3      61.471  28.983  18.200  1.00 20.77
ATOM     18  C   VAL A   3      60.669  29.638  19.333  1.00 20.99
[…]
```

Figure 5.3: Left: portion of a PDB file description of a molecule. Right: 3D representation given the whole PDB input.

the time of this writing there are more than one hundred thousand molecules described.

PDB is also the file format describing the molecules. The PDB files describe the atoms of a molecule and their coordinates in the 3D space. There exist a large number of tools that are able to visualize the proteins described by a PDB. The web resource itself (http://www.rscb.org) allows to view and interact with a molecule. Figure 5.3 shows a partial view of a PDB file and its visualization.

The state of the art solver for the Side-Chain Prediction is SCWRL4 [Krivov et al., 2009]. SCWRL4 is developed at the Dunbrack Lab in the Fox Chase Cancer Center and it is the most advanced software for solving the Side-Chain Prediction. SCWRL4 can be freely used for research purposes but its source code is proprietary and it is not accessible.

SCWRL4 is able to execute Side-Chain Prediction simulations over known proteins described in PDBs. To achieve the simulation, SCWRL4 extracts the backbone atoms and their coordinates from a PDB. Next, it generates a probabilistic model using the side-chain configurations provided by rotamer library and then apply the energy function over the different possibilities. Finally, after solving the optimization problem, SCWRL4 places the minimal configuration of side-chains in place.

One interesting feature of SCWRL4 is the option to dump the optimization problem into a file once it is internally built and opens the door to use the combination of RSCB Protein DataBank + SCWRL4 as generator of *Combinatorial Optimization* problems.

5.3 Solving the Problem

In this section we study the current performance of AD^3 when solving LP relaxations for the proteins published at the SCWRL3-Based dataset [Yanover et al., 2006]. This dataset is formed by proteins of one unique chain and up to 1000 residues. This model uses the SCWRL3 energy function [Canutescu et al., 2003].

We show how to encode the problem of Side-Chain Prediction as a Factor Graph able to be solved by means of AD^3. Next we encode all the problems of the Yanover's dataset as a Factor Graph and solve them using the public implementation of AD^3.

We observe how we are soon solving all the LP relaxations for all the problems in this dataset very fast. As AD^3 is powerful, SCWRL3 based problems in the Yanover dataset are solved very efficiently.

After concluding that it would be desirable to have a new dataset with hardest instances, we end the section building an new dataset containing harder problems.

5.3.1 How to Encode the Problem as a Factor Graph

Variables and Multi-Variables

The problem of the Side-Chain Prediction can be encoded as a Factor Graph to be solved by means of AD^3. In the Factor Graph encoding, every amino-acid is represented by a multi-variable. The term multi-variable stands for a variable that is formed by a set of binary variables. In the context of the Side-Chain Prediction, a multi-variable has as many binary variables as possible side-chains configurations can have the amino-acid, according to the information provided by the rotamer library.

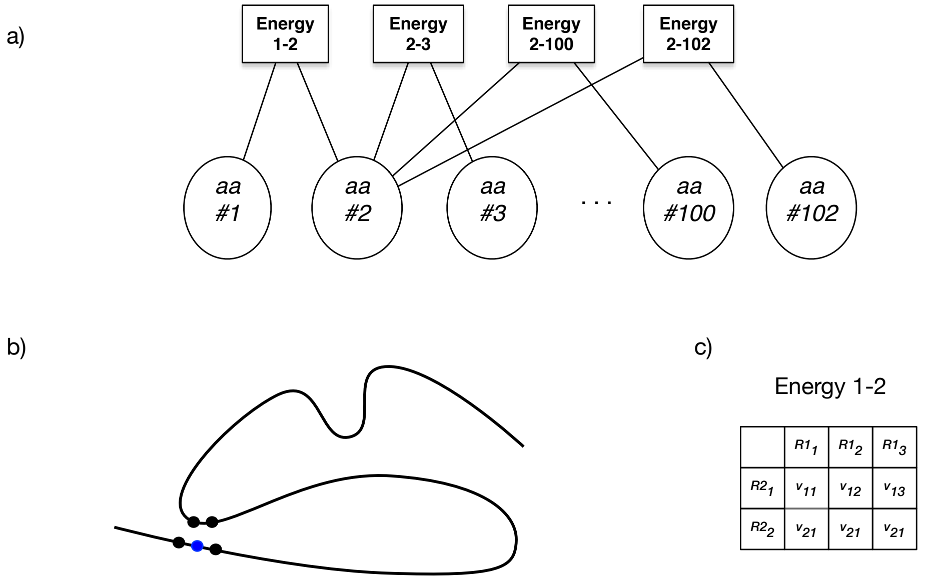
If, for instance, a given amino-acid can have two different side-chains that can be expressed in two and three configurations respectively, there will be a total 5 binary variables associated to the same multi-variable. Every multi-variable has one and only one of it's binary variables active, meaning that every amino-acid has only one radical position having one side-chain attached.

In addition, every different side-chain conformation requires a different level of energy is mapped to a potential value associated to every binary variable.

Connections Between Multi-Variables

A protein is a complex system where the forces from different atoms interact. The forces interacting between different aminoacids play a fundamental role in the protein folding process. In our model of the problem of the Side-Chain Prediction, forces are modeled as pairwise interactions, where every pair of aminoacids that are close enough in the 3D space are linked.

Figure 5.4a shows a graphical representation of a Factor Graph encoding the Side-Chain Prediction problem. In the figure we show only the interactions on aminoacid number 2. Figure 5.4 shows the shape of the protein we are trying to predict, where we mark the aminoacids that interacts with aminoacid number 2 (central point, in blue). We also show in Figure 5.4 how one of the factors looks like. It is a table of dimensions $|R_1| \times |R_2|$ where $|R_1|$ and $|R_2|$ are the number of possible radical configurations of each aminoacid of the interaction. The content of the table cells is the energy required by the protein to express the given radical selection.



	$R1_1$	$R1_2$	$R1_3$
$R2_1$	v_{11}	v_{12}	v_{13}
$R2_2$	v_{21}	v_{21}	v_{21}

Figure 5.4: a) A graphical representation of the pairwise interaction of the aminoacid number 2 in a Factor Graph. b) A schematic 2D drawing of the positions of the aminoacid influencing the aminoacid number 2. c) A table with the different stable configurations of aminoacid number 1 and aminoacid number 2; every combination requires a certain Energy.

5.3.2 Effectively Solving the LP problem

AD^3 has be proven to efficiently solve the problem of the Side-Chain Prediction, using the dataset from Yanover. Martins compares the performance of AD^3 with MPLP, using David Sontag's [Sontag et al., 2012] implementation showing the progress in the dual objective over the time for two of the most largest problem instances. In both cases, AD^3 rapidly surpasses MPLP in getting a better dual objective.

AD^3 claims to be the fastest algorithm solving the LP relaxations for Yanover's Side-Chain Prediction dataset, since Sontag' MPLP [Sontag et al., 2012] revisited the problem of the Side-Chain Prediction, finding that MPLP obtained better results than TRBP. Furthermore, TRBP was proved to obtain better performance than CPLEX –The general purpose LP solver from IBM– in [Yanover et al., 2006].

Experimental Hardware Setup

We run our experiments in a server with an Intel(R) Core(TM) i7-7500U processor with hyper-threading enabled. This processor has a Last Level Cache (L3) cache of 15GB and 6 cores with two levels of private caches. The server has 32GB of RAM.

Solving the Yanover's Dataset

We have thoroughly tested AD^3 against Yanover's dataset, with the objective of characterizing AD^3 execution. A problem can be finished in three ways:

- finding an optimal integer solution,
- monitoring the increment of the primal and dual objectives through the iterations and detecting when the result is not improving enough (according to a threshold) or
- stopping the process, reaching a maximum number of iterations or execution.

We are looking different aspects from the qualitative results of our experimentation.

Solution type. The dataset is formed by 369 proteins, AD^3 is able to find the optimal integer solution for 93.77% of the problems. In the rest of the cases AD^3 is able to converge to an approximate solution with an error smaller than

-10^6. Note that his error is negligible compared to the amount of energy we are trying to minimize is in the order of 10^3.

Solving time. AD^3 is able to solve the 72.1% of the dataset in less than one second, 22.5% in less than five seconds. The 5.4% left are the hardest cases that require a mean of 12 seconds to be solved. The hardest case is the protein named '1kwh' and it solved optimally in 31.3 seconds. Figure 5.5a shows AD^3 solving times for the dataset in growing order.

Iterations needed. Close to 96% of the proteins need less than 2000 iterations to be solved. The remaining proteins require a mean of 2822 iterations. The maximum number of iterations needed by AD^3 is 5364, and it is again for the '1kwh' protein. Figure 5.5b shows the number of iterations that AD^3 require to solve the dataset.

Summarizing the measures just described, AD^3 finds the optimal for 93.77% of the problems, it solves 94.6% of the problems in less than 5 seconds, and 96% of the executions require less than 2000 iterations. At the light of these results we can extract two main conclusions. First, AD^3 is suitable for solving the Side-Chain Prediction problem and, second, the selection of proteins in Yanover's dataset doesn't constitute a big challenge for AD^3 since almost all of them are solved optimally and efficiently.

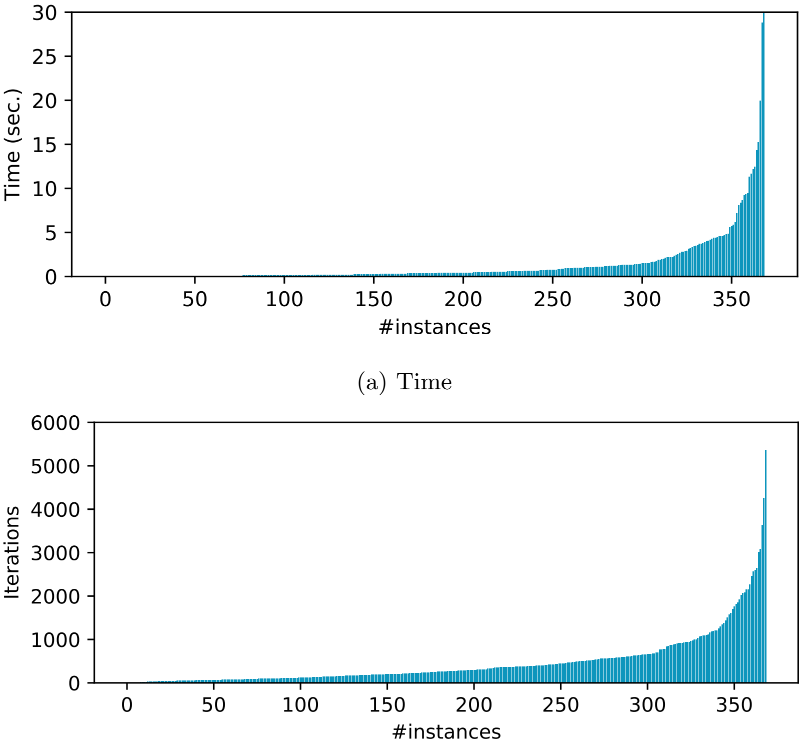
Our objective is to optimize and propose a parallel version of AD^3, and to that end we require to have computationally intensive tasks. However, in the case of the Yanover's dataset only a few portion of the problems represent a real challenge for AD^3, barely 5-10 cases. For this reason we find the need of building a new protein dataset.

5.3.3 Designing a New Dataset

Since most of the problems in Yanover's SCWRL3-Based dataset are not hard enough to stress the AD^3 algorithm, we generated a new dataset with bigger proteins. The 296 proteins in the dataset are unique chains of aminoacids containing between 1000 and 4000 residues.

The protein description files are downloaded from `http://rcsb.org`. We use the SCWRL4 [Krivov et al., 2009] tool on those files to generate the graphical models. SCWRL4 integrates a more sophisticated energy function than its predecessor (SCWRL3) to generate more accurate results, and at the same time graphical models that are much more complex

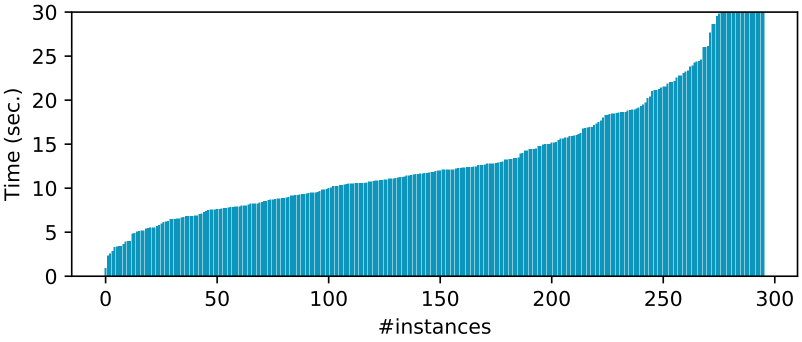
We run AD^3 for solving the experiments in the new dataset to corroborate



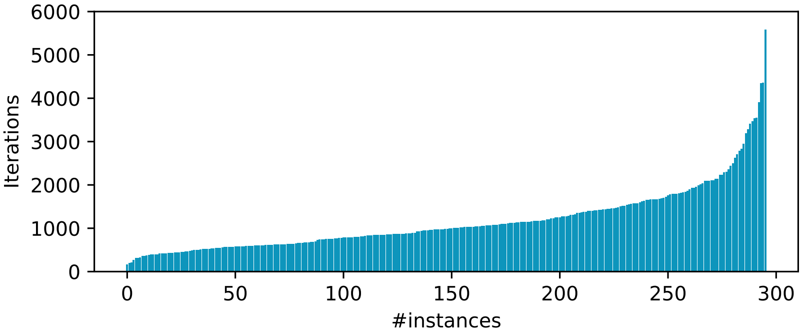
(a) Time

(b) Iterations

Figure 5.5: AD^3 Time and number of iterations required by AD^3 for each protein of the Yanover's dataset. The results are shown with the instances sorted by increasing time



(a) Time



(b) Iterations

Figure 5.6: AD^3 Time and iterations required by AD^3 for each protein with the new proposed dataset. The results are shown with the instances sorted by increasing time.

that there is a higher number of hard problems. Figure 5.6 plots the time required to find a solution by AD^3 on the instances of the new dataset as well as the number of iterations required. These two plots are analogous to the ones in Figure 5.5. We keep the scale of the axes to make the figures visually comparable. The execution times ranging between 30 seconds and 160 seconds that correspond to the 20 hardest instances in the dataset are cut at the right of the figure 5.6a.

5.4 Optimizing the Computation of *Arbitrary* Factors

This section presents the main contribution of the chapter: the realization of an optimized design of the algorithm to solve an *Arbitrary* factor. We first underline the main differences between the *Arbitrary* factor and the *AtMostOne* factor.

- **Use case.** The applicability of the *Arbitrary* factor is much wider, since it is able to implement any function.

- **Morphology.** The *Arbitrary* factor is able to represent pairwise interactions between multi-variables. In contrast, the *AtMostOne* factor operates with binary variables only, but with heterogeneous arity.

- **Complexity.** While solving the quadratic problem for first order logic factors –like the *AtMostOne* factor– is computationally very lightweight (see Algorithm 5 in Chapter 4), solving the quadratic problem for an *Arbitrary* factor requires significantly much more computation work. Moreover the computational effort that is needed to solve an *Arbitrary* factor can substantially vary from one factor to another: the workload is not balanced.

- **Integration.** Some operations involved for the *AtMostOne* factors depend only on edge-related information and can be interleaved with graph-wise operations, which provides opportunities for efficient parallel execution (see section 4.5.2). In contrast, the elementary operations needed to solve the *Arbitrary* factor depend on a high amount of factor-related information and cannot be performed outside the factor.

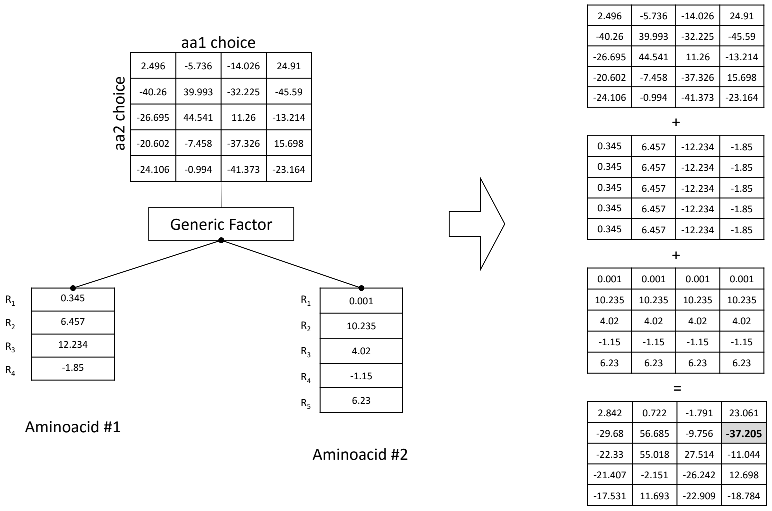
5.4.1 Algorithmic Description

This section presents an algorithmic definition for solving a quadratic problem in an *Arbitrary* factor. We focus in the type of operations and memory structures that are required to solve the problem. The reader can find a more detailed description of the principles behind the *Arbitrary* factor at [Martins, 2012b] (section 6.5).

Algorithm 7 shows an overview of the process of solving a quadratic problem for an *Arbitrary* factor. For simplicity, in the presented algorithm we have omitted the description of two computationally intense processes, in functions $projectVarPotentialsToFactor$ and $projectVarPotentialsWithDistribution$, in lines 5 and 15, respectively. They play an important role regarding execution performance, since they propagate the selection of the variables to the

Algorithm 7 *SolveArbitraryQP*: Solving an *Arbitrary* factor

```
Require: Var1 potentials ($Vp_1$), Var2 potentials ($Vp_2$), Factor Potentials ($Fp$)
function SolveArbitraryQP($Vp_1$,$Vp_2$,$Fp$)
    if $isEmpty(ActiveSet)$ then
        $A \leftarrow \begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}$
        $distribution \leftarrow [1]$
        $projectionMatrix \leftarrow projectVarPotentialsToFactor(Vp_1, Vp_2, Fp)$
        $ActiveSet_{\text{VAL}}.push(max(projectionMatrix))$
        $ActiveSet_{\text{POS}}.push(max\_pos(projectionMatrix))$
    end if
    for $i \in 1 \ldots$ MAX_ITERATIONS do
        if $hasChanged(ActiveSet)$ then
            $[\tau, z_1, \ldots, z_n] \leftarrow A^{-1} \times [1] \oplus ActiveSet_{\text{VAL}}$
        else
                                        ▷ find a potential to add to the activeSet
            $output = initialize(Vp_1, Vp_2, Fp, distribution)$
            $output = projectVarPotentialsWithDistribution(Vp_1, Vp_2, Fp, distribution)$
            if $(max\_val(output) \pm \varepsilon = \tau)$ or $(max\_pos(output) \in ActiveSet_{\text{POS}})$ then
                return
            end if
            $ActiveSet_{\text{VAL}}.push(max(output))$
            $ActiveSet_{\text{POS}}.push(max\_pos(output))$
            $A \leftarrow updateMatrix(A, ActiveSet)$
                                        ▷ adds a row and a column to matrix A
            $distribution \leftarrow update(A, ActiveSet)$
        end if
        if $blocked(ActiveSet)$ then
            $RemoveBlocking(ActiveSet)$
            Updates: $A, distribution$
        end if
    end for
end function
```



2.496	-5.736	-14.026	24.91
-40.26	39.993	-32.225	-45.59
-26.695	44.541	11.26	-13.214
-20.602	-7.458	-37.326	15.698
-24.106	-0.994	-41.373	-23.164

R_1	0.345
R_2	6.457
R_3	12.234
R_4	-1.85

R_1	0.001
R_2	10.235
R_3	4.02
R_4	-1.15
R_5	6.23

2.496	-5.736	-14.026	24.91
-40.26	39.993	-32.225	-45.59
-26.695	44.541	11.26	-13.214
-20.602	-7.458	-37.326	15.698
-24.106	-0.994	-41.373	-23.164

0.345	6.457	-12.234	-1.85
0.345	6.457	-12.234	-1.85
0.345	6.457	-12.234	-1.85
0.345	6.457	-12.234	-1.85
0.345	6.457	-12.234	-1.85

0.001	0.001	0.001	0.001
10.235	10.235	10.235	10.235
4.02	4.02	4.02	4.02
-1.15	-1.15	-1.15	-1.15
6.23	6.23	6.23	6.23

2.842	0.722	-1.791	23.061
-29.68	56.685	-9.756	**-37.205**
-22.33	55.018	27.514	-11.044
-21.407	-2.151	-26.242	12.698
-17.531	11.693	-22.909	-18.784

Figure 5.7: An example of a calculation of projectVarPotentialsToFactor

factors. We next present an overview of them.

Let us have two multivariables aa_1 and aa_2, and a factor f linking these two variables. Every multi-variable is formed by a set of $|aa_1|$ and $|aa_2|$ binary variables and their associated potentials. The factor potentials F is a matrix with dimension $|aa_1| \times |aa_2|$. The function $projectVarPotentialsToFactor$ maps all the potentials of aa_1 and aa_2 to the factor f and then finds the minimum energy. It does that defining two all-ones vectors as $ones_1$ and $ones_2$, where $|ones_1| = |aa_1|$ and $|ones_2| = |aa_2|$, and then computing the projected matrix as $P = (\overline{aa_1})^\intercal \times ones_2 + \overline{aa_2} \times (ones_2)^\intercal + F$. An example of this calculation is shown at figure 5.7. Function $projectVarPotentialsWithDistribution$ works in a very similar way, but this time some energetic values are weighted according to a distribution.

In line 21, matrix A is updated, but this update must guarantee that the resulting matrix will not be singular, otherwise the inversion in line 11 will fail. The original strategy to handle this problem was very cumbersome. we have devised a simpler solution that consists of checking the singularity of the matrix while the matrix is created: if a combination of row and column introduces a singularity, then this combination is removed.

5.4.2 Optimizing Performance

Solving factors implementing the *Arbitrary* factor requires that some data structures must be kept into memory persistently. We show how a good choice of the memory representation strategy can help to increment the performance. We also show some techniques to optimize computations. During the rest of the chapter we will refer to the original implementation of AD^3 as *Reference version* and our contributed version as *Optimized version.*

Next we present which techniques we used, why we used them and what is the expected result. Finally, we present empirical evidence of our improvements.

Memory Design

The data required to represent a problem with *Arbitrary* factors can be divided into two classes. On one hand, we have data structures that store the state of the problem; this is the case for variable potentials, factor potentials, the list of variables or factors that are active, or the value of the lambdas corresponding to the edges. On the other hand, memory is needed for some intermediate calculations, such as the aforementioned matrix operations or the current list of candidate solutions for a given factor.

The first class of data is allocated statically at the beginning of the execution in both the *Reference version* and the *Optimized version.*

On the contrary, the second class of data –the one generated for intermediate calculations– is allocated and freed dynamically in the *Reference version*, while it is statically allocated in the *Optimized version.* This decision has some implications that are discussed next.

The first and obvious implication is that allocating memory in advance avoids to eventually request memory to the operating system. Allocating new memory is, in general, expensive in terms of performance. Hence a reduction of the number of memory allocation requests has the potential of increasing performance.

In exchange, the *Optimized version* has to estimate in advance the amount of memory the algorithm is going to need. Unfortunately, this information is not known beforehand. In addition, the memory will not be evenly distributed across the factors. A precise analysis to the operations done for factor calculation reveals that a factor maintains a list of pairwise solutions, named as active set, and that memory requirements have a quadratic growth with respect to this *active set.* Hence there will be factors with high memory requirements, the ones that are very active considering more possible solutions, and idle factors with very few memory requirements. We propose to reserve memory in advance using a double strategy.

- **Per thread scratch areas**. We create scratch areas that are private to each execution thread. Scratch areas are used to store data that is going to be consumed in the short term. For instance, when inverting a matrix using LU decomposition, partial results are stored in the scratch area. Separating scratch zones per thread, we assure that there will not be undesired effects like race competitions or false cache sharing.

- **Per factor pre-allocated memory space**. We allocate a configurable amount of memory assigned exclusively to each factor. Since the memory requirements are quadratic with respect to the number of active possibilities a factor is considering, we can define this parameter as the maximum number of active possibilities a factor can hold and represent it by ϱ . Then, if it turns out that a particular factor is very active and needs more memory, the algorithm ask the system for free memory –although this call will be expensive–. A downside of this pre-allocation is that the initial memory given to each factor has to be decided in advance. However, in practice AD^3 doesn't have large memory requirements and, in the case of the Side-Chain Prediction, considering values of the order of $\varrho \approx 50$ leads to a good compromise of a limited memory usage and very few requests for extra memory.

Using a memory profiling tool, we obtain the total amount of memory allocation calls done by the program. The *Reference version* needs an average of 7.16×10^6 allocation calls when solving the Side-Chain Prediction dataset. In contrast, the *Optimized version* needs an average of 1.57×10^5, which is a significant reduction of 97.8% of the system calls.

As we will detail in the next section, the *Optimized version* executes less instructions. This fact has an impact in the memory system where we have measured an average of 57% less memory accesses to the main memory system in the Side-Chain Prediction dataset.

However, there are two features that are very similar in both algoritms. On one hand, the *Reference version* as well as the *Optimized version* have comparable memory footprints, i.e. an average of around 200MB for our dataset. On the other hand, both algorithms have a similar Last Level Cache Hit rate, 27% and 29% in the *Reference* and the *Optimized* versions, respectively. These numbers are reasonable since the data the algorithm have to consume is one order of magnitude bigger than the cache system, and the memory access pattern is arbitrary, i.e. not known beforehand. A possible future work emerges from this result, since a proper graph partition or factor reorganization could lead to an improvement in the locality of the algorithm.

5.4.3 Reducing the Number of Instructions

In a single-thread execution, the total execution time of an algorithm can be expressed as the total number of instructions executed × the average number of Instructions executed Per Cycle (IPC). Next we describe the strategies we have followed to reduce the total number of instructions executed. Our strategies pursue the objective of reducing instructions executed, but also exploiting the capacities of a modern processor and also get benefit of present and future generations of processors.

Reducing Abstraction

After a analyzing the type of operations that are performed by the *Reference version*, the *Optimized Version* expresses the operations in a more machine-understandable fashion. As other numerical libraries do (like STL, Boost or MKL) we sacrifice the encapsulation and the versatility of the Object Oriented paradigm. In exchange, we obtain a lower overhead and a bigger control of the code that is produced. As a result of this process, we produce a totally new implementation where there are deep differences in data structures and code organization.

Promoting Vectorization

Our *Optimized Version* uses vectorized instructions. Moreover, the code has been designed to be auto-vectorized by the compiler. We empirically measure that our vectorization yields a reduction of around 20% of the instruction count in our current architecture. We believe that an auto-vectorizable code will benefit better from future processors, since support for vector instructions is a trending hardware technique.

Achieving a vectorized code can be done in different ways. Our approach is to organize the code in a way that the compiler will know how to vectorize without the need of using machine-dependent intrinsics. This way we ensure portability and reusability of our code in future processor generations. Next we illustrate two paradigmatic algorithmic changes we have performed to loops in order to help the compiler to generate a vectorized code.

The first example in figure 5.8 shows a portion of code that removes a particular element with position *rmv* of different vectors simultaneously. In the case of the first implementation the compiler is not able to vectorize the code, since there is a real dependence on every iteration with the previous one. All the vector accesses depend on the variable *vector_dest_pos* which is conditionally increased in almost every iteration. For the compiler it is not possible to produce a prediction of the value of *vector_dest_pos* at compilation time, and consequently it generates code that is not vectorized (or scalar).

Instead, we alter how *vector_dest_pos* is computed, since it can be easily ob-

```

vector_dest_pos=0;
// Loop not vectorized
for (int i=0; i < end;i++) {
    if (i!=rmv) {
        activeSetCopy[vector_dest_pos] = data->activeSet[i];
        if (!keepz) zCopy[vector_dest_pos] = z[i];
        if (!keepd) dCopy[vector_dest_pos] = d[i];
        vector_dest_pos++;
    }
}
```

```
// Loop Vectorized
for (int i=0; i < rmv;i++) {
    activeSetCopy[i] = data->activeSet[i];
    if (!keepz) zCopy[i] = z[i];
    if (!keepd) dCopy[i] = d[i];
}
// Loop Vectorized
for (int i=rmv+1; i < end;i++) {
    activeSetCopy[i-1] = data->activeSet[i];
    if (!keepz) zCopy[i-1] = z[i];
    if (!keepd) dCopy[i-1] = d[i];
}
```

Figure 5.8: Vectorization example. Breaking Read-After-Write (RAW) and Write-After-Read (WAR) dependencies

```
double *output = &data->tau[0];
for (int x=0;x<width;x++) {
    output[x]=0;
    // Loop not Vectorized
    for (int y =0; y < width; y++ ){
        output[x]+=data->inverseA[x*width + y] * data->varB[y];
    }
}
```

```
double *output = &data->tau[0];
for (int x=0;x<width;x++) {
    double acc=0;
    // Loop Vectorized
    for (int y =0; y < width; y++ ){
        acc+=data->inverseA[x*width + y] * data->varB[y];
    }
    output[x]=acc;
}
```

Figure 5.9: Vectorization example. Avoiding memory aliasing problems

tained knowing the current iteration and the position that is going to be removed. This time, this loop is vectorized, potentially speeding up the execution of this particular loop up to 4 times in our current hardware. In addition this code is not machine-dependent, assuring compatibility with different architectures.

In the second example shown in figure 5.9 the compiler is not able to vectorize the inner loop. The code performs a reduction with the addition operation, which is easily vectorizable. However, the problem is that the compiler is not able to determine if the memory location of $output[x]$ aliases with a memory location of $data- > inverseA[...]$ or $data- > varB[y]$. Hence, the compiler assumes the worst case: a possible data dependence, and fails to create vectorized code.

Again, a slight change in the code, i.e. the definition of a new variable that accumulates partial results, enables code vectorization and provides a potential 4X improvement.

We have helped the compiler to successfully vectorize 17 loops in the code for solving the *Arbitrary* factor in the *Optimized version*, while only 5 loops with real data dependencies remain scalar.

Using Specialized Libraries

While the *Reference version* uses both predefined libraries and specific code to compute standard algebraic operations, the *Optimized version* expresses all the algebraic operations in terms of the standard Linear Algebra Package (LAPACK). Using a state-of-the-art optimized library such as LAPACK comes with a number of benefits: it is portable since LAPACK is extended in many different architectures; it is well-tested since it is widely used worldwide; it is optimized since it is the main purpose of the library; and also it is easy to use. LAPACK is also available through the Intel's MKL library, offering the possibility of obtaining a more specialized code for the Intel Processors.

AD^3 is a computationally intensive algorithm that works considering a set of 'active' variables for every factor, i.e. all variables that do not have a solution assigned yet. AD^3 computes scores in order to select which is the next variable to be included in the active set. This score is calculated using double precision, leading to very precise small values. When working with this level of precision, the numerical method used to compute the score can produce slightly different results. In some cases this difference makes that different numerical methods can lead to a different variable selection on particular factors. This fact introduces a divergence between two algorithms that makes very hard to contrast results. That's the case of the *Reference* and the *Optimized* versions where different numerical methods are used and they generate different solutions at particular subproblems. However, in the big picture, the algorithm converges to the same solution. We found that using LAPACK, the *Optimized version* gets the same but marginally better solutions in the case of AD^3 solving our Side-Chain Prediction dataset.

5.5 Performance Analysis

In this section, we present an analysis of the performance of the *Optimized version* of $PAR\text{-}AD^3$ with *Arbitrary* factors. In our experimentation we have solved all the instances of the protein dataset with both the *Optimized version* as well as with the *Reference version.*

We first show our results when executing the two algorithms using a single thread (single-thread execution). Note that the *Reference version* is only able to run in a single-thread way. Then, we present a deeper analysis of the performance limitations of the execution of the *Optimized version.* We make this analysis both for single-thread execution, when using only one processing core of the processor, and then for the multi-thread execution and using the full computational capacities of our system.

The *Optimized version* obtains a speedup of 2.7X in single-thread execution with respect to the *Reference version*, and a total speedup of 10.8X when using a 6-core computer.

5.5.1 Optimized vs Reference Version

We compare the performance of single-thread executions of the *Reference* and *Optimized* versions when solving LP relaxations for all the problems of the protein dataset. Our experimentation reveals that our *Optimized Version* is able to obtain an average Speedup of 2.7X with respect to the *Reference version.*

Figure 5.10 shows the time required to solve every instance of the dataset. Results are ordered by elapsed time required by the *Reference Version* to solve each case. The figure also depicts the speedup reached by the *Optimized version* with respect to the *Reference version.*

With the help of the hardware performance counters, we investigate the reason that is causing this speedup. We find that the *Optimized version* is able to execute an average of approximately one third (36.8%) of the instructions executed by the *Reference version.* In exchange, the *Optimized version* delivers a slightly higher average instruction throughput, from 1.26 Instructions per cycle (IPC) in the case of the *Reference version* to 1.19 IPC in the *Optimized version.* i.e. the *Reference version* is able to execute around a 5.5% more instructions per time unit. The significant reduction of instructions together with the slight

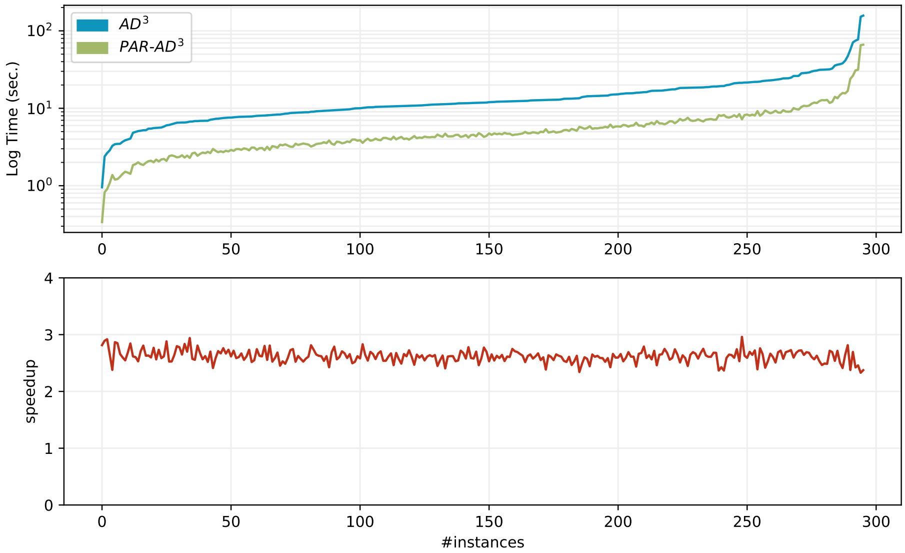


Figure 5.10: Solving time and speedup reached by the *Optimized version* with respect to the *Reference version* when solving all the LP relaxations of our protein dataset. Results are sorted by the elapsed time experimented by the *Reference version*

difference in the IPC makes the *Optimized version* obtain the aforementioned 2.7X single-thread average speedup.

Overall, the *Optimized version* has an IPC of 1.19. This is an average instruction throughput ratio. In the next two sections we inspect the probable causes that explain this moderate IPC (in the best cases, values of IPC between 2 and 3 can be found).

5.5.2 Single-Core Execution

In this section we identify the factors that limit the performance of the *Optimized version*. A moderate IPC indicates that there are instructions waiting to be executed while some computation resources are available. This can be explained by four factors:

- **Data dependencies**. The execution of the available instructions waits until a previous result has been calculated.

- **Memory latency**. The execution waits for the data to be fetched from memory.

- **Computation throughput**. The execution cannot move forward because the computational resources are busy.

- **Branch mispredictions**. The execution advances in the wrong direction after a branch prediction. This speculative execution was competing for resources while was executed, therefore introducing unnecessary overhead. In addition, the hardware has to rollback the executed code paying also a cost in performance.

In order to identify which of these performance limiting factors is having more impact in the execution, we run a set of experiments in a single-core scenario. We use two different configurations: first, we solve all the instances of the dataset using one thread, then we repeat our experiments using two threads running concurrently in the same core. Scheduling two threads concurrently in the same core allows the hardware to mask the waiting times for one thread by processing work from the other thread. This experiment helps us to identify the possible bottlenecks.

Table 5.1 summarizes the metrics we have measured. We observe that using two threads in the same hardware the performance is increased delivering a speedup of 1.11X. Although this 11% is not a large improvement, this extra gain underlines that in the two-threaded execution the hardware was able to use one thread's idle time to advance in the execution of the second thread. We also observe that the two-threaded execution is able to process more iterations –understood as effective word– per unit time, but also that it requires a higher number of instructions per work unit (reasons for this increment will be analyzed in the next section). The two-threaded configuration is able to obtain a higher IPC as it is expected from using the hyperthreading mechanism. Furthermore the two-threaded configuration exhibits a lower Last Level Cache (LLC) access per instruction ratio. However this lower ratio is solely due to the increase in the number of instructions. The two-threaded execution needs to access more to the LLC, and this is also expected, since in this execution the lower level caches are shared by the two threads. Finally we observe that the two executions have a similar branch prediction behavior, which is very good (a very small amount of mispredictions).

Considering the performance limiting factors described above, and in the light of the experimental results, we find that the two-threaded execution is able to produce more work since it delivers more iterations per time unit, but in exchange adds an overhead in terms of computational effort.

PAR-AD^3 is a complex algorithm having different stages, each one with a particular computational pattern. It is expected that some stages will be constrained by some performance limiter, while other stages may be limited

	1 thread	2 threads		1 thread	2 threads
Speedup	–	1.11X	$\frac{iterations}{elapsedtime}$	158.2	175.3
Instructions	4.98×10^{12}	5.46×10^{12}	$\frac{instructions}{iteration}$	14.48M	15.92M
IPC	1.19	1.46	$\frac{LLC\_accesses}{instruction}$	0.0092	0.0089
LLC_accesses	4.60×10^{10}	4.87×10^{10}	$\frac{branch\_misses}{instruction}$	0.0028	0.0026

Table 5.1: Performance metrics when executing in a single-core with one thread and with two threads.

by a different reason. The final observed result is an aggregation of all two factors. We next assume that all the relevant stages have a similar performance behaviour. This assumption provides an initial explanation of the performance behaviour of the algorithm. An in-deep analysis that remains an open line is to perform a detailed *per-stage* performance analysis.

We can safely say that the performance is not limited by memory problems or branch mispredictions, at least on the more expensive execution stages. The *LLC_accesses/instruction* and *branch_misses/instruction* metrics can explain the performance limiting factors of **Memory latency** and **Branch mispredictions** respectively. However we find that these numbers are not big enough to be constraining the performance of the execution. Note that the execution is performing approximately one LLC memory access every 100 instructions, which is a low ratio since our LLC can provide an access every 20 cycles approximately. We also find that there is a branch misprediction for every 500 instructions executed (0.2%), which is a very small fraction to harm performance significantly.

Therefore, we can conclude that the performance when executing in a single processing core is mainly limited by computational resources and dependencies. Using two threads in the same core we observe that the **Computation throughput** is improved but only to some extent. From that, we can infer that the current performance limitations have a double origin. First, **Data dependencies** are clearly a problem, since adding a new thread (more parallelism) yields to a higher IPC, and this increment of IPC means that the hardware schedules more work while trying to resolve a dependency. Also, the **Computational throughput** should be a problem, at least on some execution stages, and probably for different resources on different stages (like addition or multiplication), since the IPC is still moderate when executing with two threads and not explainable by any other factor.

5.5.3 Multi-Core Execution

In this section we analyze the parallel execution of the *Optimized version* using the full parallel capabilities of our 6-core processor. Our algorithm was able to obtain a 3.8X speedup using the 6 processing cores. Combined with the 2.7X speedup with respect to the *Reference version*, we obtain an aggregated average speedup of 10.26X.

Next, we present a scalability analysis of the parallel performance of the algorithm. Figure 5.11 depicts the increase of the execution performance using an increasing number cores, and allocating one or two threads per core. In our experiments we solve all the LP relaxations of the problems at the SCP dataset for every thread-core configuration. We define the performance as the number of iterations processed per time unit. In the figure we can observe how increasing the number of cores leads to a performance improvement. However, the performance increase is not linear with respect to the number of hardware resources that are used. We observe two effects. First, there is a degradation of the performance gain as the number of cores increases. Second, the hyperthreading benefits are also decreasing when increasing parallelism: in the more parallel scenario (using 12 threads in 6 cores) this effect is even harming the performance.

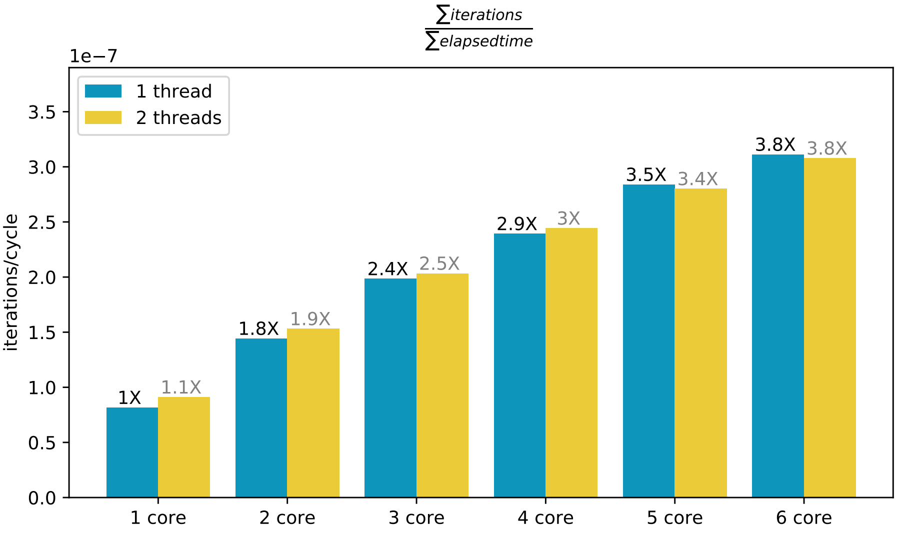


Figure 5.11: Iterations per cycle performed by $PAR\text{-}AD^3$ when using 1-6 cores and running one or two threads per processing core.

The limitations of the parallel execution performance just mentioned can be explained by three scalability limiting factors (SL1,SL2,SL3):

- **SL1. Lack of parallelism**. As stated by Amdhal's law, in a parallel program the sequential fraction of the execution limits the speedup of the overall execution.

- **SL2. Contention in Shared Memory resources**. The increment in the number of threads accessing the memory system may yield to a transient or persistent degradation of the memory system performance, which can be exacerbated by extra data communication requirements.

- **SL3. Unbalanced thread workload**. If the workload is not equally balanced between the different threads, then some threads may finish too early, decreasing part of the advantages of the parallel execution.

All these side-effects coexist in greater o less extent. Next, let us examine with more detail the particularities of our execution studying each of this three factors.

SL1. Lack of Parallelism

The lack of parallelism can explain a decrease in the performance due to the effects of the Amdahl's Law (described in chapter 2). This law defines how the sequential fraction of an algorithm constrains the maximum speedup that can be obtained by the parallel fraction. However as we saw in the previous chapter, *PAR-AD*^{3} was able to speedup the execution up to 5.2X using 6 threads (4.8X in average). Contributions in the present chapter only affect to the workload inside a thread and are not increasing the serial fraction. Quite the contrary, parallel workload is heavily increased, and in consequence the serial fraction is yet less relevant. All together makes that the effect of the Amdhal's law is having marginal impact and it is not a parallel performance limiter.

SL2. Contention in Shared Memory Resources

In a parallel execution, the memory system has to be able to provide data to different CPUs that are asking for data simultaneously. This situation can yield to a permanent contention spot or a transient saturation situation of the memory subsystem.

Let us consider a perfect parallel application that operates with data in memory and runs in a processor with N cores. This parallel application finishes its execution at time T when using 1 thread, and finishes at time T/N when using

N threads. Since the application is perfectly parallel, all the memory requests are evenly divided into threads. In this situation the memory has to provide N times more data per unit time in order to match the computation speed.

In practice, the situation is more complex. Typically, in a parallel application the memory system does not only need to feed more processing cores with data, but also there is a larger amount of data that must be provided. Taking our application as an example, we first show some reasons that explain this extra memory requests and later we examine the memory contention problem.

Extra Memory Requests

In a modern CPU with several cores, the cache system follows a hierarchical structure. Each processing core has access to some private cache area and some shared cache area. In the specific case of our processor, each core has two levels of dedicated or private caches named Level 1 (L1)[1] and Level Two (L2); and a cache that is shared by all the processing cores, known as level three (L3) r last level cache (LLC). Figure 5.12 presents a simplified diagram of the memory structure for 2 cores of our processor. We consider that the main memory, which is the higher level of the memory system, is not a member of the cache system and then it is not shown in the figure.

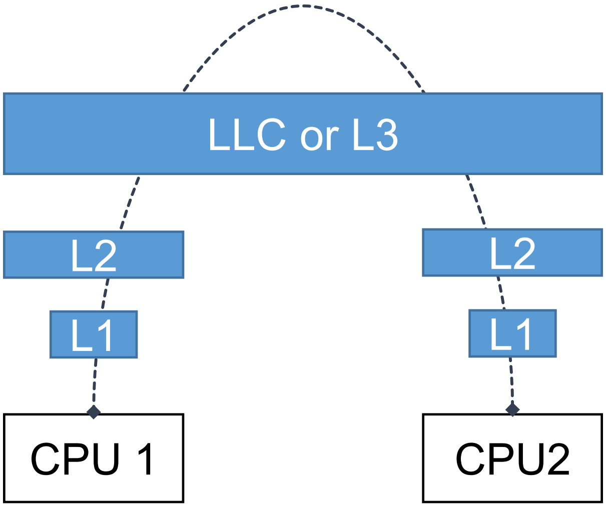


Figure 5.12: Memory cache structure for two cores.

The cache memory is not homogeneous: the caches that are close to the processor are fast but small and, as we ascend in the hierarchy, memories are slower but bigger.

A very relevant feature of the memory system is that it is transparent to

[1] In fact, our processor has two L1 caches, one for data and one for instructions.

the programmer: the memory system is seen by the CPU as a unique memory space. Computation units are physically wired only to the L1 cache. Each request that is not solved by L1 automatically ascends through the different levels in the memory hierarchy until it is satisfied. Therefore, if, for instance, a processing core requests to read a memory position that is not in any of the memory caches, the request travels through L1, then L2, then L3 (or LLC) and finally finds the data in the main memory. Then, the fetched memory line is brought to the processor through the memory hierarchy and keeping copies in every unit, hoping to be reused by future accesses.

In a parallel scenario, the private and shared caches have to maintain a coherence and the hardware establishes protocols of coherence that are also transparent to the processing unit and, henceforth, also to the user. However, although the programmer doesn't have control over cache-coherence protocols, they exist and have impact in the performance. When a cache line is updated in one private cache, it is automatically invalidated on all the other caches by the cache-coherence protocol, and this is potentially leading to a future cache miss, impairing performance. Hence, it is possible to measure to what extent the data invalidations are hurting the performance by observing the number of requests to LLC.

Figure 5.13 presents the total number of accesses to the LLC considering executions with different thread configurations. Let us focus first in the case of using one thread per core, and later in the hyper-threaded execution (2 threads per core).

- In the first case, we observe how the increasing the number of cores also increases the number of LLC accesses. The increment is around 13.5% when using 6 threads instead of using 1 thread. Since the total number of instructions also increases around 40%, the increase in LLC accesses is not very large and, in fact, the rate of LLC accesses per instruction executed is reduced.

- In the hyper-threaded execution, we observe that accesses to LLC are more frequent, increasing between 7% and 13% depending on the number of processing cores that are used. This is an expected behaviour since hyper-threading schedules two threads per core, and consequently the two threads are competing for the use of the private caches, and generate more misses in the lower level cache memories. This effect may be harming performance, since the execution with hyper-threading will naturally increase the average memory access time.

Memory contention

One of the factors that can damage the parallel scalability is the concentration of the memory requests in a shorter period of time, demanding more

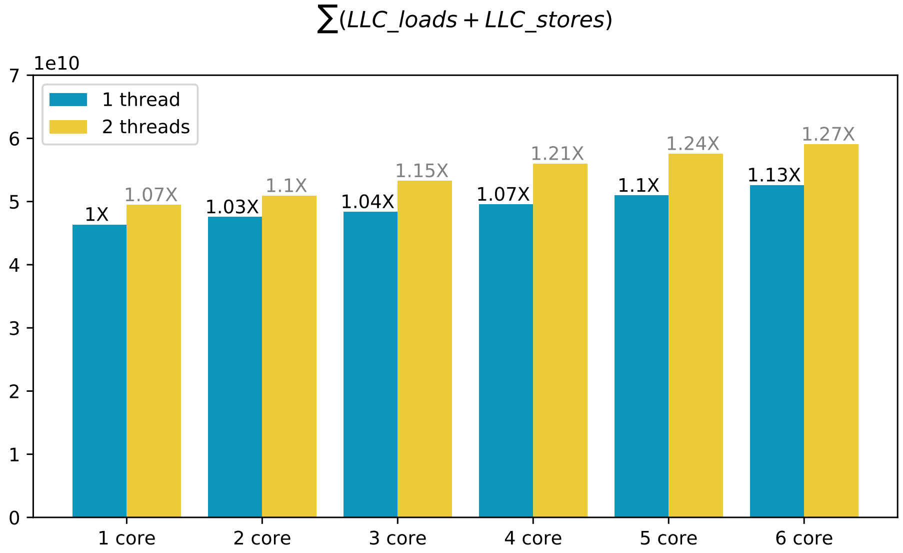


Figure 5.13: Load and store accesses to the Last Level Cache (LLC) using 1 and 2 threads per core and 1 to 6 core configurations.

bandwidth and throughput from the memory system than the system is able to support.

Figure 5.14 depicts the amount of requests per second performed by *PAR-AD*^{3}. The execution with 12 threads requires an average of 1.13×10^8 accesses to LLC per second; considering that our processor runs at 1.9GHz, we can compute that our execution is requesting a LLC access every 16.8 cycles. Since every access moves 64 Bytes, the effective bandwidth provided by the LLC is around 7.2 GB/s, which is far from the maximum peak LLC bandwidth (more than 40 GB/s). Hence the parallel execution is not limited by peak LLC bandwidth. Since the amount of LLC misses is small, we can also conclude that performance is not limited by peak DRAM bandwidth, either.

SL3. Unbalanced Thread Workload

In order to detect load balancing problems we analyze the number of instructions executed using different thread-core configurations. Figure 5.15 plots the amount of machine instructions executed for each iteration. This is repeated using different thread configurations. We observe how the number of instructions per iteration increases together with the number of threads. In an ideal parallel scenario this figure should plot equally-sized bars. We measure an increase of the computational effort, as the number of threads grows, between 5% and 10%

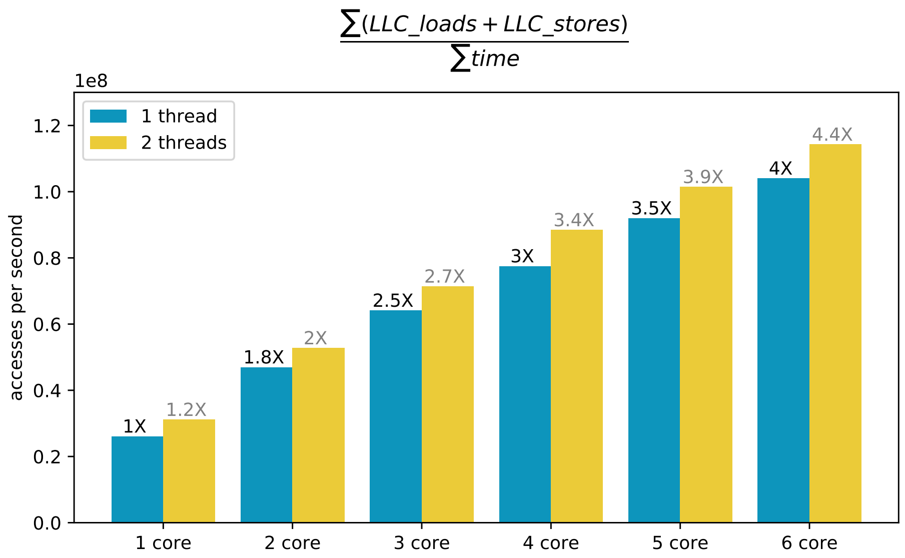


Figure 5.14: Load and store accesses to the Last Level Cache (LLC) per time unit using 1 and 2 threads per core and 1 to 6 core configurations.

per extra thread. Note that this increment is consistent regardless where the threads are scheduled. i.e. instructions/iteration measures for 2, 4 and 6 threads are the same using only one thread or two threads per core.

The growth in instructions per work unit can be explained by load balancing problems. When a parallel region reaches to an end, threads that finish first remain waiting until the rest of threads reach the same point. During this waiting time (or spin time) the threads are executing instructions. Looking at the increment of instructions per work unit gives us a good estimation of the load balancing problems. We observe a significant increase in the number of instructions executed by the processor when increasing the number of threads.

In order to better characterize the increment of instructions, we use a profiler to measure the number of instructions and number of cycles required by different functional units of the $PAR\text{-}AD^3$ in a representative problem. Figure 5.16 shows the amount of instructions and cycles required by the different execution stages of $PAR\text{-}AD^3$. We have distinguished four different stages or phases during the whole execution of the program:

- **Spin**. The processor is waiting for synchronization and executing non-productive instructions.

- **PAR-AD**^{3}. The processor is executing graph-wise $PAR\text{-}AD^3$ computations.

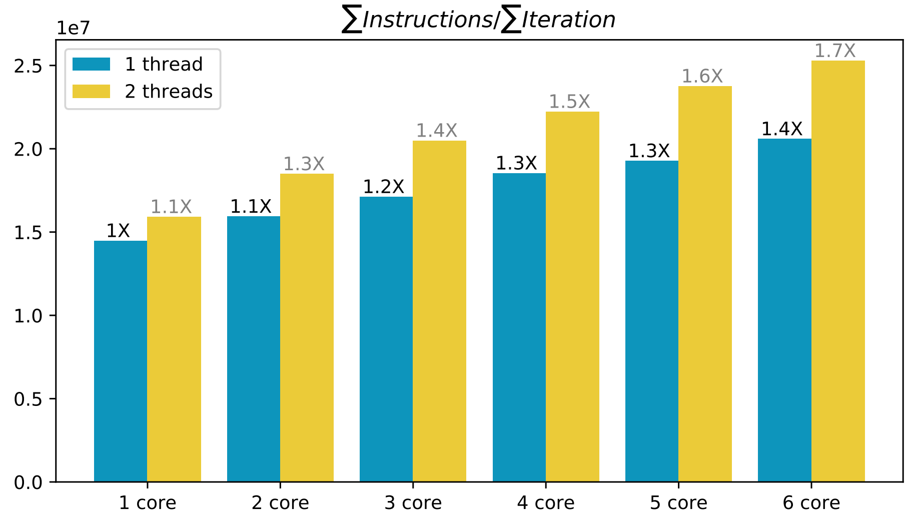


Figure 5.15: Instructions required per work unit using 1 and 2 threads per core and 1 to 6 core configurations.

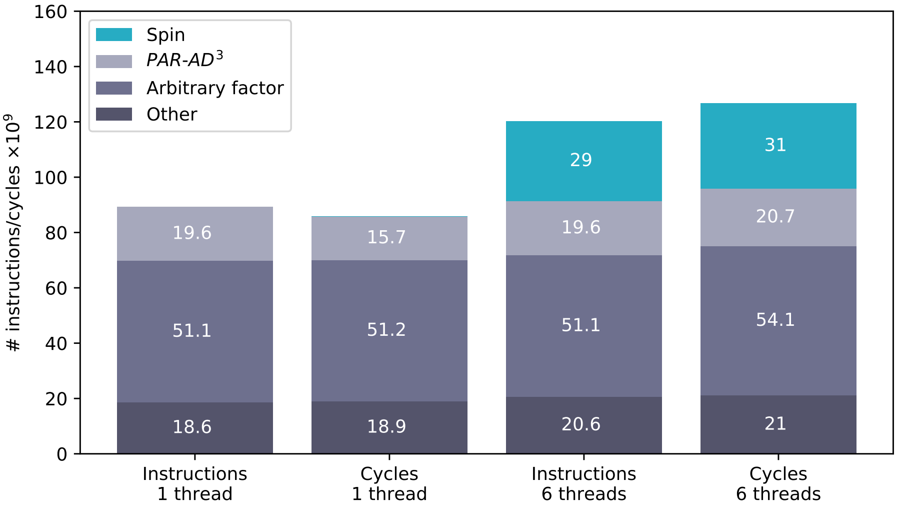


Figure 5.16: Profile of instructions executed and clock cycles consumed when solving a representative problem using 1 thread and 6 threads. Four different phases are identified along the execution.

- **Arbitrary Factor**. The processor is computing the solution for an *Arbitrary* factor.

- **Other**. The processor is doing other tasks, such as initializing vars, reading the problem from disk, communicating with the OS.

The profiling information shown in Figure 5.16 provides us with interesting information. The most obvious effect is that the growth of instructions in the parallel execution is due to spin instructions. In the 6-threaded execution, an average of around the 25% of the time is spent waiting for synchronization. We can also observe a slight growth in the number of instructions executed on the execution phase named *other*, which is due to the execution of mechanisms for handling thread parallelism, parallel data management, communication and synchronization. This is typically considered as the "overhead" of the parallel execution. Finally, we observe how the cost in cycles per instruction for the PAR-AD^3 and *Arbitrary* factor execution stages is also growing when using 6 threads instead of 1 thread. This is due to the aforementioned memory contention problems, which increase the latency of the memory accesses and decrease the IPC rate (same amount of instructions executed but they take more time to execute).

On the light of these results we can conclude that the main performance limiter is the workload balance between threads. Load balancing problems can be explained by the fact that it is not possible to know a priory which factors will require more computation. This effect is more visible in the case of solving *Arbitrary* factors, which are computationally expensive.

As future line is to study how to improve the load balancing between threads. There exist different ways to reach a better computational balance between threads but all of them have a computational overhead. Examples of methods that will lead to a better balance will be the reduction of the *chunk_size* in the parallel dynamic scheduling, or the tracking and ordering of the factors according to their load.

5.6 Conclusions

In this chapter we presented a parallel version of AD^3 that is able to solve relaxations for *Combinatorial Optimization* problems encoded as Factor Graphs using *Arbitrary* factors. In our chapter, we solve several problem instances of Side-Chain Prediction, since they are well-known hard problems and they have already been used to benchmark *Combinatorial Optimization* algorithms [Yanover et al., 2006, Sontag et al., 2012, Martins, 2012b]. The principal contribution of this chapter is the realization and evaluation of an optimized version of *Arbitrary* factors, which is a multipurpose factor. Adding *Arbitrary* factors to PAR-AD^3 widens the applicability of our parallel algorithm since it provides the capacity of solving relaxations for a larger family of *Combinatorial Optimization* problems.

Our optimized version of $PAR\text{-}AD^3$ is 2.7 times faster when running a single thread than the original AD^3; moreover, the multi-threaded version runs in parallel multicore systems and reaches a 10.8X speedup using a 6-core processor. The achievement of this speedup is due to the application of general hardware-aware strategies, that can be grouped in two categories: we provide strategies to improve the memory system performance, and also provide strategies to reduce the machine instruction count. These strategies go beyond the particularities of our algorithm and processor, and can be applied to other similar algorithms.

In this chapter we also found that the existing dataset for Side-Chain Prediction based on the SCWRL3 tool [Yanover et al., 2006] has not enough hard instances to allow us to properly analyze $PAR\text{-}AD^3$. As a consequence, we build our own dataset with the help of the SCWRL4 tool and made it available to the community in order to provide an easy way to reproduce our experiments and also to evaluate new algorithms.

Finally we provide a study of the performance and scalability limitations of our contributed $PAR\text{-}AD^3$ algorithm with *Arbitrary* factors. In a single core execution we find that our algorithm is CPU-bounded. Then we perform a scalability study where we observe that $PAR\text{-}AD^3$ is increasing performance as more cores are used. However the efficiency of the parallel execution decreases with the increase of the number of cores: the performance increase is not linear. We analyze the possible causes for this limitation and conclude that the main performance limiter is the workload balance between threads. Finally, we give hints to possible solutions to improve load balancing.

Chapter 6

Conclusions and Future Lines

In this chapter we first summarize the work developed during this dissertation in section 6.1. Then in section 6.2 we draw the most notable conclusions attained from it, and finally present some lines for future research in section 6.3.

6.1 Summary

The goal of this thesis is to investigate how to widen the applicability of *Combinatorial Optimization* algorithms that exploit the structure of the problems to solve. We do so from a computer's hardware perspective, pursuing the full exploitation of the computational resources offered by today's hardware.

Our research is motivated by the difficulty shown by recent advances in artificial intelligence to get benefit from the full power that is present in modern CPUs. We identify reasons that justify such distance between advances in AI and hardware resources exploitation:

- the complexity of nowadays hardware,
- the increasing level of abstraction of modern programming languages,
- the difficulty associated to grasping the parallel paradigm, and
- the outdated conceptual computer model used by programmers to design algorithms.

After identifying the main reasons that explain the gap between potential and effective algorithm performance, our work is centered on showing how to make algorithms that better exploit computational resources. To this end, we explore different problems with varying different features for the sake of generality:

- The *Coalition Structure Generation* problem, characterized by a small number of decision variables and a large number of linear constraints.

- The *Winner Determination Problem for Combinatorial Auctions*, characterized by a large number of decision variables subject to a large number of linear constraints.

- The *Side-Chain Prediction*, characterized by a large number of decision variables subject to a large number of declarative constraints.

We analyze state-of-the-art algorithms solving each of the three problems and we show how to improve their performance. The improvements rely on the level of achievement of three different performance enhancement goals:

- reducing the amount of low-level machine instructions to process,
- increasing the memory system throughput and
- increase potential parallelism for executing machine instructions

Coalition Structure Generation Problem

In Chapter 3 we presented our work on the Coalition Structure Generation Problem where we proposed the first optimized and parallel implementation of the IDP algorithm [Rahwan and Jennings, 2008b].

First, we identified IDP's performance limiters as (i) the low efficiency of the splittings enumeration strategy, (ii) the non-uniform memory access pattern and (iii) the lack of parallelism; Thereafter, we contribute with an IDP-based algorithm that overcomes the above-mentioned performance-limiting problems. In our contribution we propose the following strategies to fulfill our performance enhancement goals:

- a method for efficient enumeration of splittings;
- a compact representation of the data in memory; and
- a method to parallelize the IDP algorithm.

Our optimized and parallel version is able to obtain a well thread-balanced execution that significantly reduces the IDP running times reported in the literature: from the two months required by IDP for a problem with 27 agents in a 8-core computer to the 1.2 hours required by our algorithm. Therefore, we obtained a significant three orders of magnitude speedup.

Winner Determination Problem for Combinatorial Auctions

In Chapter 4 we faced the WDP for CA. We observed that state-of-the-art solvers effectively find the optimal solution for up to medium-sized WDPs. However, when a problem is large –having thousands of decision variables– exact methods are very time consuming and unpractical (solutions can take weeks, months or even years). In such cases LP relaxations are able to provide an approximation to the solution of the problem. We found that in the large-scale setup, the convex optimization community is actively developing algorithms able to efficiently approximate LP relaxations to large *Combinatorial Optimization* problems.

In light of recent developments, we propose to use AD^3 [Martins et al., 2014] as a method to solve LP relaxations of the WDP for large-scale CAs. To that end, we first show how to encode a WDP for CA to be solved by means of AD^3. Then we determine the scope of applicability of AD^3 using distributions generated by a standard Combinatorial Auction Generator [Leyton-Brown et al., 2000].

Our main contribution is the proposal of an AD^3-based parallel and optimized algorithm, the so-called $PAR\text{-}AD^3$. It addresses our performance enhancement goals thanks to the use of different strategies:

- Addressing problem specificity, producing a faster code but less general.
- The use of a Structure of Arrays (SoA) data representation model.
- The embracing of the edge-centric parallelization.
- The promotion of vectorization

$PAR\text{-}AD^3$ is able to solve WDPs for CAs 3 times faster than AD^3 using one thread, plus the possibility of running in parallel delivering an additional 4.8X using 6 threads in a 6 core computer. The overall result is an accumulated 12.4X average speedup in our processing resources.

We also evaluate $PAR\text{-}AD^3$ against CPLEX solving WDPs for CAs. As stated in the literature [Leyton-Brown et al., 2009], the hardness of every WDP for CA depends on its size and distribution. We determine the problem sizes and distributions that are better suited for $PAR\text{-}AD^3$. We observe that $PAR\text{-}AD^3$ is well suited for the hardest distributions having a high number of variables/constraints -thousands-, obtaining a peak speedup of 23X with respect to CPLEX. We also observe that $PAR\text{-}AD^3$ is able to scale better than CPLEX algorithms in parallel architectures.

Side-Chain Prediction (SCP)

In Chapter 5, we extended $PAR\text{-}AD^3$ algorithm to be able to cope with *Arbitrary* factors. This extension widens the applicability of $PAR\text{-}AD^3$ to a larger scope of problems, since *Arbitrary* factors are multi-purpose factors able to represent any discrete function, and hence any *Combinatorial Optimization* problem.

We benchmarked *PAR-AD*^{3} performance by solving LP relaxations for Side-Chain Prediction problems. To perform our experimentation we first used a public SCP-based dataset [Yanover et al., 2006]. Thereafter, since we needed harder instances, we built a new dataset and made it publicly available.

The main contribution in this chapter is the realization of an extended version of *PAR-AD*^{3} that includes the optimized computation of *Arbitrary* factors. The algorithm performing the optimized computation of *Arbitrary* factors achieves our performance enhancement goals by applying the following techniques:

- Improve memory management. On the one hand, we use thread-private scratch areas in order to perform partial computations, hence promoting memory locality. On the other hand, *PAR-AD*^{3} applies a more sophisticated memory allocation strategy that yields a reduction of 97.8% memory allocation calls than *AD*^{3}.

- Reduce the number of executed instructions. *PAR-AD*^{3} reduces to one third (38%) the number of instructions executed by *AD*^{3}. This is accomplished thanks to: (i) reducing abstraction in the source code; (ii) promoting vectorization; and (iii) using specialized libraries for standard algebraic operations.

PAR-AD^3 solves LP relaxations for our Side-Chain Prediction dataset 2.7 times faster than *AD*^{3} when using one thread. It also offers the possibility of running in parallel. A parallel *PAR-AD*^{3} execution in a 6-core computer delivers an average speedup of 10.8X with respect to *AD*^{3}.

6.2 Lessons Learned

We recall the challenges we proposed at the introduction of this dissertation and discuss how we approached them. When dealing with the CSG problem, we faced two challenges, the first one being the following one:

C1. *What are the algorithmic features of DP and IDP that most impact their performance in a modern processor?*

First, to address this challenge, thanks to the use of profiling tools we determined that the most critical operation used in DP/IDP is the splitting generation.

Second we observed that the IDP algorithm memory access pattern is irregular. The downsides of the irregular memory access pattern are most evidenced when the memory required to represent an instance of the program does not fit in the system's Last Level Cache (LLC). When this is the case, the average number of Cycles Per Instruction (CPI) starts growing (or the average Instructions Per Cycle, IPC, starts decreasing), with the corresponding performance degradation.

Finally, we underline that the sequential nature of DP and IDP prevents them from getting benefit from parallel architectures.

Recall now our second challenge when dealing with the CSG problem, namely:

C2. *How to enhance DP/IDP to benefit from the parallel paradigm and better exploit hardware resources?*

To address this challenge, we enhanced the execution of DP/IDP y combining different approaches.

We proposed a novel method able to split coalitions. This method is based on the reduction of the problem of splitting coalitions to an enumeration problem. Then we make use of bit-mask operations that have very efficient hardware implementation to obtain a high throughput of the splits generated. We also show how to get benefit from parallelism, and with that end, we contribute with an algorithm that assigns disjoint parts of the work to different threads, hence avoiding thread communication. Finally, the proposed a very compact data representation that minimizes the amount of data stored and transferred from memory to the CPU.

Besides the two challenges tackled for the CSG problem, we also posed two further challenges related to the WDP for combinatorial auctions. As a first challenge, we posed:

C3. *How to solve a relaxation of a combinatorial auction winner determination problem?*

We showed how to encode the WDP for CA as a Factor Graph. In our encoding, every factor enforces a constraint that ensures that only one bid attached to the factor will be part of the solution. Then we proposed to apply AD^3 [Martins et al., 2014] to find the MAP of the Factor Graph, which is equivalent to finding the LP relaxation. In our enocding all the factors of the Factor Graph implement the same function: the *AtMostOne* function.

After some experimentation, we observed that AD^3 is competitive with CPLEX, a state-of-the-art MILP solver, when solving LP relaxations for large-scale WDPs in hard distributions. We also observe that although AD^3 is suitable for such problems it is prone to optimization and parallelization, and this directly drive us to the next challenge.

C4. *How to design a shared-memory parallel algorithm to solve relaxations of the Winner Determination Problem?*

We created a novel AD^3-based algorithm that is able to run in parallel in a shared-memory system, the so-called *PAR*-AD^3.

In our parallel version design, we proposed a novel way of representing data by promoting the edges of the underlying graphical model as active entities, hence following an edge-centric paradigm. In this paradigm we proposed to express as many operations as possible from the perspective of an edge. An edge-centric model has advantages in terms of memory representation because the morphology of an edge is fixed, hence it can be represented by static and compact memory structures. Static and compact data structures are, by design, good for the locality and performance.

We also showed how to grasp parallelization by decomposing the AD^3 algorithm in a set of stages that contain parallelizable operations. Inside every stage the workload is divided into the available threads. Threads solve their part of work concurrently and then they wait for the last thread to finish in order to synchronize.

Since *PAR*-AD^3 , as designed in Chapter 4, only allows to solve problems whose encoding only uses *AtMostOne* factors, our next challenge has aimed at widening the scope of applicability of *PAR*-AD^3. We do so by facing the following challenge:

C5. *How to design a shared-memory parallel algorithm to solve relaxations of the Side-Chain Prediction problem?*

In the literature, we find that AD^3 has already been proposed to solve relaxations of the Side-Chain Prediction (SCP) problem [Martins, 2012b]. We observe that the resolution of the relaxation for the SCP requires the use of factors implementing *Arbitrary* functions.

To fulfill this challenge we extended the *PAR*-AD^3 algorithm, by incorporating the functionality of solving *Arbitrary* factors. Then we devised several strategies to improve the performance of the algorithm when solving *Arbitrary* factors. Our strategies can be grouped in two categories. On the one hand, the strategies that pursue an improvement in the management of the memory system. On the other hand, the strategies that pursue the reduction of the number of machine instructions executed. We applied both strategies to the resolution of the *Arbitrary* factors with satisfactory results.

Our *PAR*-AD^3 enjoys the benefits of parallelization as an outcome of our contribution to challenge C4. Furthermore, it adds the optimized operation of the computation performed by *Arbitrary* factors.

We also analyzed the algorithm performance limiters. After a detailed analysis we conclude that *PAR-AD*^{3} is a CPU-bounded algorithm. We also explore the parallel scalability performance and find that the algorithm is able to scale with the number of cores and that its main scalability limiter factor is the differences in thread's load balance.

Finally learned that besides the specific details of implementation, the strategies we follow that satisfy our performance enhancement goals (reduce machine level instructions, improve memory system performance, embrace parallelism) can lead to significant improvements in similar algorithms.

6.3 Future Work

The research carried out in this dissertation opens paths to several future lines.

We find that our work on the efficient and parallel version of IDP could be extended in the following directions:

- **Improve locality.** We propose to represent the data in a very compact way. However when the problem does not fit in the memory cache system we observe a degradation of the algorithm performance. Remains open the study of a more convenient data representation in memory that helps to improve the locality of memory access without sacrificing compactness. A memory access pattern with more locality may potentially yield to better performance.

- **Improve performance in NUMA systems.** An open line is to improve the performance of our algorithm in a NUMA system. An intelligent allocation of data in the different per-CPU memory systems together with a proper division of the work across the different CPUs available, could lead to an increase of performance of the algorithm in a NUMA system execution.

- **Benchmark against ODP-IP** After presenting our contributions in [Cruz-Mencía et al., 2013], ODP-IP [Michalak et al., 2016] was developed. ODP-IP combines two algorithms –ODP and IP–. The ODP part stands for Optimal DP, an improved version of IDP that guarantees the optimal solution exploring less nodes than IDP. ODP also makes use lexicographic ordering as well as fast split bit-mask operations as they were suggested in [Cruz-Mencía et al., 2013]. It remains an open line to measure the performance of our algorithm against ODP-IP using the same hardware as well as study the ways of parallelizing ODP-IP.

- **Apply techniques to other algorithms.** We can explore the application of the performance enhancement techniques employed in IDP, like the Fast Split or the work division between threads, to other CSG algorithms in the IDP family such as IP-IDP [Rahwan and Jennings, 2008a].

The cases of the *Winner Determination Problem for Combinatorial Auctions* and *the Side-Chain Prediction* problems are addressed with *PAR-AD*^{3}. Future lines that remain open for *PAR-AD*^{3} are described next.

- **Encode more factors.** *PAR-AD*^{3} encodes two different factors: *AtMostOne* and *Arbitrary*. However, in the literature and also in the original *AD*^{3} description, we find factors that could be added to *PAR-AD*^{3}. Although the *Arbitrary* is able to represent any function through enumeration of cases, adding new factors will provide a more specialized code to solve specific problems. Hence the development of specialized factors are a natural way of extending *PAR-AD*^{3}.

- **Integrate other algorithms.** We also know that there are other algorithms belonging to the same family as *AD*^{3} that have a very similar structure, such as ADMM [Gabay and Mercier, 1976] or MPLP [Globerson and Jaakkola, 2008] to name a few. An interesting future line will be to reuse the performance enhancement techniques in the conception of *PAR-AD*^{3} (e.g. following the edge-centric model, reusing the stage definition, or the memory representation) to produce the parallel counterparts of state-of-the-art message-passing algorithms for graphical models.

- ***PAR-AD*^{3} autotune.** We let unexplored the parametrization of *AD*^{3}. In our experimentation we have manually tuned *AD*^{3} parameters. However it is still unexplored how to adjust these parameters to improve in the resolution of a problem in an automated way using state-of-the-art techniques for the automated configuration of optimization algorithms (e.g. iRACE [López-Ibáñez et al., 2016]).

- **Improve parallel load balance.** We discovered that an improvement in *PAR-AD*^{3} load balance would entail an improvement in the scalability of the parallel algorithm. Exploring the ways of improving load balancing remains as a future line.

- **Evaluate in a more parallel environment.** *PAR-AD*^{3} should be a good algorithm for processors with many cores but its computational pattern –such as the unbalanced computational load exhibited by the subproblems– make it unappropriated for a GPU-style hardware. However current trends in Computer Architecture (i.e. increasing number of cores per CPU) make $PAR\text{-}AD^3$ a promising algorithm to be run on top of future multicore developments. At the time of the writing there are processors in the market with more than 20 cores –Intel Xeon E7-8894 has currently 24 cores–. Therefore, empirically quantifying the actual speed-ups of $PAR\text{-}AD^3$ over such architectures remains a subject of future research.

Bibliography

[Achterberg, 2009] Achterberg, T. (2009). SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41.

[Aguiar et al., 2011] Aguiar, P., Xing, E. P., Figueiredo, M., Smith, N. A., and Martins, A. (2011). An augmented lagrangian approach to constrained map inference. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 169–176.

[Andersson et al., 2000] Andersson, A., Tenhunen, M., and Ygge, F. (2000). Integer programming for combinatorial auction winner determination. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 39–46. IEEE.

[Ausiello et al., 2012] Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. (2012). *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media.

[Ball, 2011] Ball, M. O. (2011). Heuristics based on mathematical programming. *Surveys in Operations Research and Management Science*, 16(1):21–38.

[Bellman, 1954] Bellman, R. (1954). The theory of dynamic programming. Technical report, RAND CORP SANTA MONICA CA.

[Bernhard and Vygen, 2008] Bernhard, K. and Vygen, J. (2008). Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.*

[Bertsimas and Tsitsiklis, 1997] Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific, 1st edition.

[Bower et al., 1997] Bower, M. J., Cohen, F. E., and Dunbrack Jr, R. L. (1997). Prediction of protein side-chain rotamers from a backbone-dependent rotamer library: a new homology modeling tool1. *Journal of molecular biology*, 267(5):1268–1282.

[Boyd et al., 2011] Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122.

[Boyd and Vandenberghe, 2004] Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

[Canutescu et al., 2003] Canutescu, A. A., Shelenkov, A. A., and Dunbrack, R. L. (2003). A graph-theory algorithm for rapid protein side-chain prediction. *Protein science*, 12(9):2001–2014.

[Chan et al., 2017] Chan, S. H., Wang, X., and Elgendy, O. A. (2017). Plug-and-play admm for image restoration: Fixed-point convergence and applications. *IEEE Transactions on Computational Imaging*, 3(1):84–98.

[Cramton et al., 2006] Cramton, P., Shoham, Y., and Steinberg, R. (2006). *Combinatorial auctions*. MIT Press.

[Cruz-Mencía et al., 2013] Cruz-Mencía, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2013). Optimizing performance for coalition structure generation problems' IDP algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 706. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), July 22-25, las Vegas, Nevada.

[Cruz-Mencia et al., 2015a] Cruz-Mencia, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2015a). Parallelisation and application of AD^3 as a method for solving large scale combinatorial auctions. In *International Conference on Coordination Languages and Models*, pages 153–168. Springer.

[Cruz-Mencia et al., 2015b] Cruz-Mencia, F., Cerquides, J., Espinosa, A., Moure, J. C., and Rodriguez-Aguilar, J. A. (2015b). Paving the way for large-scale combinatorial auctions. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1855–1856. International Foundation for Autonomous Agents and Multiagent Systems.

[Cruz-Mencia et al., 2017] Cruz-Mencia, F., Espinosa, A., Moure, J. C., Cerquides, J., Rodriguez-Aguilar, J. A., Svensson, K., and Ramchurn, S. D. (2017). Coalition structure generation problems: optimization and parallelization of the IDP algorithm in multicore systems. *Concurrency and Computation: Practice and Experience*, 29(5).

[Cruz-Mencía et al., 2013] Cruz-Mencía, F., Cerquides, J., Espinosa, T., Moure, J. C., Ramchurn, S. D., and Rodriguez-Aguilar., J. A. (2013). Coalition structure generation problems: optimization and parallelization of the IDP algorithm. In *Proceedints of the 6th International Workshop on Optimisation in Multi-Agent Systems, OPTMAS 2013, workshop affiliated with AAMAS 2013.*

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[De Vries and Vohra, 2003] De Vries, S. and Vohra, R. V. (2003). Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309.

[Desmet et al., 1992] Desmet, J., De Maeyer, M., Hazes, B., and Lasters, I. (1992). The dead-end elimination theorem and its use in protein side-chain positioning. *Nature*, 356(6369):539.

[Duchi et al., 2008] Duchi, J., Shalev-Shwartz, S., Singer, Y., and Chandra, T. (2008). Efficient projections onto the l 1-ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM.

[Dunbrack, 2002] Dunbrack, R. L. (2002). Rotamer libraries in the 21st century. *Current Opinion in Structural Biology*, 12(4):431 – 440.

[Dunbrack Jr, 1999] Dunbrack Jr, R. L. (1999). Comparative modeling of casp3 targets using psi-blast and scwrl. *Proteins: Structure, Function, and Bioinformatics*, 37(S3):81–87.

[Dunbrack Jr and Karplus, 1993] Dunbrack Jr, R. L. and Karplus, M. (1993). Backbone-dependent rotamer library for proteins application to side-chain prediction. *Journal of molecular biology*, 230(2):543–574.

[Eckstein and Bertsekas, 1992] Eckstein, J. and Bertsekas, D. P. (1992). On the douglas?rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(1-3):293–318.

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960.

[Forum, 1993] Forum, T. M. (1993). Mpi: A message passing interface.

[Fraenkel, 1997] Fraenkel, A. S. (1997). Protein folding, spin glass and computational complexity. In *DNA Based Computers*, pages 101–122. Citeseer.

[Fujishima et al., 1999] Fujishima, Y., Leyton-Brown, K., and Shoham, Y. (1999). Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 548–553.

[Gabay and Mercier, 1976] Gabay, D. and Mercier, B. (1976). A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1):17–40.

[Globerson and Jaakkola, 2008] Globerson, A. and Jaakkola, T. S. (2008). Fixing max-product: Convergent message passing algorithms for map lp-relaxations. In *Advances in neural information processing systems*, pages 553–560.

[Glowinski and Marroco, 1975] Glowinski, R. and Marroco, A. (1975). Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 9(R2):41–76.

[Guide, 2011] Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2.

[Hammer et al., 2000] Hammer, P., Johnson, E., and Korte, B. (2000). Conclusive rermarks. *Discrete Optimization II, Annals of Discrete Mathematics*, 5:427–453.

[Han and Poor, 2009] Han, Z. and Poor, H. V. (2009). Coalition games with cooperative transmission: a cure for the curse of boundary nodes in selfish packet-forwarding wireless networks. *IEEE Transactions on Communications*, 57(1):203–213.

[Hazan and Shashua, 2010] Hazan, T. and Shashua, A. (2010). Norm-product belief propagation: Primal-dual message-passing for approximate inference. *Information Theory, IEEE Transactions on*, 56(12):6294–6316.

[Hennessy and Patterson, 2011] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[Hestenes, 1969] Hestenes, M. R. (1969). Multiplier and gradient methods. *Journal of optimization theory and applications*, 4(5):303–320.

[Holm and Sander, 1991] Holm, L. and Sander, C. (1991). Database algorithm for generating protein backbone and side-chain co-ordinates from a cα trace: application to model building and detection of co-ordinate errors. *Journal of molecular biology*, 218(1):183–194.

[Horling and Lesser, 2004] Horling, B. and Lesser, V. (2004). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316.

[Hwang and Liao, 1995] Hwang, J.-K. and Liao, W.-F. (1995). Side-chain prediction by neural networks and simulated annealing optimization. *Protein Engineering, Design and Selection*, 8(4):363–370.

[Ibm, 2011] Ibm (2011). *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual.*

[Ihler et al., 2005] Ihler, A. T., John III, W. F., and Willsky, A. S. (2005). Loopy belief propagation: Convergence and effects of message errors. *Journal of Machine Learning Research*, 6(May):905–936.

[Kann, 2009] Kann, V. (2009). Complexity and approximation. urlhttp://www.nada.kth.se/ viggo/approxbook/.

[Koehl and Delarue, 1995] Koehl, P. and Delarue, M. (1995). A self consistent mean field approach to simultaneous gap closure and side-chain positioning in homology modelling. *Nature Structural and Molecular Biology*, 2(2):163.

[Kolmogorov, 2006] Kolmogorov, V. (2006). Convergent tree-reweighted message passing for energy minimization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(10):1568–1583.

[Komodakis et al., 2007] Komodakis, N., Paragios, N., and Tziritas, G. (2007). Mrf optimization via dual decomposition: Message-passing revisited. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE.

[Kono et al., 1994] Kono, H. et al. (1994). Energy minimization method using automata network for sequence and side-chain conformation prediction from given backbone geometry. *Proteins: Structure, Function, and Bioinformatics*, 19(3):244–255.

[Krivov et al., 2009] Krivov, G. G., Shapovalov, M. V., and Dunbrack, R. L. (2009). Improved prediction of protein side-chain conformations with scwrl4. *Proteins: Structure, Function, and Bioinformatics*, 77(4):778–795.

[Land and Doig, 1960] Land, A. H. and Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520.

[Larson and Sandholm, 2000] Larson, K. S. and Sandholm, T. W. (2000). Anytime coalition structure generation: an average case study. *J. of Experimental & Theoretical Artificial Intelligence*, 12(1):23–42.

[Laughton, 1994] Laughton, C. (1994). Prediction of protein side-chain conformations from local three-dimensional homology relationships. *Journal of molecular biology*, 235(3):1088–1097.

[Lee and Subbiah, 1991] Lee, C. and Subbiah, S. (1991). Prediction of protein side-chain conformation by packing optimization. *Journal of molecular biology*, 217(2):373–388.

[Lehmann et al., 2006] Lehmann, D., Müller, R., and Sandholm, T. (2006). The winner determination problem. *Combinatorial auctions*, pages 297–318.

[Leyton-Brown et al., 2009] Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)*, 56(4):22.

[Leyton-Brown et al., 2000] Leyton-Brown, K., Pearson, M., and Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM.

[Liang and Grishin, 2002] Liang, S. and Grishin, N. V. (2002). Side-chain modeling with an optimized scoring function. *Protein Science*, 11(2):322–331.

[Lin, 1975] Lin, C.-H. (1975). *Corporate tax structures and a special class of set partitioning problems*. PhD thesis, Case Western Reserve University.

[Loeliger, 2004] Loeliger, H.-A. (2004). An introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41.

[López-Ibáñez et al., 2016] López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., and Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

[Loughry et al., 2000] Loughry, J., van Hemert, J., and Schoofs, L. (2000). Efficiently enumerating the subsets of a set. http://www.applied-math.org/subset.pdf.

[Lovell et al., 2000] Lovell, S. C., Word, J. M., Richardson, J. S., and Richardson, D. C. (2000). The penultimate rotamer library. *Proteins: Structure, Function, and Bioinformatics*, 40(3):389–408.

[Makhorin, 2000] Makhorin, A. (2000). Glpk. urlhttp://www.gnu.org/software/glpk/glpk.html.

[Martins, 2012a] Martins, A. (2012a). AD3 source code. urlhttps://github.com/andre-martins/AD3.

[Martins et al., 2015] Martins, A. F., Figueiredo, M. A., Aguiar, P. M., Smith, N. A., and Xing, E. P. (2015). Ad 3: Alternating directions dual decomposition for map inference in graphical models. *The Journal of Machine Learning Research*, 16(1):495–545.

[Martins, 2012b] Martins, A. F. T. (2012b). *The Geometry of Constrained Structured Prediction: Applications to Inference and Learning of Natural Language Syntax*. PhD thesis, Columbia University.

[Martins et al., 2014] Martins, A. F. T., Figueiredo, M. A. T., Aguiar, P. M. Q., Smith, N. A., and Xing, E. P. (2014). Ad^3: Alternating directions dual decomposition for map inference in graphical models. *Journal of Machine Learning Research*, 46. to appear.

[Mendes et al., 1999] Mendes, J., Baptista, A. M., Carrondo, M. A., and Soares, C. M. (1999). Improved modeling of side-chains in proteins with rotamer-based methods: A flexible rotamer model. *Proteins: Structure, Function, and Bioinformatics*, 37(4):530–543.

[Michalak et al., 2016] Michalak, T., Rahwan, T., Elkind, E., Wooldridge, M., and Jennings, N. R. (2016). A hybrid exact algorithm for complete set partitioning. *Artif. Intell.*, 230(C):14–50.

[Michalak et al., 2010] Michalak, T., Sroka, J., Rahwan, T., Wooldridge, M., McBurney, P., and Jennings, N. (2010). A distributed algorithm for anytime coalition structure generation. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1007–1014. International Foundation for Autonomous Agents and Multiagent Systems.

[Miksik et al., 2014] Miksik, O., Vineet, V., Perez, P., and Torr, P. H. S. (2014). Distributed non-convex admm-inference in large-scale random fields. In *British Machine Vision Conference (BMVC)*.

[Nemhauser and Wolsey, 1988] Nemhauser, G. L. and Wolsey, L. A. (1988). Integer programming and combinatorial optimization. *Wiley, Chichester. GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, 20:8–12.

[Nesterov and Nemirovskii, 1994] Nesterov, Y. and Nemirovskii, A. (1994). *Interior-point polynomial algorithms in convex programming*, volume 13. Siam.

[Optimization, 2017] Optimization, G. (2017). Gurobi optimizer. software.

[Parsons et al., 2011] Parsons, S., Rodriguez-Aguilar, J. A., and Klein, M. (2011). Auctions and bidding: A guide for computer scientists. *ACM Comput. Surv.*, 43(2):10:1–10:59.

[Paschos, 2012] Paschos, V. (2012). *Concepts of Combinatorial Optimization*. ISTE. Wiley.

[Paschos, 2013] Paschos, V. T. (2013). *Applications of combinatorial optimization*. John Wiley & Sons.

[Pierce and Winfree, 2002] Pierce, N. A. and Winfree, E. (2002). Protein design is np-hard. *Protein engineering*, 15(10):779–782.

[Rahwan and Jennings, 2008a] Rahwan, T. and Jennings, N. (2008a). Coalition structure generation: dynamic programming meets anytime optimisation. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, pages 156–161.

[Rahwan and Jennings, 2008b] Rahwan, T. and Jennings, N. R. (2008b). An improved dynamic programming algorithm for coalition structure generation. In *AAMAS (3)*, pages 1417–1420.

[Rahwan et al., 2015] Rahwan, T., Michalak, T. P., Wooldridge, M., and Jennings, N. R. (2015). Coalition structure generation: A survey. *Artificial Intelligence*, 229:139–174.

[Rahwan et al., 2009] Rahwan, T., Ramchurn, S., Jennings, N., and Giovannucci, A. (2009). An anytime algorithm for optimal coalition structure generation. *Journal of Artificial Intelligence Research*, 34(1):521–567.

[Rahwan et al., 2007] Rahwan, T., Ramchurn, S. D., Dang, V. D., Giovannucci, A., and Jennings, N. R. (2007). Anytime optimal coalition structure generation. In *AAAI*, pages 1184–1190.

[Ramchurn et al., 2009] Ramchurn, S. D., Mezzetti, C., Giovannucci, A., Rodriguez-Aguilar, J. A., Dash, R. K., and Jennings, N. R. (2009). Trust-based mechanisms for robust and efficient task allocation in the presence of execution uncertainty. *Journal of Artificial Intelligence Research*, 35(1):119.

[Ramchurn et al., 2008] Ramchurn, S. D., Rogers, A., Macarthur, K., Farinelli, A., Vytelingum, P., Vetsikas, I., and Jennings, N. R. (2008). Agent-based coordination technologies in disaster management. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*, pages 1651–1652.

[Rohl et al., 2004] Rohl, C. A., Strauss, C. E., Chivian, D., and Baker, D. (2004). Modeling structurally variable regions in homologous proteins with rosetta. *Proteins: Structure, Function, and Bioinformatics*, 55(3):656–677.

[Rothkopf et al., 1998] Rothkopf, M. H., Pekeč, A., and Harstad, R. M. (1998). Computationally manageable combinational auctions. *Management science*, 44(8):1131–1147.

[Roy et al., 2013] Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM.

[Rush and Collins, 2012] Rush, A. M. and Collins, M. (2012). A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research*, 45:305–362.

[Rush et al., 2010] Rush, A. M., Sontag, D., Collins, M., and Jaakkola, T. (2010). On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11. Association for Computational Linguistics.

[Samudrala and Moult, 1998] Samudrala, R. and Moult, J. (1998). Determinants of side chain conformational preferences in protein structures. *Protein Engineering*, 11(11):991–997.

[Sandholm et al., 1999] Sandholm, T., Larson, K., Andersson, M., Shehory, O., and Tohmé, F. (1999). Coalition structure generation with worst case guarantees. *Artificial Intelligence*, 111(1-2):209–238.

[Sandholm et al., 2001] Sandholm, T., Suri, S., Gilpin, A., and Levine, D. (2001). Cabob: A fast optimal algorithm for combinatorial auctions. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 1102–1108.

[Sandholm and Lesser, 1997] Sandholm, T. W. and Lesser, V. R. (1997). Coalitions among computationally bounded agents. *Artificial Intelligence*, 94:99–137.

[Santos Jr, 1991] Santos Jr, E. (1991). On the generation of alternative explanations with implications for belief revision. In *Proceedings of the Seventh conference on Uncertainty in Artificial Intelligence*, pages 339–347. Morgan Kaufmann Publishers Inc.

[Sheffi, 2004] Sheffi, Y. (2004). Combinatorial auctions in the procurement of transportation services. *Interfaces*, 34(4):245–252.

[Shehory and Kraus, 1998] Shehory, O. and Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artif. Intell.*, 101(1-2):165–200.

[Shoham and Leyton-Brown, 2008] Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent systems: Algorithmic, game-theoretic, and logical foundations.* Cambridge University Press.

[Sierra et al., 2001] Sierra, C., de López Màntaras, R., and Busquets, D. (2001). Multiagent bidding mechanisms for robot qualitative navigation. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 198–212. Springer.

[Smith and Eisner, 2008] Smith, D. A. and Eisner, J. (2008). Dependency parsing by belief propagation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 145–156. Association for Computational Linguistics.

[Sontag et al., 2012] Sontag, D., Meltzer, T., Globerson, A., Jaakkola, T. S., and Weiss, Y. (2012). Tightening lp relaxations for map using message passing. *arXiv preprint arXiv:1206.3288.*

[Summers and Karplus, 1989] Summers, N. L. and Karplus, M. (1989). Construction of side-chains in homology modelling: Application to the c-terminal lobe of rhizopuspepsin. *Journal of molecular biology*, 210(4):785–811.

[Sutter, 2005] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210.

[Svensson et al., 2013] Svensson, K., Ramchurn, S., Cruz-Mencía, F., Rodriguez-Aguilar, J. A., and Cerquides, J. (2013). Solving the coalition structure generation problem on a GPU. In *Proceedints of 6th International Workshop on Optimisation in Multi-Agent Systems, OPTMAS 2013, workshop affiliated with AAMAS 2013*.

[Thrall and Lucas, 1963] Thrall, R. M. and Lucas, W. F. (1963). N-person games in partition function form. *Naval Research Logistics (NRL)*, 10(1):281–298.

[Tuffery et al., 1991] Tuffery, P., Etchebest, C., Hazout, S., and Lavery, R. (1991). A new approach to the rapid determination of protein side chain conformations. *Journal of Biomolecular structure and dynamics*, 8(6):1267–1289.

[Wainwright et al., 2003] Wainwright, M. J., Jaakkola, T. S., and Willsky, A. S. (2003). Tree-reweighted belief propagation algorithms and approximate ml estimation by pseudo-moment matching. In *Workshop on Artificial Intelligence and Statistics*, volume 21, page 97. Society for Artificial Intelligence and Statistics Np.

[Wainwright et al., 2005] Wainwright, M. J., Jaakkola, T. S., and Willsky, A. S. (2005). Map estimation via agreement on trees: message-passing and linear programming. *IEEE transactions on information theory*, 51(11):3697–3717.

[Wilson et al., 1993] Wilson, C., Gregoret, L. M., and Agard, D. A. (1993). Modeling side-chain conformation for homologous proteins using an energy-based rotamer search. *Journal of molecular biology*, 229(4):996–1006.

[Wooldridge, 2009] Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons.

[Xiang and Honig, 2001] Xiang, Z. and Honig, B. (2001). Extending the accuracy limits of prediction for side-chain conformations1. *Journal of molecular biology*, 311(2):421–430.

[Yanover et al., 2006] Yanover, C., Meltzer, T., and Weiss, Y. (2006). Linear programming relaxations and belief propagation – an empirical study. *J. Mach. Learn. Res.*, 7:1887–1907.

[Yun Yeh, 1986] Yun Yeh, D. (1986). A dynamic programming approach to the complete set partitioning problem. *BIT Numerical Mathematics*, 26:467–474. 10.1007/BF01935053.