Universitat Autònoma de Barcelona

Master's Degree in High Performance Computing,
Information Theory and Security

Optimizing Performance for Coalition Structure Generation Problems in Multicore Systems

Author:
Francisco Cruz Mencia

Advisor:
Antonio Espinosa

July, 2012

Abstract (English)

The Coalition Structure Generation Problem (CSGP) is well-known in the area of MultiAgent Systems. Its goal is establishing coalitions between agents while maximizing the global welfare. Between the existing different algorithms designed to solve the CSGP, DP and IDP are the ones with less temporal complexity. After analyzing the performance of the DP and IDP algorithms, we detect which is the most frequent operation and we propose an optimized method for performing it: The Fast Split Method (FSM). Then, based on the FSM, we propose two algorithms (FSDP and FSIDP) which are up to 30 times faster than the original DP and IDP. In addition, we study the possibility of dividing the work in different threads and propose a method for doing it. Then we present a parallel version of the FSIDP Algorithm using a shared memory paradigm: SMIDP. This version enables us to reach a speedup of 182X using our two six-core processor computer.

Acknowledgements

This research would not have been possible without the support of many people. I would like to express my gratitude to both my advisor, Prof. Antonio Espinosa and to Prof. Juan Carlos Moure who offered me invaluable assistance, support and guidance.

I also want to thank Dr. Jesús Cerquides and Dr. Juan Antonio Rodríguez for giving me advice, help, unconditional support and encouragement.

Many thanks to the staff of the Electronics and Computer Science lab of the Faculty of Physical and Applied Sciences of the University of Southampton for kindly providing me with information about their developments and for being always available, especially to Dr. Sarvapali Ramchurn and to Dr. Thomas D. Voice.

I would also like to thank everyone at IIIA-CSIC, the CAOS department, and my Master's Degree colleagues.

My biggest gratitude goes to my wonderful family and to Cristina. Thank you.

Contents

Chapter 1

Introduction

1.1 Motivation

1
In a multi agent system environment, agents usually collaborate in order to reach an objective. This collaboration between agents is known as *coalition*.

Just like in real life, there are situations where collaboration is the most efficient way to reach an objective. We can find many examples of ordinary situations where *coalition* formation is applied, such as two people who want to move two big tables. The most efficient way to do this is working together, moving one table first and then the other. The two people are therefore establishing a *coalition*.

When incrementing the complexity of the system, determining which is the best combination of *coalitions* that will improve the performance of the whole system is not an easy task.

In the agents field, a *coalition* is an agreement between two or more agents which decide to collaborate in order to reach a common goal. Given a set of agents, any possible combination of *coalitions* is called Coalition Structure. The Coalition Structure Generation Problem (CSGP) [2] studies how to assess the *coalition* that maximizes the social welfare.

Besides the initial application of the CSGP, which is finding the optimal Coalition Structure for a given set of agents, there is a large number of similar applications that can be modeled using the same representation, such as distributed vehicle route planning [12], task allocation [14], airport slots allocation [10] and social networks [16], among others.

In the last two decades, different strategies have been proposed to solve the CSGP: methods based on Dynamic Programming [17, 11, 8], heuristic approaches [13, 14], and anytime algorithms [7, 9, 6, 4].

Finding a solution to the CSGP requires the exploration of an exponential number of combinations which makes the algorithm computationally demanding. It is so demanding that computation time is the main problem when solving CSGP.

This report presents the work done optimizing a method for solving the CSGP.

The objective of this work is to study how to solve CSGP faster than current state of the art approaches. We focus on a Dynamic Programming approach, the only one being both optimal and having less temporal complexity. Our results are compared with the DP[11] and IDP[8] algorithms in terms of *SpeedUp*.

In order to increase performance when solving the CSGP, we tackle the problem in two stages:

- First, stuying the number of instructions needed and propose and optimization to minimize execution time..
- Second, propose a new parallel implementation of the DP algorithm where the Fast Split Method is introduced.

1.2 Contributions

We developed a software framework able to validate any CSGP algorithm implementation functionality and performance.

We implemented DP and IDP algorithms as they are described in their articles and incorporated them to our framework. Then we measured them in order to analyze their execution pattern.

We developed a method for executing the most frequent operation found in CSGP DP implmementation in a computationally efficient way. We call it the Fast Split Method (FSM). Thanks to this, we propose two new algorithms: FSDP and FSIDP.

We describe performance results obtained with FSDP and FSIDP. Considering a problem of size 25, DP needs 16 hours and 30 minutes to find the optimal solution, while IDP needs almost 6 hours. Our FSDP and FSIDP Algorithms solve the same CSGP using the same hardware in 52 and 30 minutes respectively. FSIDP reaches a *SpeedUp* of 30X over DP for a problem of size 25.

We developed a parallel version of the algorithm in order to be run in a MultiCore system: Shared Memory IDP (SMIDP). This algorithm solves the same problem of size 25 in 3 minutes and 30 seconds using a PC with two six-core processors reaching a *SpeedUp* of 182X.

A roadmap of our contributions can be seen in Figure 1.1, which shows the relations between the different algorithm versions we develop. Our starting point is an existing version of the IDP algorithm coded in Java.

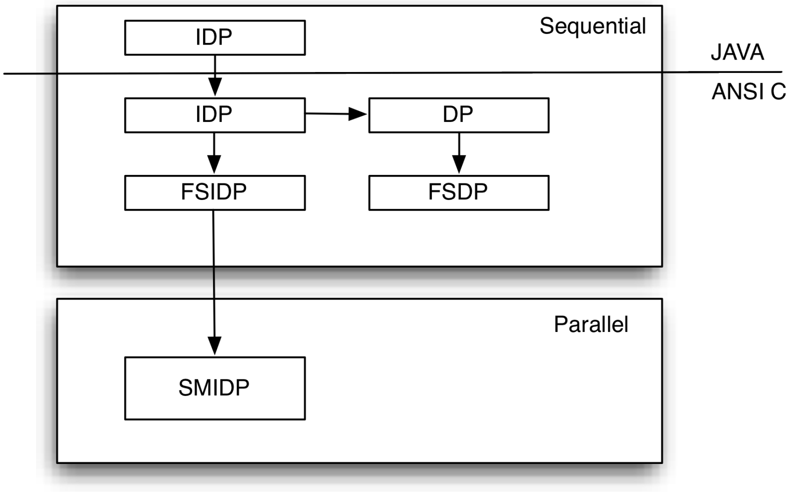

Figure 1.1: Roadmap. Different versions of the algorithm presented in this report.

1.3 Summary

This report is organized in 7 chapters.

The present chapter introduces our work and presents our contributions.

Chapter 2 introduces the reader to the state of the art of the CSGP, and it presents the different approaches that the community has taken to solve the CSGP. We focus on the Dynamic Programming strategies, which are described in detail paying special attention to the DP and IDP algorithms.

In Chapter 3 we describe the work done in order to reproduce the State of the Art developments by coding DP and IDP algorithms and measuring the results.

Chapter 4 proposes a novel method for performing a key operation which leeds to a significant performance improvement. The Fast Split Method is presented and from it we propose new algorithms based on the original DP and IDP. Our implementations are called FSDP and FSIDP. We compare the different algorithms' performance.

In Chapter 5 we analyze FSDP and FSIDP in detail in order to obtain a better characterization of their behavior.

In Chapter 6 we introduce a technique for solving CSGP in a multicore system using shared memory and openMP. Results are compared with the ones found in the sequential version.

Chapter 7 concludes the report, summarizing the experiments and presenting the conclusions and future work.

Chapter 2

State of the art

This chapter describes in detail the Coalition Structure Generation Problem (CSGP) and exposes the different state of the art approaches used for solving the problem.

We focus our attention on the Dynamic Programming family algorithms. More specifically, on the DP and IDP Algorithms, which are thoroughly analyzed.

To this end, we first introduce some basic concepts, such as *coalition* or *splitting*, then we detail how DP is able to find the optimal solution for a CSGP. We provide information of the algorithm complexity and a formula for computing the *splittings* evaluated by DP.

Finally, we present IDP as a variation of DP, an then we consider a theoretical model to study the different behavior between DP and IDP.

2.1 Problem Description

The Coalition Structure Generation Problem (CSGP) belongs to the Autonomous Agents research line in the Artificial Intelligence field, where the goal to be achieved is establishing collaborations, or *coalitions* between agents.

Agents are software entities able to communicate with other agents and establish *coalitions* in order to satisfy their own or collective goals. In fact, one of the big potentials of the agents technology is precisely the possibility to communicate and cooperate. Forming effective *coalitions* is one of the main challenges that are faced today by the Autonomous Agents community.

There are a number of factors that can affect the quality of a *coalition*, like the affinity between agents or the cost of establishing the *coalition.* Regardless of the reason that makes a *coalition* desirable or not, a value indicating the quality of this *coalition* is defined for each *coalition.* This value is called a *coalition value.*

The Coalition Formation is the process whereby a group of agents establish *coalitions* between them. Finding the group of *coalitions* that maximizes the global satisfaction is what is pursued by CSGP.

Applications for this problem can be found in real-life situations, such as distributed vehicle route planning, airport slots allocation or market analysis. All these situations have the same common factor: the explosion of possibilities given the combinatorial behavior.

In order to better illustrate the idea of a Coalition Structure, we consider a typical application for CSGP called Combinatorial Auctions. A Combinatorial Auction is a type of smart market that can be solved using the same algorithms used for solving CSGP. A Combinatorial Auction is formed by: a seller, a number of items or products to be sold and a set of offers for buying the products.

For example, a seller has 4 items to be sold :

🍴 🫖 🥂 ☕

Buyers place bids for individual or groups of products as shown in Table 2.1. The goal is to find the combination of bids which maximizes the benefit of the seller. In the example, the best combination is to accept the offers from Ann, Mark and John obtaining an income of $166.

Buyer	**Desired items**	**Offer**
Rose Marie	🍴	$33
Ann	🫖	$39
Mark	☕	$40
Laura	🍴 🫖	$87
John	🍴 🥂	$87
Nick	🫖 ☕	$52
Matt	🥂 ☕	$67
Julia	🍴 🫖 🥂	$97

Table 2.1: Offers in a Combinatorial Auction market

Note that in the example on Table 2.1 there are offers only for some combinations of items, not all. When we have a problem behaving like this, we say that the problem does not have a complete input. Being n the number of products to be sold, the number of possible combinations of elements is determined by $2^n - 1$.

There is a complete analogy between the Combinatorial Auctions and the CSGP (shown in Table 2.2). CSGP is about Agents, *Coalitions*, and *Coalition Values* instead of Items, Buyers and Offers. But the problem is numerically equivalent.

Combinatorial Auctions	**CSGP**
Items to be sold	Agents
Buyers	Coalitions
Offer	Coalition Value

Table 2.2: Analogy between Combinatorial Auctions and CSGP

2.2 CSGP Algorithms

There are different algorithms designed to solve CSGP. We can categorize them as follows:

- Dynamic Programming: Algorithms of the Dynamic Programming family [17, 8] guarantee finding the optimal solution. They explore all the search space, storing the best combinations found so far. They minimize temporal complexity when considering the exploration of all the search space. The complexity of this algorithm is $O(3^n)$.
- Heuristic approaches: These algorithms try to find a solution guided by a heuristic function [13, 14]. They do not guarantee the optimal solution. They avoid the evaluation of certain nodes, so they tend to be fast algorithms. They are useful when dealing with big sets of data.
- Anytime algorithms: These algorithms start giving an arbitrary solution to the problem and as they have time, they obtain better solutions. These algorithms explore all possible combinations, therefore they are optimal, but the time spent to obtain a solution can be very big depending on the input. The complexity of the basic algorithm and variations [3, 7] is $O(n^n)$. However, they can perform very well working with input data presenting some characteristic distribution. There is a range of improvements [7, 9, 6] done on the basis of the original idea, including a distributed version [4].

2.3 Dynamic Programming Algorithms

As mentioned in the previous section, CSGP solvers can be divided in three categories. In this work we focus our attention on the Dynamic Programming Algorithms family. We have chosen this algorithmic family due to its interesting properties, i.e. first, Dynamic Programming algorithms have the lowest temporal complexity, meaning that they are the fast algorithms in the worst case scenario; second, they are input-independent, which means that they will not perform significantly different choosing different input data; finally, we have found that the structure of these algorithms can be adapted to solve the problem in parallel.

2.3.1 Basic definitions

Before a more in-depth analysis, it is important to define some concepts required for understanding how DP works.

- The *agent set* is a collection containing all the agents participating in the CSGP.

  For example :
  $A = \{a_1, a_2, a_3, a_4, a_5\}$
  where a_1, a_2, a_3, a_4 and a_5 are all the agents in the system.

- A *subset* is a collection of agents contained in the agent set.

  For example :
  $B = \{a_1, a_4, a_5\}$ is a subset of A

- Given a subset with more than one agent, a *split* can be performed. The *split* operation will generate a *splitting*.

  For example :
  The splitting $\{\{a_1\}, \{a_4, a_5\}\}$ is generated performing a split on $\{a_1, a_4, a_5\}$.

2.3.2 Description of the algorithm (DP)

As we have seen before, Dynamic Programming algorithms guarantee an optimal solution with less temporal complexity. In this section we describe how a particular Dynamic Programming algorithm works, specifically the one by D. Y. Yeh [17], henceforth referred to DP.

The DP algorithm evaluates all possible *coalitions* by first evaluating those with less elements and increasing the number of elements as the execution progresses. When computing a given *coalition*, it evaluates whether it is better to split the *coalition* or keep it as is.

Consider a group of 4 agents $A = \{a_1, a_2, a_3, a_4\}$ that want to achieve an objective together. Some agents have more affinity with some others when establishing a collaboration; this affinity expresses how well these group of agents perform the problem together. This value is known and it is given as the input for the problem: it is called *coalition value* and it is usually expressed as v.

For example, the *coalition value* of $\{a_1\}$ is written as $v(\{a_1\})$ and it is equal to 33, whereas the *coalition value* for the *coalition* of a_1 with a_2 is represented by $v(\{a_1, a_2\})$ and it is equal to 87. One possible input for a problem with 4 agents is presented in Table 2.3, where all possible *coalitions* with their *coalition values* are shown.

Two agents will be likely to collaborate or to establish a *coalition* if the sum of their *coalition values* is lower than the *coalition value* of the two of them together.

Coalition	**Coalition Value**
$\{a_1\}$	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 2.3: Sample input data for a problem of size 4.

The DP algorithm proceeds by evaluating all the possible combinations. In our example it first considers all combinations of size 2; in a second step, it considers all the combinations of size 3; finally, it considers all combinations of size 4.

Saying that the algorithm considers all the combinations of a certain size means that for every combination the algorithm evaluates all possible splittings in two groups and will compare the best splitting with the value of the *coalition*.

For instance, at some point, the algorithm will evaluate the *coalition* $\{a_2, a_3, a_4\}$ for the problem shown in Table 2.3, by evaluating the values of all its *splittings*:

$$\begin{gathered} v(\{a_2, a_3\}) + v(\{a_4\}) = 76; \\ v(\{a_2, a_4\}) + v(\{a_3\}) = 65; \\ v(\{a_3, a_4\}) + v(\{a_2\}) = 106 \end{gathered}$$

By comparing the best value to the value of the *coalition* $v(\{a_2, a_3, a_4\}) = 132$, since 132 is greater than 106 (the best *coalition value* of the splittings), the formation of the *coalition* will be desirable.

DP perform these operations for every possible combination of agents. To this end, all the combinations are evaluated. The strategy followed by DP is exploring all the possible values by using three nested loops.

- **Coalition size selection** (outer loop). This loop sets how many elements are considered at each iteration (m). After an iteration is done all *coalitions* of at most size m are evaluated.
- **Coalition generation** (middle loop). This loop is responsible for generating all the possible ways of choosing m elements from the *element set*. At each iteration the loop will produce one *coalition* to be evaluated.
- **Coalition evaluation** (internal loop). For each *coalition* chosen in the middle loop, the internal loop computes all the possible ways to split it in two groups. For each *splitting*, the algorithm computes whether it is better to split the *coalition* or keep it as it is.

After every single iteration in the most internal loop, DP determines if it is better to split a *coalition* or not. That amounts to computing the maximum between the value of the *coalition* without splitting versus the sum of the values of the two splits of the *coalition*. It stores this maximum value and the Coalition Structure generated in memory, therefore two tables are needed: one for storing maximum values and another for storing Coalition Structures.

However, Rahwan & Jennings demonstrated in [8] that it is not needed to maintain a table for storing Coalition Structures. The memory required for storing the information about the Coalition Structures found so far can be avoided, releasing the 33% of the memory required by DP. Once a solution is found, it is possible to compute the Coalition Structure providing that solution.

This calculation can be performed after finding the value of the Coalition Structure by adding some extra computation which is insignificant compared to the computation time took by the whole problem.

The pseudocode of the DP algorithm is presented in Algorithhm 1.

Algorithm 1 Pseudo-code of the Dynamic Programming Algorithm

```
for $m \leftarrow [2 \dots n]$ do
    for $coalition \leftarrow coalitionsOfSize(m)$ do
        $max\_value \leftarrow value[coalition]$
        $(S1, S2) \leftarrow init()$
        for $(S1, S2) \leftarrow nextSplit(coalition, (S1, S2))$ do
            if $(max\_value < value[S1] + value[S2]$ then
                $max\_value \leftarrow value[S1] + value[S2]$
            end if
        end for
        $value[coalition] \leftarrow max\_value$
    end for
end for
```

Table 2.4 presents a trace of the execution of the DP algorithm with the input data presented in Table 2.3. The first, second, and fourth columns represent the value of the three nested loops, the third and fifth columns represent the data that is evaluated on every iteration, and the last column shows the final value stored which is the maximum value between the two previous columns.

Let us focus on Table 2.4, taking a look at the rows with m=3. Note that the stored value is always done in the *coalition* of size 3, but the information needed to make this writing is either the overwritten value or the information calculated with m=2. In fact, for every value of m, the information needed is calculated in previous steps. This fact will guarantee us that when we consider any split of a *coalition*, it is certainly the best possible value for the given elements.

2.3.3 Complexity

The DP algorithm receives as input the values for all possible combinations of a given set of n elements. As we saw before, it requires storing $2^n - 1$ positions. This memory is rewritten as the algorithm proceeds and no additional memory is required. Therefore the spatial complexity of the algorithm is determined only by the input data size and it is equal to $O(2^n)$.

However, the temporal complexity is determined by the algorithm behavior and needs more careful analysis. Since the three nested loops in Algorithm 1 are bounded, we can compute the work effectively done inside every loop. The temporal complexity is $O(3^n)$, according to [8].

In fact, the number of iterations inside the internal loop can be determined by having the bounds of every loop:

- bound for the external loop: $m = n - 1$
- bound for the middle loop : $\binom{n}{m}$
- bound for the internal loop: $2^{m-1} - 1$

therefore the total number of iterations executed in the internal loop is defined by the expression:

$$\sum_{m=2}^{n} \binom{n}{m} 2^{m-1} - 1,$$

which can be rewritten as

$$\frac{3^n - 2^{n+1} + 2}{2}. \tag{2.1}$$

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 2.4: Trace of execution of a problem of size 4.

2.3.4 The Improved Version (IDP)

In [8] Rawal & Jennings presented an improved version of the DP Algorithm. The Improved Dynamic Programming Algorithm (IDP) proposes two different improvements:

- A technique to save memory.
- A technique to skip the evaluation of some nodes.

The original DP algorithm uses three matrices in order to represent different information: one for storing the initial values, another for storing calculated values, and a third one to store the configuration of the *coalitions*. In IDP we can see that all the information can be represented using only one table. This table will be the input data and it will be updated until all the search space is explored. This technique made the algorithm use only a 33.3% of the memory resources. On balance, the algorithm needs to make an extra computation once the search space is explored, but this computation is not relevant since it has a small complexity $O(2n)$.

The other tecnique is the evaluation avoidance of some nodes, at it is based on the fact that DP explores the search space doing some redundant calculation. More specifically, let A be a set of elements, and F_x be the splitting function. We have two possible splitings of A determined by $F_1(A) = \{S_1, S_2\}$ and $F_2(A) = \{S_3, S_4\}$ where $S_1, S_2, S_3, S_4 \neq \varnothing$ and $S_1, S_2 \neq S_3; S_1, S_2 \neq S_4$. IDP considers that there is a split of $F_x(\{S_1, S_2\})$ that leads us to the same result as $F_x(\{S_3, S_4\})$. Therefore, the same result can be reached through different paths.

IDP proposes a method to avoid the paths that will lead us to a state that can be reached by other paths. This method evaluates only the splittings of certain sets of data. In other words, IDP reaches the same result evaluating less combinations. According to their results, IDPonly needs 38,7% of DP operations to get the same results.

Algorithm 2 Pseudo-code of the IDP Algorithm

```
for $m \leftarrow [2 \dots n]$ do
    for $coalition \leftarrow coalitionsOfSize(m)$ do
        $max\_value \leftarrow value[coalition]$
        $(S1, S2) \leftarrow init()$
        for $(S1, S2) \leftarrow nextSplit(coalition, (S1, S2))$ do
            if $(sizeOf(S1) \geq n - m)$ then
                if $(max\_value < value[S1] + value[S2]$ then
                    $max\_value \leftarrow value[S1] + value[S2]$
                end if
            end if
        end for
        $value[coalition] \leftarrow max\_value$
    end for
end for
```

2.3.5 Where does IDP improve DP?

Figure 2.1 presents a scheme of how the DP and IDP algorithms execute the internal loop. t_1 represents the cost of generating and evaluating a single splitting for a *coalition*, t_2 stands for the cost of the operation of evaluating if the splitting is evaluated and t_3 represents the cost evaluating a *coalition* in IDP.

The basic idea behind IDP is to add a small computation overhead in order to save operations.

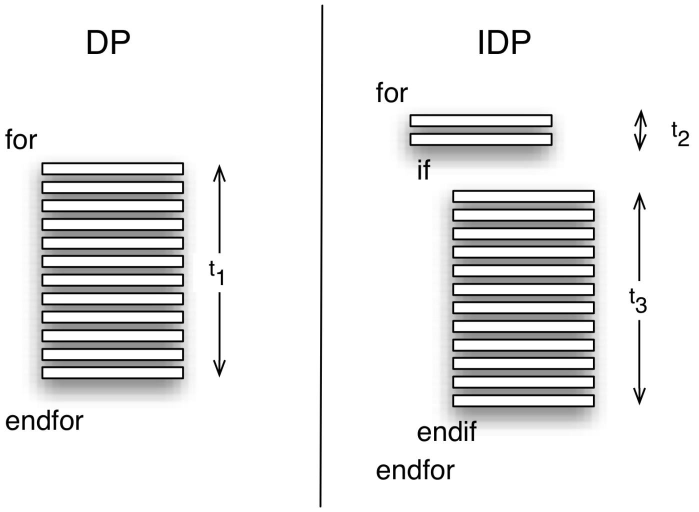

Figure 2.1: Scheme of the internal loop execution for DP, IDP, FSDP and FSIDP.

If fact, it is possible to define when IDP improves DP. We define p as the probability of entering in the conditional. With this, we can conclude that IDP will improve DP while the following in-equation is satisfied:

$$p < \frac{t_1 - t_2}{t_3}$$

Ideally, t_2 is small and t_1 and t_3 have to be similar, meaning that IDP is likely to improve DP since the fraction is close to one. This model is revised and updated as the algorithms are analyzed, and a revision for this model is presented in section 4.4.2.

Chapter 3

Coding DP and IDP

In the previous chapter, different algorithms to solve the CSGP were presented putting the stress on the DP and IDP algorithms.

In the present chapter we evaluate and measure the existing DP and IDP algorithms and also discuss the need to migrate the codification of the IDP algorithm from JAVA to ANSI C.

Since we need to work with different algorithmic implementations, we propose a software platform able to host and manage CSGPs. This enable us to run different versions and compare results.

This platform will be hosting the different implementations we propose along this report.

This chapter concludes with a section dedicated to experiments, where the execution pattern of the DP and IDP algorithms is widely measured and characterized.

3.1 Rewriting existing algorithms

From the different algorithms of the Dynamic Programming family, the one which presents less complexity is IDP. This algorithm is written in JAVA.

Our objective is improving DP and IDP performance as much as we can. To this end, we have rewritten the existing IDP Java implementation into ANSI C.

There are a number of reasons for having chosen ANSI C as our working programming language, but we can highlight:

- Using ANSI C allows us to access low-level instructions.
- We are interested in parallelism. There are some important developments for ANSI C which provide APIS in order to use different parallel programming paradigms.

- Migrating a program from JAVA to ANSI C can be done relatively fast.

One of the characteristics of JAVA is automatic memory management provided by the Garbage Collector. Again, this can be very helpful for the programmer but delegates more responsibility to the Virtual Machine.

During the migration to ANSI C, the memory management migration is the biggest source of errors. The automatic management provided by Java has to be implemented and defined explicitly in ANSI C. For this purpose, we have used the Valgrind [5] profiling tool, which is very useful for detecting memory leaks and segmentation faults.

Moreover, coding the program into ANSI C implies decoupling all the classes programmed in Java in the corresponding variables and functions. As a consequence, this codification is not a blind process, but implies a knowledge of what the algorithm is doing.

IDP is a more restrictive version of DP. It is possible to extract the DP algorithm from the IDP code by removing some constraints from the original code. Figure 3.1 shows the different versions implemented.

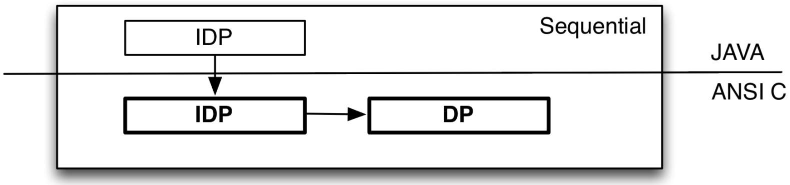

Figure 3.1: Rewriting IDP Java version to IDP and DP.

3.2 A framework for running experiments

Along this work, different versions for the DP and IDP algorithms are proposed, as we want to measure the behavior of the different implementations. These measurements have to be taken in fair conditions, that is to say, different algorithms must be able to solve the same problem given the same input data.

To this end, we develop a solver, which has different functionalities.

- Generate a CSGP Dynamically.
- Write a CSGP to disc.
- Read a CSGP from disc.
- Solve a CSGP selecting the algorithm used to solve the problem (algorithms can be easily added).

- Generate execution statistics, such as execution time or hardware counters.

Our solver can be called from the command line where different arguments can be specified, such as: the size of the CSGP to be solved, the location of the input data (either dynamically generated or read from disk) or the algorithm used to solve the CSGP with additional parameters.

The solver computes the solution using the specified algorithm and also gives statistics of use, such as the real time spent. In addition, the solver can be compiled with access to hardware counters which provide a wide range of CPU performance statistics. We use the Performance Application Programming Interface (PAPI) [1].

3.3 Measuring DP and IDP

One of the most important decisions of this work is the appropriate measurement of the application performance. Having good metrics allows doing a better analysis of what the application does and how it can be optimized

We characterize the behavior of the application using tools like the GNU Profiler or metrics such as total execution time, cycles per instruction, or number of executed instructions per read operation.

Our experiments have been performed on a computer with two six-core Intel Xeon E5645 Processors at 2.40GHz & 96GB RAM. Each processor has three different cache levels:

- L1 cache: 32KB for data and 32KB for instructions
- L2 cache: 256KB for each cores
- L3 cache: 12MB shared by all cores

The server runs a CentOS 6.2 Linux (kernel 2.6.32) OS and programs are compiled with gcc 4.7.0 enabling the -O3 option (optimize code).

3.3.1 Analyzing the impact of the programming language

The first measure taken is the improvement produced by reprogramming the IDP algorithm in ANSI C. The same program running in the same machine performs around 5% faster by only translating the program in ANSI C.

3.3.2 Profiling DP and IDP

Once we have obtained measures referring to the total time the application needs in order to solve the algorithm, profiling can provide us with more information about where the application can be optimized. We use the GNU profiler to identify the portions of the code that are executed the most. The experiment is performed running a CSGP of size $n = 19$

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.92     31.84    31.84   524287     0.00     0.00  evalSplits
  0.13     31.88     0.04   524287     0.00     0.00  CombinatorialIterator_nextSet
  0.03     31.89     0.01        1     0.01    31.89  runDP_original
  0.00     31.89     0.00       19     0.00     0.00  CombinatorialIterator_firstSet
  0.00     31.89     0.00        3     0.00     0.00  IDPprintCoalition
  0.00     31.89     0.00        2     0.00     0.00  findCS_DP
  0.00     31.89     0.00        2     0.00     0.00  sampleTime
  0.00     31.89     0.00        1     0.00     0.00  CombinatorialIterator_init
  0.00     31.89     0.00        1     0.00     0.00  GenerateProblem
  0.00     31.89     0.00        1     0.00     0.00  check_params
  0.00     31.89     0.00        1     0.00     0.00  findSolution
```

Figure 3.2: Output of the GNU profiler

Figure 3.2 shows a portion of the output provided by the GNU profiler utility, where 99.9% of the execution is done inside a function called `evalSplits`. As a consequence, we determine that the best option to achieve a significant improvement is to optimizing the function `evalSplits`. Next chapter presents a complete analysis of the split mechanisms and how to optimize them.

3.3.3 Measuring the number of iterations

	DP	IDP	p
17	64439010	32285040	0.501
18	193448101	96855121	0.501
19	580606446	290565366	0.500
20	1742343625	871696099	0.500
21	5228079450	2615088300	0.500
22	15686335501	7845264901	0.500
23	47063200806	23535794706	0.500
24	141197991025	70607384119	0.500
25	423610750290	211822152360	0.500
26	1270865805301	635466457081	0.500

Table 3.1: Number of coalitions evaluated by DP and IDP.

We introduce counters in the algorithms in order to measure experimentally the number of times that a *splitting* is evaluated. The results are presented in Table

3.1.

We verify what our analytical findings. The number of iterations of the internal loop determines the number of *splittings* evaluated and can be computed using a formula. Moreover, we find that IDP evaluates approximately half of them, and when incrementing the CSGP size, this value gets closer to 0.5.

We define p as the probability of evaluating a *splitting* by the IDP. We use this value along this report, and for the sake of simplification we approximate this p to 0.5 regardless of the problem size. Thus we consider that DP evaluates twice of the *splittings* evaluated by IDP.

3.3.4 Verifying DP-IDP performance.

Our next experiment verifies that IDP effectively improves the performance of DP. The DP and IDP algorithms are executed varying the problem size from a range of [17..26]. As shown in Table 3.2, we register an average improvement of 3.12 with a low standard deviation (0.13), which means that IDP improves DP regardless of the size of the problem.

n	17	18	19	20	21	22	23	24	25	26
DP	3.21	10.04	31.54	98.41	306.63	1048.36	4109.69	15642.50	59684.42	220278.64
IDP	1.09	3.27	9.97	29.67	96.38	343.00	1354.53	5169.44	19168.2	71519.04
SpeedUp	2.94	3.07	3.16	3.32	3.18	3.06	3.03	3.02	3.11	3.08

Table 3.2: Time spent by DP and IDP for solving problems of size n (in seconds)

This *SpeedUp* of 3x confirms what we found in the literature [8], so we are able to reproduce the results of the state of the art approaches.

In addition, the problem has a predictable behavior, since when incrementing the problem size in one, the time is multiplied approximately by three plus an overhead. These values are presented in Table 3.3.

This overhead starts to be significant when the problem grows from $n = 21$ to $n = 22$. In the following chapters we analyze why this overhead happens.

	17 →18	18→19	19→20	20→21	21→22	22→23	23→24	24→25	25→26
DP	3.13	3.14	3.12	3.12	3.42	3.92	3.81	3.82	3.69
IDP	3.00	3.05	2.98	3.25	3.56	3.95	3.82	3.71	3.73

Table 3.3: Time increment when increasing the problem size

Chapter 4

Optimized Implementation

In this chapter we study how the data is represented in the CSGP and we explore how to optimize the operation that is performed most frequently, which is the *coalition* split.

Thus we propose an alternative method for calculating the splittings of a *coalition* using binary operations with the objective of making this operation as fast as possible; we call it method the Fast Split Method (FSM).

Using this method we propose a new implementation for the DP and IDP algorithms, using FSM. Our versions are called Fast Split DP (FSDP) and Fast Split IDP (FSIDP). They are presented in Figure 4.1.

After presenting the theoretical background, we evaluate their performance and we find that

- FSDP produces an average of 19.9 *SpeedUp* over DP.
- FSIDP produces an average of 23.9 *SpeedUp* over DP and an average of 1.15 over FSDP.

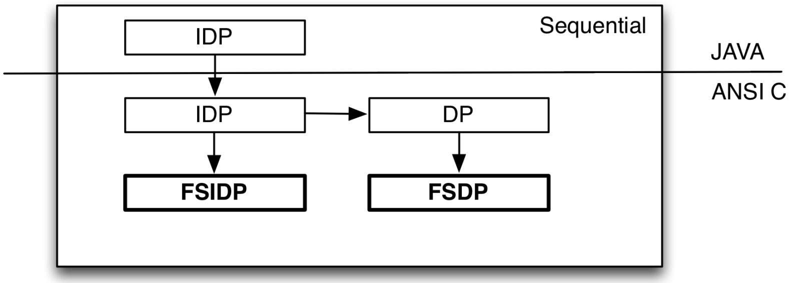

Figure 4.1: FSDP and FSIDP

4.1 Data representation

The CSGP has as an input a collection of values for each possible combination of a group of elements (*coalitions*) namely 2^n valuations. It is needed to store both the *coalition* and the value. There exist different alternatives for representing the *coalition*, but probably the optimal and most compact option is to use a binary representation.

Given a number of agents n, a binary word of n bits is enough to represent all possible *coalitions* where one agent is represented by one bit. Table 4.1 shows how to encode four *coalitions* whose elements belong to a set of size eight. In this case, it is enough with one byte to represent all possible *coalitions*.

Coalition	Binary	Decimal	*Coalition*	Binary	Decimal
$\{a_1\}$	00000001	1	$\{a_4, a_6, a_7, a_8\}$	11101000	232
$\{a_1, a_2, a_3, a_4\}$	00001111	15	$\{a_1, a_3, a_5, a_7\}$	01010101	85

Table 4.1: *Coalition* representation samples given 8 elements

Every *coalition* has associated its *coalition value*. The DP and IDP algorithms require storing both values for every *coalition*. IDP proposes using the *coalition* as an index of a matrix of size 2^n. Using this technique, the entire data presented in Table 2.3 can be stored as a vector with these contents [33, 39, 87, 13, 87, 36, 97, 40, 70, 52, 111, 67, 100, 132, 155]. This way of storing both *coalitions* and *coalition values* provides high compactness, which is very useful when having problems with a large input.

4.2 Optimizations in the internal loop

As seen in section 2.3.3, DP's complexity is exponential and it is characterized by three nested loops. The code executed inside the inner loop is critical in the sense that it is the one most frequently executed. As an example, consider the execution of a problem of size 30. The number of iterations per loop are shown in Table 4.2 where the last two columns show the number of times that the code in the inner loop is executed.

As a reminder, the pseudocode is again presented in Algorithm 1, shown further in the chapter. Along this chapter, we are interested in the internal loop behavior (lines 5-10).

Inside this loop, two logical groups of operations can be identified. First, the calculation of the splittings using the *nextSplit()* function. Second, the memory accesses in order to evaluate the *coalition values* and, in some cases, update it.

Size of *coalition* (m)	# of coalitions of size m	# splitting of a single *coalition* of size m	#splittings of all coalitions of size m	
			value	log_{10}
2	435	1	435	2.64
3	4060	3	12180	4.09
4	27405	7	191835	5.28
5	42506	15	2137590	6.33
6	593775	31	18407025	7.26
7	2035800	63	128255400	8.11
8	5852925	127	743321475	8.87
9	14307150	255	3648323250	9.56
10	30045015	511	15353002665	10.19
11	54627300	1023	55883727900	10.75
12	86493225	2047	177051631575	11.25
13	119759850	4095	490416585750	11.69
14	145422675	8191	1191157130925	12.08
15	155117520	16383	2541290330160	12.41
16	145422675	32767	4765064791725	12.68
17	119759850	65535	7848461769750	12.89
18	86493225	131071	11336753493975	13.05
19	54627300	262143	14320164303900	13.16
20	30045015	524287	15752210779305	13.20
21	14307150	1048575	15002119811250	13.18
22	5852925	2097151	12274467516675	13.09
23	2035800	4194303	8538762047400	12.93
24	593775	8388607	4980945121425	12.70
25	142506	16777215	2390853800790	12.38
26	27405	33554431	919559181555	11.96
27	4060	67108863	272461983780	11.44
28	435	134217727	58384711245	10.77
29	30	268435455	8053063650	9.91
30	1	536870911	536870911	8.73

Table 4.2: Number of coalitions and splittings in DP for a problem of size 30

Algorithm 1 Pseudo-code of the Dynamic Programming Algorithm

```
for $m \leftarrow [2 \dots n]$ do
    for $coalition \leftarrow coalitionsOfSize(m)$ do
        $max\_value \leftarrow value[coalition]$
        $(S1, S2) \leftarrow init()$
        for $(S1, S2) \leftarrow nextSplit(coalition, (S1, S2))$ do
            if $(max\_value < value[S1] + value[S2]$ then
                $max\_value \leftarrow value[S1] + value[S2]$
            end if
        end for
        $value[coalition] \leftarrow max\_value$
    end for
end for
```

4.2.1 The nextSplit function

This function generates a new *splitting* given a *coalition* and the previous *splitting*. Note that, as defined in section 2.3.1, a *split* is the action of dividing a *coalition* in two subsets, while *splitting* is the result of the application of a *split*.

The total number of *splittings* is defined by the number of elements of the *coalition*. If m is the size of the actual *coalition*, the total number of *splittings* will be $2^{m-1} - 1$.

Let $T = \{a_1, .., a_m\}$ be an element set from which we have a *coalition* formed by a subset of elements $S \subset T$. The split function has to be understood as the selection of two disjoint subsets S_1 and S_2, such that $S = S_1 \cup S_2$, and $S_1, S_2 \neq \varnothing$.

The initial set of elements to *split* is encoded by a binary number. Consider for instance a set $T = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$ of size 8 and $S = \{a_1, a_2, a_3, a_5\}$ a subset of T. As shown in section 4.1, this selection can be represented as a number $s = 00010111_{/b}$ (in decimal 23).

The internal loop will go through all the possible *splittings* of this subset. That is, iterate through all the possible ways of selecting two subsets containing 1 and 3 elements of S respectively and all the possible ways of selecting two subsets containing 2 and 2 elements of S.

We propose a method in order to calculate this in a computationally efficient way. We call it Fast Split Method.

The Fast Split Method

Let C be the binary representation of the *coalition* to be splitted.

- First we calculate the two's complement of C, denoted by C^{**}.
- Then we can compute the first splitting performing the operation:

$S_1 \leftarrow (C^{**} \text{ AND } C)$
$S_2 \leftarrow (C - S_1)$.

- For the next iterations we need the previous S_1 (denoted by S_1'). The calculation of the new splitting can be performed as :

$S_1 \leftarrow ((C^{**} + S_1')\, AND\, C)$.
$S_2 \leftarrow (C - S1)$.

Note that if we initialize S_1 as zero the the first split can be obtained using the general term calculation.

Example

We want get all the splittings for the *coalition* $\{a_1, a_2, a_3, a_5\}$ represented by the bitmask 00010111

Initializations:
$C = 00010111$
$C^{**} = 11101001$
$S_1 = 0$

Since the number of elements in the *coalition* is 4 we can compute the number of *splittings* as $2^{4-1} - 1 = 7$. Applying our Fast Split Method we will obtain the *splittings* shown in Table 4.3.

Iteration	1	2	3	4	5	6	7
Subset 1	$\{a_1\}$ 00000001	$\{a_1\}$ 00000010	$\{a_1, a_2\}$ 00000011	$\{a_3\}$ 00000100	$\{a_1, a_3\}$ 00000101	$\{a_2, a_3\}$ 00000110	$\{a_1, a_2, a_3\}$ 00000111
Subset 2	$\{a_2, a_3, a_5\}$ 00010110	$\{a_1, a_3, a_5\}$ 00010101	$\{a_3, a_5\}$ 00010100	$\{a_1, a_2, a_5\}$ 00010011	$\{a_2, a_5\}$ 00010010	$\{a_1, a_5\}$ 00010001	$\{a_5\}$ 00010000

Table 4.3: Fast Split Method trace

4.2.2 Memory accesses

Once one *splitting* is calculated, the pair forming the *splitting* determines the two memory positions where the *coalition values* are stored. The algorithm accesses those positions in order to evaluate what is better: the *coalition value* associated with the initial *coaliton* or the one associated with the current *splitting*.

4.3 The IDP Algorithm

The IDP Algorithm introduces some extra computation in order to avoid the evaluation of some nodes where in fact there are redundant calculations. The basic idea behind this approach is to introduce a little overhead in order to avoid a significant amount of calculation. This algorithm can be implemented following the schema presented in Algorithm 2, which only differs from DP in the sixth line where a condition is added.

Algorithm 2 Pseudo-code of the IDP Algorithm

```
 1: for $m \leftarrow [2 \dots n]$ do
 2:     for $coalition \leftarrow coalitionsOfSize(m)$ do
 3:         $max\_value \leftarrow value[coalition]$
 4:         $(S1, S2) \leftarrow init()$
 5:         for $(S1, S2) \leftarrow nextSplit(coalition, (S1, S2))$ do
 6:             if $(sizeOf(S1) \geq n - m)$ then
 7:                 if $(max\_value < value[S1] + value[S2]$ then
 8:                     $max\_value \leftarrow value[S1] + value[S2]$
 9:                 end if
10:             end if
11:         end for
12:         $value[coalition] \leftarrow max\_value$
13:     end for
14: end for
```

In our implementation we coded this extra conditional in a very efficient way, specifically, the function $sizeOf()$ which is in charge of counting how many members a *coalition* has, can be implemented in a computationally efficient way, since in the data representation we use, this number is determined by the number of bits equal to 1. Counting this number of bits is an operation that is actually implemented in most of the modern processors and it is, in consequence, very fast.

4.4 FSDP and FSIDP performance analysis

Our optimized algorithms are called FSDP and FSDP respectively. The objective of these algorithms is to reduce the total number of instructions executed. Therefore, they do not modify the behavior and the complexity of the existing algorithms. We perform some experiments in order to contrast our algorithms' results with the results obtained using DP and IDP.

4.4.1 Measuring execution time.

	DP	**IDP**		**FSDP**		**FSIDP**	
n	Time	Time	*SpeedUp*	Time	*SpeedUp*	Time	*SpeedUp*
17	3.21	1.00	3.21	0.17	18.77	0.17	19.09
18	10.04	3.28	3.06	0.50	20.04	0.47	21.39
19	31.54	10.01	3.15	1.50	21.02	1.36	23.22
20	98.41	29.64	3.32	4.51	21.81	3.98	24.73
21	306.64	96.35	3.18	13.49	22.73	11.75	26.10
22	1048.37 (17 min)	349.08 (5:49 min)	3.00	55.84	18.77	43.83	23.92
23	4109.69 (68 min)	1426.23 (23 min)	2.88	270.56 (4:30 min)	15.19	179.21 (3 min)	22.93
24	15642.51 (4:20h)	5395.69 (1:30h)	2.90	980.01 (16 min)	15.96	609.43 (10 min)	25.67
25	59684.43 (16:30h)	20978.75 (5:50h)	2.84	3121.13 (52 min)	19.12	1965.43 (32 min)	30.37

Table 4.4: Time in seconds and *SpeedUp* for DP, IDP, FSDP, FSIDP algorithms .

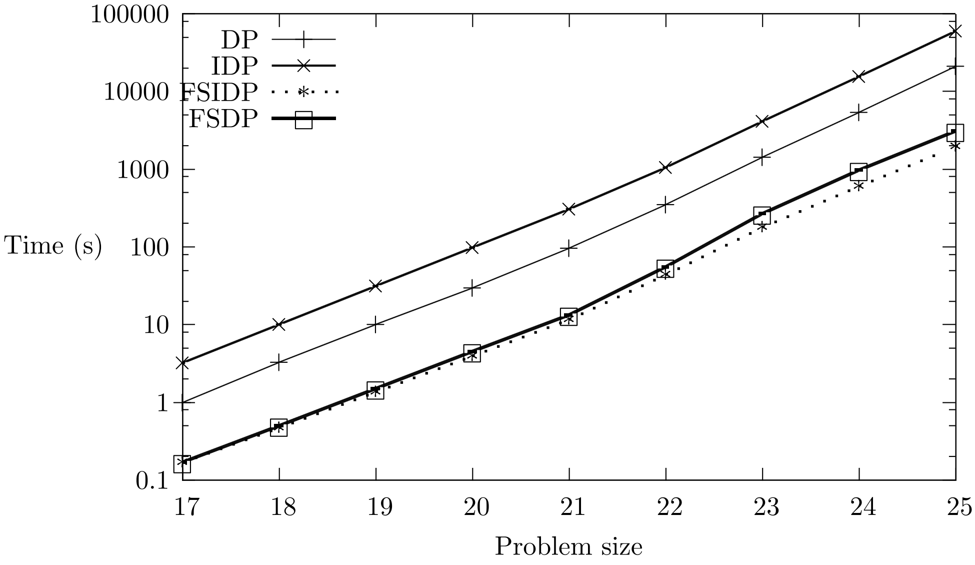

Figure 4.2: Execution time in seconds for DP, IDP, FSDP, FSIDP algorithms.

We run problems varying their size from 17 to 25 and solve them using the different algorithms: DP, IDP, FSDP, FSIDP. The results are presented in Table 4.4 and Figure 4.2.

The first conclusion we obtain from the results is that FSM significantly improves the performance of both DP and IDP. We also find that the improved version of the original algorithm (IDP), improves DP on a significant factor (3X), whereas FSIDP improves FSDP only on an average of (1.15X).

Another interesting conclusion that can be obtained from our experiment is that for problems bigger than $n = 21$ there is a degradation of the performance, which is more accented in the FSDP algorithm. As we detail in the next chapter, this degradation is basically due to the memory system latency.

4.4.2 Reviewing the analytical model

In Section 2.3.5 we presented a model to determine when IDP is improving DP and in Section 3.3.3 we found a value for the probability of evaluating a spitting by IDP ($p = 0.5$).

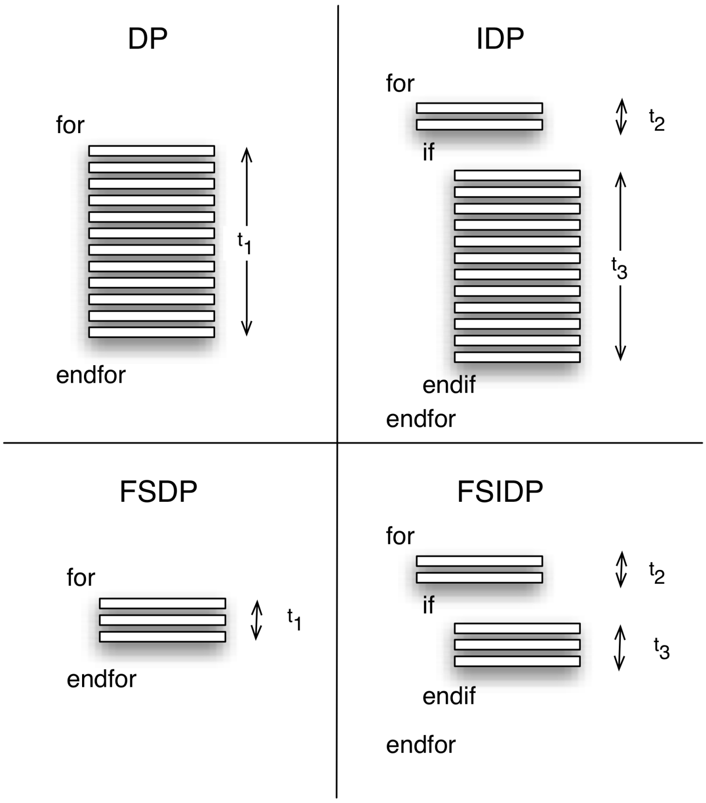

Figure 4.3: Scheme of the internal loop execution for DP, IDP, FSDP and FSIDP

With this data we are able to compute more precisely when IDP improves DP. As seen before, it will be determined by t_1, t_2 and t_3. Figure 4.3 gives an idea of the different strategies used by the algorithms in order to obtain performance. While IDP reduces the number of executions for the block t_3, FSDP tries to reduce the total number of executed instructions. FSIDP combines both techniques.

From Figure 4.3, and considering the probability of satisfying the conditional as $p = 0.5$, we can deduce that IDP will improve DP while

$$2t_2 + t_3 < 2t_1.$$

In addition it is possible to compute the *SpeedUp* expected from IDP and FSIDP as

$$SpeedUp = \frac{2t_1}{2t_2 + t_3}.$$

If we suppose that $t_1 \leq t_2 + t_3$, we find that the *SpeedUp* has an upper bound that is equal to 2. But we found experimentally that IDP improves DP in a factor of 3. The consequence of this is that our model is not valid for DP and IDP algorithms, which require more computational effort when operating with bigger *coalitions* than for smaller ones. In Section 5.2.1 we present a more detailed discussion about this behavior.

However, FSDP and FSIDP can be approximated by this analytical model which is a simplified version of the real execution pattern.

Chapter 5

Performance analysis

In the previous chapters we have presented and measured four algorithms for solving the Coalition Structure Generation Problem (CSGP). Those algorithms are DP, IDP, FSDP and FSIDP.

Until now, we measured them using the total execution time. In the present chapter we perform a deeper analysis, taking more measurements which allow us to have a better characterization of the different algorithms' behavior.

5.1 Experiments

We use the Performance Application Programming Interface [1] (PAPI) in order to have access to the CPU hardware counters. This library allows us to know more precisely how the different algorithms are using the hardware resources and allow us to analyze the application's execution pattern.

The use of PAPI requires to adapt the application including calls to the PAPI libraries. Once PAPI is instrumented we have access to different measures such as total number of instructions executed, total number of CPU cycles or number of accesses to different cache levels.

The computer used for running the experiments is a six-core Intel Xeon E5645 Processor at 2.40GHz (already presented in Section 3.3).

5.2 Characterizing algorithms

We instrumented the different algorithms with PAPI and we took measurements of the application's performance.

5.2.1 Total instructions per complexity

The complexity of the algorithm is an objective measure that gives us an idea of how the computation grows when increasing the problem size. Thanks to the

Hardware counters, we are able to know how many instructions the algorithm executes.

In Table 5.1 we present the results of executing the measure varying the problem size.

n	17	18	19	20	21	22	23	24	25
DP	142.87	146.55	150.23	153.89	157.54	161.16	164.75	168.28	171.76
IDP	44.09	47.46	48.45	47.34	50.48	51.49	50.52	53.54	54.56
FSDP	6.55	6.53	6.52	6.51	6.51	6.51	6.50	6.50	6.50
FSIDP	6.05	6.03	6.02	6.02	6.01	6.01	6.00	6.00	6.00

Table 5.1: Total instructions per 3^n (complexity).

From the results obtained, we can draw two conclusions: First, that the original DP and IDP algorithms need much more instructions per unit of work than the FS versions. This overwork is especially significant in the case of the DP. Second, that in the DP and IDP versions, when incrementing the complexity of the problem, the number of instructions per complexity grows. This growth over the complexity means that there must be some inefficiencies in the implementation.

Given a CSGP size, the complexity is constant, therefore this measure allows us to establish a fair comparison between algorithms in terms of instructions executed. A reduction of the number of instructions executed usually leads to a reduction of the final execution time.

DP and IDP clearly use more instructions and need more CPU time than FSDP and FSIDP for solving the same CSGP. They are far from FSDP and FSIDP and in consequence we will focus our analysis only on the FSDP and FSIDP algorithms.

5.2.2 Instructions per memory access

This metric provides an estimation of how many instructions need to be executed in order to fetch a single value from memory. We obtain the value of the total number of instructions reading them from the hardware counters, while the number of access to memory is obtained in Section 3.3.3. Results are presented in Table 5.2.

On the one hand, FSDP requires an average of 6.52 instructions to fetch a value from memory, while FSIDP needs around 12. On the other hand, FSIDP is faster than FSDP. The fact is that FSIDP executes approximately half of the memory accesses. Therefore, even though the computation of the addressing for FSIDP is expensive, it is balanced by the reduction of memory access.

n	17	18	19	20	21	22	23	24	25	26
FSDP	6.56	6.54	6.53	6.52	6.51	6.51	6.51	6.50	6.50	6.50
FSIDP	12.10	12.07	12.04	12.03	12.02	12.01	12.01	12.01	12.00	12.00

Table 5.2: Instructions per memory access

Figures 5.1, and 5.2, present the code executed inside the internal loop for FSDP and FSIDP, and Figures 5.3 and 5.4 show their corresponding translation to assembler.

```
set1=(ca2 + set1) & bitMask;
set2=bitMask-set1;
tmp=values[set1]+values[set2];
if (tmp>max) max=tmp;
```

Figure 5.1: Internal loop FSDP (ANSI C)

```
if (__builtin_popcount(set1) >=m) {
    set2=bitMask-set1;
    tmp=values[set1]+values[set2];
    if (tmp>max) max=tmp;
}
```

Figure 5.2: Internal loop FSIDP (ANSI C)

```
   .L72:
01   addl %r10d, %edx
02   movl %eax, %ecx
03   andl %eax, %edx
04   subl %edx, %ecx
05   movslq %edx, %r9
06   movslq %ecx, %rcx
07   movl (%r8,%rcx,4), %ecx
08   addl (%r8,%r9,4), %ecx
09   cmpl %ecx, %esi
10   cmovl %ecx, %esi
11   addl $1, %edi
12   cmpl %r13d, %edi
13   jbe .L72
```

Figure 5.3: Internal loop FSDP (ASSEMBLER)

```
   .L61:
01   addl %r9d, %ecx
02   andl %eax, %ecx
03   popcntl %ecx, %edi
04   cmpl %edi, %ebx
05   ja .L60
06     movl %eax, %edi
07     movslq %ecx, %r15
08     subl %ecx, %edi
09     movslq %edi, %rdi
10     movl (%r8,%rdi,4), %edi
11     addl (%r8,%r15,4), %edi
12     cmpl %edi, %edx
13     cmovl %edi, %edx
   .L60:
14   addl $1, %esi
15   cmpl %r13d, %esi
16   jbe .L61
```

Figure 5.4: Internal loop FSIDP (ASSEMBLER)

FSDP code analisys

The assembly code corresponding to the FSDP uses thirteen instructions that include two memory access operations (instructions 7 and 8). These numbers verify the results found experimentally. That is, the average number of instructions per access is 6.5.

For small CSGPs this ratio value is slightly over 6.5 due to the rest of instructions executed, but the bigger the CSGP is, the relevance of the internal loop overshadows any other measure.

FSIDP code analisys

The assembly code generated for the FSIDP algorithm is quite more complex. It is formed by 8 instructions that are always executed (instructions 01, 02, 03, 04, 05, 14, 15, 16) and other 8 that are only executed if the condition in line 06 is satisfied (instructions 06, 07, 08, 09, 10, 11, 12, 13). According to the data obtained in Experiment 3.3.3, the conditional in line 06 is satisfied in around 50% of the executions and, when it is satisfied it will lead to the execution of the two load operations at 10 and 11. Performing some basic calculation, we find that the average memory access is done every 12 cycles. This value confirms what we found in the experimental data.

5.2.3 Cycles per Instruction (CPI)

The ratio between the total number of cycles and the total number of instructions can reveal useful information about the application's behavior. The mean time per instruction in CPU cycles expresses how fast the processor executes instructions. A high CPI indicates that the Processing unit is waiting for something, typically a memory access or a data for a calculation.

Table 5.3 and Figure 5.5 reflect the CPI measured for the different algorithms varying the data size from $n = 17$ to $n = 25$.

n	17	18	19	20	21	22	23	24	25
Problem size (MB)	0.5	1	2	4	8	16	32	64	128
FSDP	0.49	0.51	0.53	0.54	0.54	0.75	1.22	1.48	1.57
FSIDP	0.51	0.51	0.51	0.51	0.51	0.64	0.88	1.00	1.07

Table 5.3: Different algorithm CPI & growing CSGP size.

Examining the results, we can see that there are two different regions having similar behavior. For CGSP executions having an input data smaller than $n = 22$, a growth in the problem has no significant impact on the CPIs. There are no important differences between algorithms in terms of CPIs both algorithms need around 0.5 cycles per instruction. When the CGSP is bigger, the number of CPI starts growing extensively.

In this case, the most important factor affecting the CPI is the effect of cache misses. For small CSGP, all the required data fit in the cache, so when the program performs a LOAD operation, the data is probably hitting in the cache.

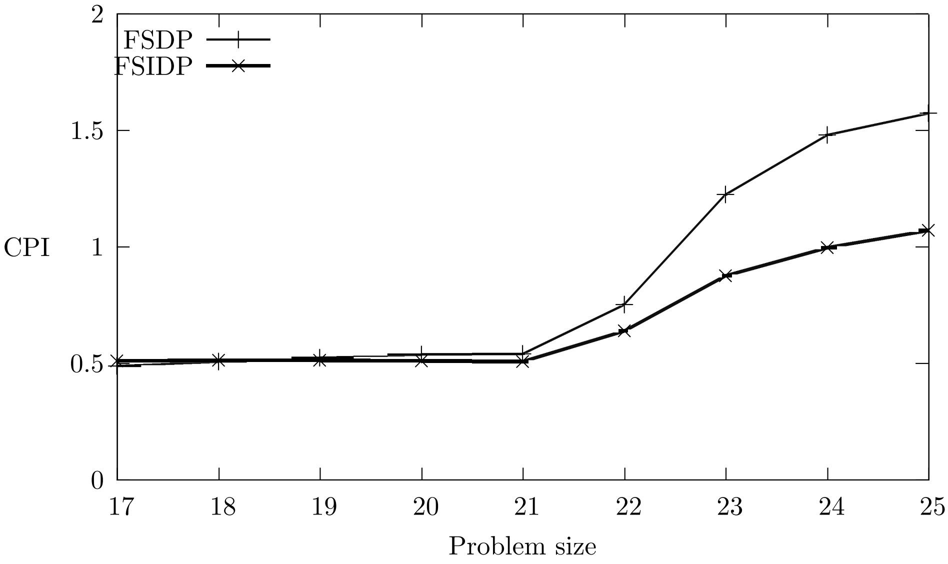

Figure 5.5: CPI for DP,IDP,FSDP,FSIDP algorithms.

However, when the CSGP is bigger, memory accesses randomly hit or miss in the cache, producing a slower average memory latency.

In addition, FSIDP needs less CPIs than FSDP. This is a consequence of what we found in Section 5.2.3, the ratio of memory instruction per iteration is approximately one-to-six in the case of FSDP and one-to-twelve in the case of the FSIDP.

5.2.4 Cache misses

We computed the percent of last level cache misses, shown in Table 5.4. This measurement is calculated having the total number of access and the total number of cache misses.

n	17	18	19	20	21	22	23	24	25
Problem size (MB)	0.5	1	2	4	8	16	32	64	128
FSDP (%)	0.001	0.005	0.010	0.008	0.060	21.023	61.693	77.613	88.815
FSIDP (%)	0.014	0.023	0.033	0.026	0.085	22.219	61.840	79.908	88.267

Table 5.4: % Cache misses for FSDP and FSDIP and problem memory requirements

This cache miss ratio verifies that the CPI curve is related to the memory access instructions. Cache misses are not significant for problems smaller than $n = 22$, whilst for bigger problems this ratio of cache misses grows. Moreover, this cache misses curve presents a high correlation with the CPIs, meaning that has a clear influence on the average cost of the instructions.

Another interesting consideration is that both FSDP and FSIDP present a similar cache miss ratio, meaning that there are no big differences between the memory access patterns for both algorithms.

5.2.5 Memory Bandwidth

We measure Memory Bandwidth by using the total number of cycles needed for the algorithm, the clock frequency specified for our processor, which is 2,40GHz and the number of memory accesses.

n	17	18	19	20	21	22	23	24	25
Problem size (MB)	0.5	1	2	4	8	16	32	64	128
FSDP (MB/s)	2860	2751	2668	2615	2596	1868	1150	951	895
FSIDP (MB/s)	1479	1479	1482	1489	1496	1193	871	766	712

Table 5.5: Memory Bandwidth (MB/s) for FSDP and FSIDP algorithms

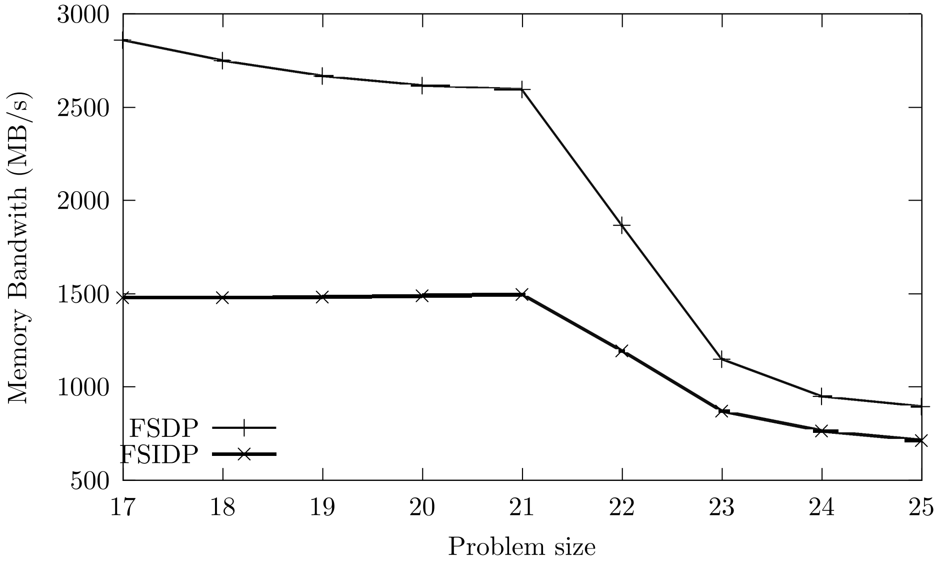

Figure 5.6: Memory Bandwidth (MB/s) for FSDP and FSIDP algorithms.

The provided results shown in Table 5.5 and Figure 5.6 are according to what we found in previous sections. Data is obtained fast for small problems, and when the problem size exceeds the cache, there is a fall in Memory Bandwidth.

FSDP fetches data with more frequency than FSIDP, thus it gets a higher memory bandwidth ratio.

We took measurements of the capacity of our memory system using a tool for benchmarking memory [15]. The measurements we get is that our memory system maximum bandwidth is 42540.1 MB/s. This result is far from the results we obtain with our algorithms. Which means that Memory Bandwidth is not limiting our application performance.

5.3 Conclusions

After analyzing the behavior of the DP and IDP algorithms in front of the FSDP and FSIDP, we find that our FS versions are considerably faster. There is a big difference between the DP and IDP algorithms and the FSDP and FSIDP ones. In consequence, we will focus on the study of the FS versions, analyzing their performance and their characterization.

FSDP and FSIDP are behaving equally when the problem is bigger, that is to say the number of instructions executed is increasing according to the complexity.

The memory access pattern is irregular, this lack of locality when accessing memory system becomes relevant when the problem representation does not fit in the last level cache, producing a high cache miss ratio which is expensive in terms of CPU cycles. Therefore when CSGP representation in memory is bigger than the memory cache, the performance of the algorithm decreases.

FSIDP avoids the half of the memory access adding an extra computation. For minor CSGPs, this extra computation is scarcely balancing the savings in time due to the memory access avoidance. Nevertheless, when the problem is bigger, this saving has more relevance as memory access are expensive. We are reinforcing the original IDP article idea of saving operations when these are costly memory accesses.

Another consequence of the disfavorable access pattern is the small memory bandwidth consumed, around a 2.5% of the peak bandwidth of the system.

Chapter 6

Shared Memory Approach

In the previous chapter we saw how the DP algorithm works using a sequential paradigm and we discussed how to optimize the execution to run as fast as possible in one processing core. Those optimizations were made in the portion of code inside the deepest loop (*coalition* evaluation loop), which is both computation and memory intensive.

However, solving a CSGP implies that a lot of CPU time is required, and nowadays typical processors are provided with two or more cores.

In this chapter we propose a new algorithm to parallelize a CSGP using a Shared Memory paradigm: the SMIDP algorithm.

Our proposal is based on the FSIDP algorithm (See Figure 6.1). The main idea behind this approach is dividing a CSGP in different threads which are executed in different processing cores.

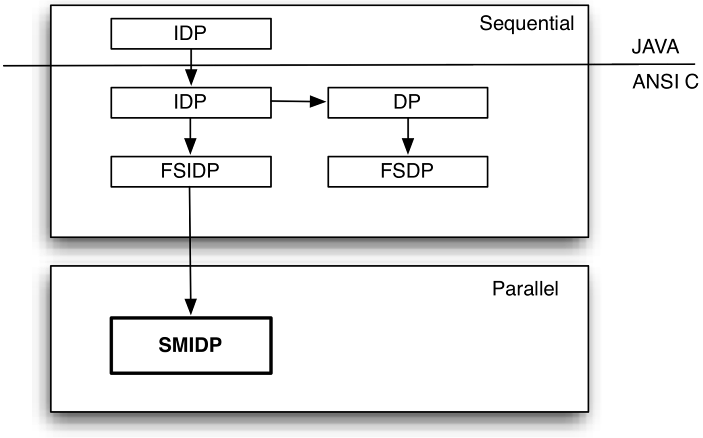

Figure 6.1: Roadmap for SMIDP

Nevertheless, dividing the problem in a way that achieves good performance (high efficiency) is not easy. CSGPs require the evaluation of all possible Coalition Structures, which are highly coupled and interrelated, i.e. there may exist a high amount of dependencies. Thus ,it is not possible to simply divide the input data between different threads.

This chapter presents the algorithm analysis made to parallelize CSGP and the rationale behind the methods used to distribute the work along different processor units.

6.1 Algorithm analysis

Our objective is to distribute the work between different computation nodes, therefore in this case we focus on how the data is generated and the execution scheduled. The optimizations done for the sequential part remain unchanged whereas the code responsible for orchestrating the execution will be modified.

A good parallel algorithm must:

- Reduce synchronization between threads. Assigning big chunks of work to every thread is desirable when possible.
- Balance the work. The workloads of each thread have to be balanced. Having different workload for each thread leads to an infra-utilization of hardware resources.
- Reduce the use of shared resources, typically memory.

Having these three points in mind, we analyze the algorithm in order to see when it is possible to execute different parts of the code in different threads. These parts have to be as big as possible, as balanced as possible and sharing the least possible amount of resources.

In Section 2.3.2 we said that the execution of DP is characterized by three nested loops:

- **Coalition size selection** (outer loop).
- **Coalition generation** (middle loop).
- **Coalition evaluation** (internal loop).

We start trying to parallelize the most external loop. We see that the *coalition* size selection loop (outer loop) is responsible for establishing what the size of the *coalitions* to be evaluated is (defined as m). DP imposes an unbreakable dependence condition: it is necessary to compute all the *coalitions* of size $m-1$ in order to process all the *coalitions* of size m. Consequently, it is not possible to divide the execution of this loop among different threads. The iterations have to be processed sequentially.

The *coalition* selection loop (the middle loop) selects the *coalitions* that must be evaluated. As we will see, the *coalitions* selected by this loop can be executed in parallel. We will next analyze the behavior of this loop.

A single iteration of this loop can be described as:

- Each iteration of the loop is responsible for evaluating a group of elements $A = \{a_1, ..., a_m\}$. It **reads** the *coalition value* from memory.

$$max\_value \leftarrow v[A].$$

- The set of elements is split in all possible subsets S_1, S_2 where $S_1 \cup S_2 = A$ and $S_1, S_2 \neq \varnothing$. For each pair (S_1, S_2) there are two **read** operations $v[S_1]$ and $v[S_2]$. On each internal loop iteration max_value is updated as $max\_value \leftarrow max(max\_value, v[S_1] + v[S_2])$.
- The final value in max_value is **written** back to memory. This writing can be skipped if $max\_value = v[A]$.

$$v[A] \leftarrow max\_value.$$

Every iteration of this loop performs a number of read operations and only one writing. It is important to stress that every iteration stores a value in the memory position, the one corresponding to the *coalition* A. Thus every write operation for a given *coalition* is performed only once. From this, we ensure that the operations performed in the selection loop can be executed in parallel, without altering the final result.

Table 6.1 illustrates a trace of one iteration for the *coalition* selection loop for a size-3 *coalition* on a size-4 problem. We notice that for every different *coalition* A, there is a block calculating and storing its max_value. Indeed, it is possible to perform this computation independently from the data calculated in the other *coalitions'* blocks. In this example, the calculation of the best split for the *coalition* $\{a_1, a_2, a_3\}$ can be performed at the same time as the one for *coalitions* $\{a_1, a_2, a_4\}$,$\{a_1, a_3, a_4\}$ and $\{a_2, a_3, a_4\}$.

To sum up, the behavior of the DP algorithm implies first calculating the *coalition* values for the smallest possible *coalitions* (size two), then calculating the *coalitions* of size three, using the *coalitions* calculated in the previous step, then *coalitions* of size four, and so on. This way of proceeding guarantees that after the calculation of a *coalition* of size m, this data will not be modified in the future, and it will only be needed for coalitions bigger than m. Accordingly, it is possible to carry out the *coalition* generation and evaluation distributed in different threads without any risk of overwriting useful data.

6.2 Distributing work

In order to distribute the execution among different CPUs, we need to develop a novel method of computing how the *coalitions* are generated.

m	A	$v(A)$	S_1,S_2	Evalutation	max_value
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

Table 6.1: Parallel distribution analsys for a problem of size 4

We propose two different functions to generate coalitions.

- the `getCombination` function. Computes the k-th *coalition* of m members (bits) from a group of n. It is slow.
- the `getNext` function. Computes the k-th +1 *coalition* from the kth *coalition*. It is considerably faster.

Generating a *coalition* means selecting m elements from a group of n. It is known that they are $\binom{n}{m}$ possible options, and what is needed is to define a way to generate all the different possibilities and distribute them without repeating work. In fact, we will need to define an order in which the *coalitions* are generated. Using a fixed order it is easy to schedule which thread is going to process each range of *coalitions*.

Table 6.2 shows all the possible combinations of 4 elements having a total number of $n = 8$ elements in lexicographical order. Although there are a number of alternatives, the lexicographical order enables us to construct the direct access method implemented in the `getCombination` function.

In the original version of the DP algorithm, *coalitions* are computed in a sequential way. We required a *coalition* in order to compute the next. This data dependency was hindering the possibility of parallelizing the coalition selection loop. This is the method used by FSDP and FSIDP, because it is both efficient and fast.

The function `getCombination` is ideal for distributing the job in different computational units. Let us take in consideration the case of having to evaluate the *coalitions* of size 4 for a problem of size 8, as shown in Table 6.2; and let us suppose that we dispose of two computational units to perform the work. We have

Order	Encoding		Combination	Order	Encoding		Combination
(k)	Bin	Dec		(k)	Bin	Dec	
1	1111	5	$\{a_1, a_2, a_3, a_4\}$	36	...1111.	30	$\{a_2, a_3, a_4, a_5\}$
2	...1.111	23	$\{a_1, a_2, a_3, a_5\}$	37	..1.111.	46	$\{a_2, a_3, a_4, a_6\}$
3	..1..111	39	$\{a_1, a_2, a_3, a_6\}$	38	.1..111.	78	$\{a_2, a_3, a_4, a_7\}$
4	.1...111	71	$\{a_1, a_2, a_3, a_7\}$	39	1...111.	142	$\{a_2, a_3, a_4, a_8\}$
5	1....111	135	$\{a_1, a_2, a_3, a_8\}$	40	..11.11.	54	$\{a_2, a_3, a_5, a_6\}$
6	...11.11	27	$\{a_1, a_2, a_4, a_5\}$	41	.1.1.11.	86	$\{a_2, a_3, a_5, a_7\}$
7	..1.1.11	43	$\{a_1, a_2, a_4, a_6\}$	42	1..1.11.	150	$\{a_2, a_3, a_5, a_8\}$
8	.1..1.11	75	$\{a_1, a_2, a_4, a_7\}$	43	.11..11.	102	$\{a_2, a_3, a_6, a_7\}$
9	1...1.11	139	$\{a_1, a_2, a_4, a_8\}$	44	1.1..11.	166	$\{a_2, a_3, a_6, a_8\}$
10	..11..11	51	$\{a_1, a_2, a_5, a_6\}$	45	11...11.	198	$\{a_2, a_3, a_7, a_8\}$
11	.1.1..11	83	$\{a_1, a_2, a_5, a_7\}$	46	..111.1.	58	$\{a_2, a_4, a_5, a_6\}$
12	1..1..11	147	$\{a_1, a_2, a_5, a_8\}$	47	.1.11.1.	90	$\{a_2, a_4, a_5, a_7\}$
13	.11...11	99	$\{a_1, a_2, a_6, a_7\}$	48	1..11.1.	154	$\{a_2, a_4, a_5, a_8\}$
14	1.1...11	163	$\{a_1, a_2, a_6, a_8\}$	49	.11.1.1.	106	$\{a_2, a_4, a_6, a_7\}$
15	11....11	195	$\{a_1, a_2, a_7, a_8\}$	50	1.1.1.1.	170	$\{a_2, a_4, a_6, a_8\}$
16	...111.1	29	$\{a_1, a_3, a_4, a_5\}$	51	11..1.1.	202	$\{a_2, a_4, a_7, a_8\}$
17	..1.11.1	45	$\{a_1, a_3, a_4, a_6\}$	52	.111..1.	114	$\{a_2, a_5, a_6, a_7\}$
18	.1..11.1	77	$\{a_1, a_3, a_4, a_7\}$	53	1.11..1.	178	$\{a_2, a_5, a_6, a_8\}$
19	1...11.1	141	$\{a_1, a_3, a_4, a_8\}$	54	11.1..1.	210	$\{a_2, a_5, a_7, a_8\}$
20	..11.1.1	53	$\{a_1, a_3, a_5, a_6\}$	55	111...1.	226	$\{a_2, a_6, a_7, a_8\}$
21	.1.1.1.1	85	$\{a_1, a_3, a_5, a_7\}$	56	..1111..	60	$\{a_3, a_4, a_5, a_6\}$
22	1..1.1.1	149	$\{a_1, a_3, a_5, a_8\}$	57	.1.111..	92	$\{a_3, a_4, a_5, a_7\}$
23	.11..1.1	101	$\{a_1, a_3, a_6, a_7\}$	58	1..111..	156	$\{a_3, a_4, a_5, a_8\}$
24	1.1..1.1	165	$\{a_1, a_3, a_6, a_8\}$	59	.11.11..	108	$\{a_3, a_4, a_6, a_7\}$
25	11...1.1	197	$\{a_1, a_3, a_7, a_8\}$	60	1.1.11..	172	$\{a_3, a_4, a_6, a_8\}$
26	..111..1	57	$\{a_1, a_4, a_5, a_6\}$	61	11..11..	204	$\{a_3, a_4, a_7, a_8\}$
27	.1.11..1	89	$\{a_1, a_4, a_5, a_7\}$	62	.111.1..	116	$\{a_3, a_5, a_6, a_7\}$
28	1..11..1	153	$\{a_1, a_4, a_5, a_8\}$	63	1.11.1..	180	$\{a_3, a_5, a_6, a_8\}$
29	.11.1..1	105	$\{a_1, a_4, a_6, a_7\}$	64	11.1.1..	212	$\{a_3, a_5, a_7, a_8\}$
30	1.1.1..1	169	$\{a_1, a_4, a_6, a_8\}$	65	111..1..	228	$\{a_3, a_6, a_7, a_8\}$
31	11..1..1	201	$\{a_1, a_4, a_7, a_8\}$	66	.1111...	120	$\{a_4, a_5, a_6, a_7\}$
32	.111...1	113	$\{a_1, a_5, a_6, a_7\}$	67	1.111...	184	$\{a_4, a_5, a_6, a_8\}$
33	1.11...1	177	$\{a_1, a_5, a_6, a_8\}$	68	11.11...	216	$\{a_4, a_5, a_7, a_8\}$
34	11.1...1	209	$\{a_1, a_5, a_7, a_8\}$	69	111.1...	232	$\{a_4, a_6, a_7, a_8\}$
35	111....1	225	$\{a_1, a_6, a_7, a_8\}$	70	1111....	240	$\{a_5, a_6, a_7, a_8\}$

Table 6.2: Combinations generated using lexicographical order

up to 70 *coalitions* to be evaluated, so we can schedule the work distributing 35 to first processor, and the other 35 to the second processor.

While the first processor is calling the function $getCombination(8, 4, 1)$, the second one is calling the same function but with $k = 35$, i.e. $getCombination(8, 4, 35)$. Both processors make the appropriate computation and update the results if it is needed. After finishing, they calculate the next *coalitions* by using the $getNext()$ function.

6.2.1 The $getCombination(n, m, k)$ function

$getCombination(n, m, k)$ calculates the kth *coalition*. It receives parameters n, m and k, where n is the total number of agents, m is the desired size of the *coalitions* and k is the cardinality of the desired *coalition* according to the lexicographical order (first column of Table 6.2).

This function computes the *coalition* bit by bit. Algorithm 3 presents this pseudocode.

Algorithm 3 $getCombination(n, m, k)$ pseudocode

```
$ret \leftarrow 0$
for $current\_bit \leftarrow [1..n]$ do
    if $(m > 0)$ then
        $unassigned\_bits \leftarrow n - current\_bit$
        $highb \leftarrow \binom{unassigned\_bits}{m-1}$
        if $(k \leq highb)$ then
            $ret \leftarrow setBitOn(ret, currentbit)$
            $m \leftarrow m - 1$
        else
            $k \leftarrow k - highb$
        end if
    else
        return ret
    end if
end for
return ret
```

The rationale behind the algorithm is described as dollows:

- At the beginning the *coalition* is still undefined, and there are $\binom{n}{m}$ possible *coalitions* that could be formed by n bits with m of them activated. Half of the combinations have the less significant bit selected, the other half do not.
- For each consecutive bit, the algorithm checks how many bits have been selected so far, and how many elements are still unassigned. These two values can be used to compute a new binomial coefficient which determines whether the next bit is selected or not.

Example
n=8,m=4,k=2.

First iteration, $current\_bit \leftarrow 1$
We compute $unassigned\_bits \leftarrow current\_bit - 1$ and then $\binom{unassigned\_bits}{m-1} = \binom{7}{3} = 35$. Since $k \leq 35$ $(k = 2)$ the first bit is set to 1.
ret ← 0000000**1**
m ← 3

Second iteration, $current\_bit \leftarrow 2$
We compute $unassigned\_bits \leftarrow current\_bit - 1$ and then $\binom{unassigned\_bits}{m-1} = \binom{6}{2} = 16$. Since $k \leq 16$ ($k = 2$) the second bit is set to 1.
ret ← 000000**1**1
m ← 2

Third iteration, $current\_bit \leftarrow 3$
We compute $unassigned\_bits \leftarrow current\_bit - 1$ and then $\binom{unassigned\_bits}{m-1} = \binom{5}{0} = 5$. Since $k \leq 5$ ($k = 2$) the third bit is set to 1.
ret ← 00000**1**11
m ← 1

Fourth iteration, $current\_bit \leftarrow 4$
We compute $\binom{n-1}{m-1} = \binom{4}{0} = 1$.
Since $k > 1$ ($k = 2$), we decrement k in $\binom{5}{0}$.
k ← k−1 = 1

Fifth iteration, $current\_bit \leftarrow 5$
We compute $unassigned\_bits \leftarrow current\_bit - 1$ and then $\binom{unassigned\_bits}{m-1} = \binom{3}{0} = 1$. Since $k \leq 1$ ($k = 1$) the fifth bit is set to 1.
ret ← 000**1**0111
m ← 0

In the next iteration m=0, therefore the returned value is 00010111.

The cost of calling the function $getCombination(n, m, k)$ is high, since for every bit it is needed to compute a binomial coefficient calculation.

6.2.2 The $getNext(m, n, previous)$ function

Given a specific *coalition*, it is possible to compute the next *coalition* in lexicographical order at a lower cost (in an efficient way). This work will be done by the $getNext(m, n, previous)$ function.

Function $getNext(m, n, previous)$ first looks at the most significant bit.

- If the most significant bit is zero, the next *coalition* is computed by adding one bit in the position of the highest bit, e.g.

  $getNext(8, 4, 00\mathbf{1}10101_{/b})$ will return $00\mathbf{1}10101_{/b}$+$00\mathbf{1}00000_{/b}$=$0\mathbf{10}10101_{/b}$

- If the most significant bit is one, the first 01 sequence has to be found starting from the most significant bit. We need to count all the 1s we skip. Once the sequence 01 is found it will be replaced by 10. All the 1s found so far are added to the left of the substituted sequence. e.g.

  $getNext(8, 4, 11100\mathbf{01}_{/b})$ will return the substituted sequence $000000\mathbf{10}_{/b}$ + the skipped ones $000\mathbf{111}00_{/b} = 000\mathbf{11110}_{/b}$

Algorithm 4 $getNext(n, m, previous)$ pseudocode

```
$p1 \leftarrow msb\_position(previous)$
if $p1 \neq n$ then
    $ret \leftarrow previous + 1\ shl\ (p1 - 1)$
else
    $p2 \leftarrow first\_01\_position(previous)$
    $mask \leftarrow 1\ shl\ (p2 + 1) - 1$
    $sequence\_01\_plus\_01 \leftarrow (mask\ AND\ previous) + 1\ shl\ (p2 - 1)$
    $skipped\_ones \leftarrow count\_bits(NOT\ mask\ AND\ previous)$
    $ret \leftarrow sequence\_01\_plus\_01 + ((1\ shl\ skipped\_ones) - 1)\ shl\ p2$
end if
return ret
```

This method can be efficiently implemented using processor builtin instructions like *population count* or *count leading zeros*, which are available in modern processor instructions set. A pseudo-code for this algorithm is presented in the figure Algorithm 4.

6.3 The Shared Memory FSIDP Algorithm (SMIDP)

As shown in the present chapter, it is possible to effectively distribute the execution of the CSGP in different computation units a sharing memory paradigm. In order to do this distribution we were forced to introduce some changes in the original FSIDP conception about how to generate the *coalitions*.

The functions $getCombination(n, m, k)$ and $getNext(m, n, previous)$ can be used in the sequential version with no impact . Once they are added to the FSIDP Algorithm implementation, the transition to the parallel version is done by adding the appropriate directives to execute the FSIDP in parallel.

Algorithm 5 shows the algorithm MSIDP, where the appropriate calls to the functions *getCombinations* and *getNext* are, as well as the definition of the parallel section. In the in real implementation it is needed to define the scope of the variables. Some of them are global (m, n, value, total_coalitions), and others are private to each Core (coreID, coalition, i, max_value, S1, S2).

Each thread receives as a parameter the combination number where it must start, computed as $threadID \dfrac{total\_coalitions}{total\_threads} + 1$.

In the previous example we have 70 total coalitions and 2 threads. The first one (threadID=0) would start by the coalition number 1, the second thread (threadID=1) would start by the coalition number 36.

Algorithm 5 Pseudo-code of the SMIDP Algorithm

```
for $m \leftarrow [2 \ldots n]$ do
    PARALEL_SECTION BEGIN
    $coalition \leftarrow getCombination(n, m, threadID\frac{total\_coalitions}{total\_threads} + 1)$
    for $i \leftarrow [1 \ldots total\_coalitions]$ do
        $max\_value \leftarrow value[coalition]$
        $(S1, S2) \leftarrow init()$
        for $(S1, S2) \leftarrow nextSplit(coalition, (S1, S2))$ do
            if $(sizeOf(S1) \geq n - m)$ then
                if $(max\_value < value[S1] + value[S2]$ then
                    $max\_value \leftarrow value[S1] + value[S2]$
                end if
            end if
            $value[coalition] \leftarrow max\_value$
        end for
        $coalition \leftarrow getNext(n, m, coalition)$
    end for
    PARALEL_SECTION END
end for
```

6.4 Experimentation

We have run different CGSP using the SMIDP algorithm with different configurations. CSGPs size ranges from $n = 18$ to $n = 26$, and we have tested the configuration using 6, 12, 24 threads.

The computer executing the experiments is defined in Section 3.3. It has two six-core Intel Xeon X5645 Processors at 2.4GHz and 96GB RAM. The Processors have the Hyper-Threading Technology enabled.

Figure 6.2 presents the results of the different executions and the *SpeedUps* reached compared to the FSIDP version.

Execution using 6 threads

In the 6-threaded version, the operating system ideally schedules the 6 threads in the same physical processor, i.e. one thread per core.

We can appreciate a growing speedup until the problem size is $n = 21$. The algorithm uses the 6 cores available in the first physical processor, therefore the cache memory is shared between all the cores.

In the sequential versions the cache was used by only one core. When the CSGP fits in the cache ($n \leq 21$) there is no negative impact on sharing cache space. On the contrary, bigger CSGPs decrement locality producing higher miss ratios.

In addition, there are a number of factors that explain not reaching the top *SpeedUp* of 6X:

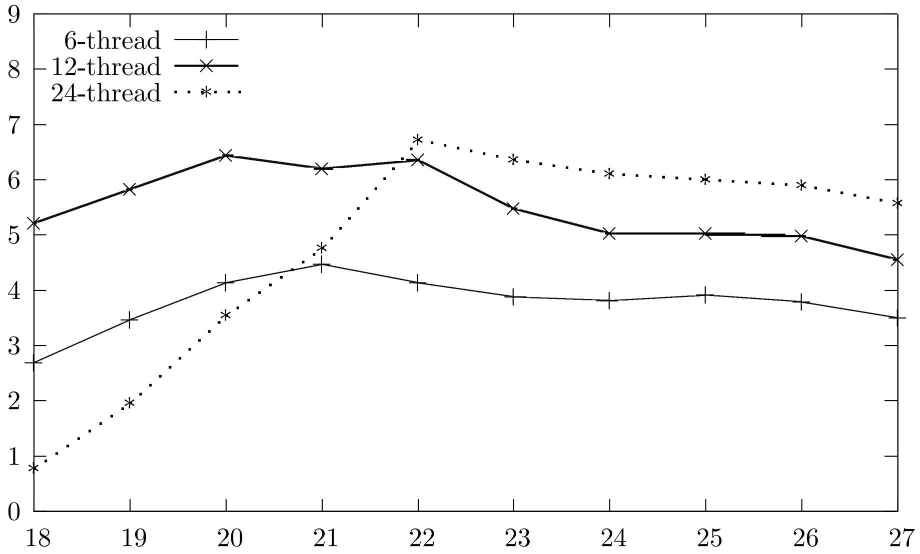

Figure 6.2: Speedup obtained for different thread configurations

- First, the synchronization barriers drive the execution, making some cores to be idle whilst others are computing. Within every iteration of the external loop, all the threads must be synchronized. Even though the distribution of instructions between the different threads is balanced, the execution time of the different threads is unknown.
- Second, according to Amdahl's Law, the *SpeedUp* is constrained by the portion of sequential code executed. And in fact, there is a portion of sequential code in charge of making the corresponding initializations and the computation for driving the execution. The sequential part is less relevant when the CSGP is bigger.
- Third, the effect of the turboBoost. When the processor uses only one core, it can work at a higher clock frequency than when it uses all the cores.
- Finally, the effect of sharing the last level cache between all the cores.

Execution using 12 threads

In the experiment run with 12 threads, the OS scheduler maps each thread to a single core across the two processors.

The behavior follows the same pattern as in the 6-thread execution, but this time *SpeedUp* is around 6, far from the ideal 12 provided by the 12 cores.

Our computer has a Non-Uniform Memory Access (NUMA) architecture, which means that the memory system does not provide the same latency for the two physical processors. In our case, the memory allocation is done by the master thread in the first processor of the system, so all the memory is attached to the first physical processor. Threads executed in the second processor have to access the memory through the first one, reducing the potential gain provided by the parallelization.

Execution using 24 threads

Executing CSGP using 24 threads takes advantage of the Hyper-Threading Technology. This technology allows a single core to execute two threads using the same computational units. Hyper-Threading multiplexes physical resources to hide instruction latencies.

From the results, we can observe that for small CSGPs (less than $n = 21$), the algorithm perform worse than in the 6-thread and the 12-thread configuration. This is caused by the effect of having too many threads for that small amount of work (barely 2-3 seconds of total execution). The OS scheduler does not assign threads correctly to cores and there is a high context switching which is killing the performance. However, when CSGP is bigger, threads have more work and this effect disappears.

Moreover, Hyper-Threading performs better when the CPI is higher and, as seen in Figure 5.5, CPI increments when the problem is bigger than $n = 21$. A high CPI means some computational units are underused, therefore the multiplexing produced by the Hyper-Threading Technology increment the utilization of the hardware resources.

As seen in the previous chapter, one of the main constrains to reach more performance is the memory access pattern. In the case of the parallel versions, a higher number of Threads imply more memory requests per unit of time, and it leads to a big Memory Bandwidth. Hyper-threading is getting benefit of this situation.

However, we can appreciate that the *SpeedUp* is decreasing when the problem size increases. This effect is present for all thread configurations. When using threads, some resources such as the last level cache or the memory buses are shared between all the cores. In consequence, it is reducing the performance of each individual core.

Chapter 7

Summary, Conclusions and Future Work

7.1 Summary

The main objective of this work is to solve CSGPs faster than current state of the art approaches. To this end, we have studied different existing methods to solve CSGPs. We have chosen the DP and IDP algorithms. They are the best when working with random input distributions.

We have studied both algorithms, defined the bounds they have, then coded and measured them.

We have found the main bottlenecks that algorithms have, and we propose an alternative method to solve this bottleneck: the Fast Split Method.

By virtue of the Fast Split Method we have proposed two new algorithms. FSDP and FSIDP. Both increment the *SpeedUp* considerably. We have analyzed the performance of the FSD and FSIDP algorithms extensively.

Finally, we have studied the possibility of implementing a version of the FSIDP in a multicore environment. Hence, we propose a new algorithm which enables the division of the CSGPs in different threads.

Figure 7.1 shows the different versions with their corresponding *SpeedUp* obtained.

7.2 Conclusions

In this work we have studied how to optimize and parallelize an algorithm to solve the Coalition Structure Generation Problem (CSGP). CSGPs are problems which need a large computation effort to be solved. Different techniques have been developed in order to solve them and, in order to be able to deal with bigger CSGPs, some of them only work with CSGPs that present a given

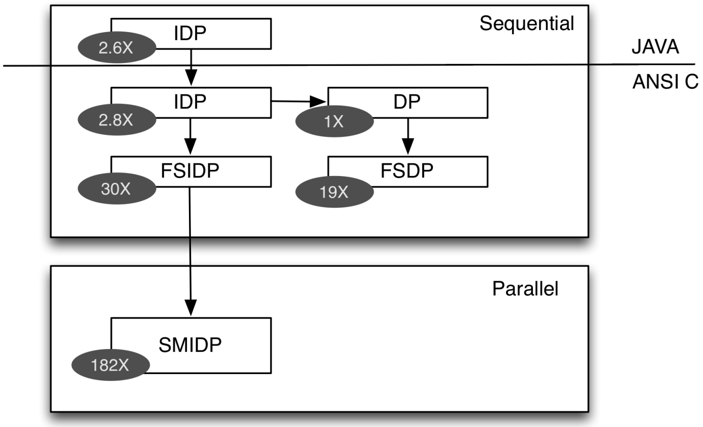

Figure 7.1: Roadmap with corresponding *SpeedUp*s compared to DP

pattern or distribution.

Our efforts are focused on solving any kind of CSGPs and doing it as fast as possible. For this reason we have studied DP and IDP algorithms and we have characterized both algorithms analytically and experimentally.

We found that existing implementations neglect the underlying architecture where the algorithms are executed, and that seems to be a common factor in the related literature we found.

We have proposed a novel method (FSM) to split *coalitions* faster. This operation is key to increasing the performance when solving CSGPs. Hence, we have developed two different implementations: FSDP and FSIDP, which run considerably faster than DP (up to 30 times faster).

CSGPs require a high amount of memory to be represented. In addition, DP and IDP algorithms access the memory without presenting any ordered pattern. This lack of spatial locality has a clear impact on the performance of the application.

We have also developed a technique to divide the execution in different threads, using a Shared Memory paradigm. We have reached a *SpeedUp* of 6x (compared to FSIDP) in a multicore environment due to parallelization on a 12-core, 2 socket processor.

Last version obtained (SMIDP) reaches a high *SpeedUp* compared to other state of the art algorithms. We have analyzed the causes that constrain the execution: address computation and memory latency. These problems are associated with our multicore processor, therefore a migration to GPU can help us to deal with both problems.

7.3 Future Work

The analyzed algorithms present a high cache miss ratio which degrades the performance of the solver. An interesting future line of research is to study the possibility of altering how the CSGP is mapped to memory.

The technique used to implement the Shared Memory parallelization can be reused to divide the execution among different computation nodes connected by a network using a Message Passing Parallel Language (e.g. MPI). This new approach is very interesting in terms of the potential scalability.

As was pointed in the Conclusions section, building an implementation of our algorithms for a GPU environment can lead to a big *SpeedUp*. Modern GPUs offer the possibility of massive parallelization which can lighten the computation phase and enable higher memory bandwidths, offering a good scenario for running our algorithms.

Alternatively, our algorithms do not consider any distribution in the input data. This is positive in the sense that the algorithms can solve any kind of CSGP. However, for those CSGPs whose data does have a concrete distribution, some techniques can be applied in order to skip computation. FSDP and FSIDP can be adapted to work with different input data distribution.

As a final consideration, we found that there is a considerable number of existing algorithms working with analog concepts which not prepared for exploiting parallelization. We can export and reuse the techniques used in FSDP and FSIDP and especially the methodology and apply them to other similar algorithms.

Bibliography

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[2] J.P. Kahan and A. Rapoport. *Theories of coalition formation.* L. Erlbaum Associates, 1984.

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

[4] T. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N.R. Jennings. 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, 2010.

[5] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

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

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

[8] Talal Rahwan and Nicholas R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, pages 1417–1420, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[9] Talal Rahwan, Sarvapali D. Ramchurn, Viet Dung Dang, Andrea Giovannucci, and Nicholas R. Jennings. Anytime optimal coalition structure generation. In *In Proceedings of the 22nd National Conference on Artificial Intelligence*, 2007.

[10] S.J. Rassenti, V.L. Smith, and R.L. Bulfin. A combinatorial auction mechanism for airport time slot allocation. *The Bell Journal of Economics*, pages 402–417, 1982.

[11] Michael H. Rothkopf, Aleksandar Pekec, and Ronald M. Harstad. Computationally manageable combinatorial auctions, 1998.

[12] Tuomas W. Sandholm and Victor R. Lesser. Coalitions among computationally bounded agents. *Artificial Intelligence*, 94:99–137, 1997.

[13] Sandip Sen, Ip Sen, and Partha Sarathi Dutta. Searching for optimal coalition structures. In *Proceedings of the Fourth International Conference on Multiagent Systems*, pages 286–292. IEEE, 2000.

[14] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation, 1998.

[15] Zach Smith. Bandwidth: a memory bandwidth benchmark. `http://zsmith.co/bandwidth.html`, May 2012.

[16] Thomas Voice, Sarvapali Ramchurn, and Nick Jennings. On coalition formation with sparse synergies. 2012.

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