

This is the **accepted version** of the article:

Cruz Mencía, Francisco; Cerquides Bueno, Jesús; Rodríguez-Aguilar, Juan A.; [et al.]. «Coalition structure generation problems : optimization and parallelization of the IDP algorithm in multicore systems». Concurrency and Computation : Practice and Experience, Vol. 29, Issue 5 (March 2017), art. e3969. DOI 10.1002/cpe.3969

This version is available at <https://ddd.uab.cat/record/203087>

under the terms of the  **IN** COPYRIGHT license

Coalition structure generation problems: optimization and parallelization of the IDP algorithm in multicore systems

Francisco Cruz¹ * Antonio Espinosa², Juan C. Moure², Jesús Cerquides¹, Juan A. Rodríguez-Aguilar¹, Kim Svensson³, Sarvapali D. Ramchurn³

¹*Institut d'Investigació en Intel·ligència Artificial -CSIC- Spain*

²*Computer Architecture and Operating Systems Department. Universitat Autònoma de Barcelona. Spain*

³*School of Electronics and Computer Science, University of Southampton, Southampton*

SUMMARY

The Coalition Structure Generation (CSG) problem is well-known in the area of Multi-Agent Systems. Its goal is to establish coalitions between agents while maximizing the global welfare. Among the existing different algorithms designed to solve the CSG problem, DP and IDP are the ones with smaller temporal complexity. After analyzing the operation of the DP and IDP algorithms, we have identified which are the most frequent operations and propose an optimized method. In addition, we study and implement a method for dividing the work into different threads. To describe incremental improvements of the algorithm design we first compare performance of an improved single CPU core version where we obtain speedups ranging from 7x to 11x. Then, we describe the best resource usage in a multi-thread optimized version where we obtain an additional 7.5x speedup running in a 12-core machine.

Copyright © 2010 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: *Concurrency Computat.: Pract. Exper.*

1. INTRODUCTION

Coalition formation is one of the most important coordination mechanisms in multi-agent systems. It usually involves the coming together of a number of disjoint autonomous agents to achieve some individual or group objectives. Coalition formation can be applied to many actual world problems such as distributed vehicle route planning, task allocation [1], airport slot allocation [2] and social network analysis [3]. According to [4], the coalition formation process is divided into three main activities:

- Coalition structure generation (CSG).
- Solving the optimization problem of each coalition.
- Dividing the value of the generated solution among agents.

*Correspondence to: Institut d'Investigació en Intel·ligència Artificial -CSIC- Spain. E-mail: tito@iia.csic.es

The CSG problem is an important step whose major challenge is the search for an effective set of coalitions that maximises the social welfare [1]. The CSG problem is a hard combinatorial optimisation problem whose search space has exponential complexity in the number of agents. This makes it a computationally complex problem [5].

First, the input size is exponential in the number of agents. The input to a coalition structure formation algorithm needs to store the values of the coalitions. For a given number of n agents, we have $2^n - 1$ coalitions. Another problem is that the number of coalition structures to evaluate grows faster than the number of coalitions, being in the order of n^n .

To address the CSG problem, there are several algorithms in the literature that take different perspectives [6]. They are typically grouped in three categories:

- Dynamic programming. They offer guaranteed run times over arbitrary coalition value distributions. Examples are DP [7] and IDP [8] algorithms.
- Complete algorithms with anytime properties. Their time to solution largely depends on the coalition value distribution. Examples are the IP [9], IP-IDP [10] and D-IP [11].
- Heuristics approximation algorithms that aim at computing solutions faster than complete algorithms without offering quality guarantees [12, 1].

Each category has its advantages and drawbacks. DP can be seen as a brute-force algorithm which explores all the possible combinations with the penalty of repeating some work. Taking DP as a starting point, the scientific community has proposed new algorithms that can solve the Coalition Formation problem faster. From here, IDP accomplishes this mission by removing the redundant work done in DP. Both DP and IDP are able to solve the problem regardless of the data distribution the problem encodes, meanwhile all IP, IDP-IP and D-IP algorithms solve the Coalition Formation problem assuming that the input data follows a given distribution (i.e normal or a uniform distribution).

In practice, the computational costs for complete algorithms are highly demanding even for a moderate number of agents. Both heuristic and anytime algorithms are designed to provide a non-optimal but fast solution. A complete reference of the Coalition Structure Generation topic can be found at [13].

Our main motivation is to show how one of these algorithms can be adapted to explore the full capabilities of current computer architectures, leading to significant speedups. In this work, we are evaluating the performance and propose the optimisation of DP and IDP algorithms to solve the CSP problem. Both have a described time complexity in the order of $O(3^n)$ and are not the fastest algorithms in all scenarios, but are not affected by the input data distribution

Sequential computation for dynamic programming algorithms has been extensively studied. Also, there are many contributions to parallelize this kind of algorithms. Most works focus on the optimization or the extension of a sequential programming model for a given target computer architecture. For instance, tools are provided for exploiting pipeline parallelism [14]. In this work, the programmers are given a virtually infinite pipeline of algorithm stages that will be automatically mapped to the actual target number of processors. General approaches have been presented, [15] where parallelization strategies are classified as bottom-up or top-down and propose a generalized way of approaching the parallelization of dynamic programming algorithms.

Our IDP implementation defines a bottom-up dynamic programming approach as coalitions of size $m+1$ are evaluated after analysing all coalitions of size m . We propose a serial reference implementation that defines a 2^n array where coalitions are codified into 4 byte words. The array index represents the actual coalition and each element stores its associated value.

We propose a thread-level strategy to parallelize IDP algorithm. As we must find the splittings of the coalitions of a specific size and the evaluation of every single coalition is independent from the evaluation of the rest of coalitions, we are going to assign an individual thread to manage a disjoint set of coalitions. That is, all parallel threads will receive scattered starting points in the shared array of coalitions. Then, each thread will evaluate an arbitrary number of consecutive coalitions using the sequential algorithm.

We are also proposing a strategy method to generate the k_{th} coalition of m elements from a set of n , so that we can quickly find the data structure locations for distributing the dynamic programming work among the threads.

Our CSG parallelization strategy is performance oriented and proposes the use of an integer array for representing the problem space. Obtaining the following benefits:

- Shared one-dimensional integer array representation of the problem is more compact than a hash table. For the CSG we can precisely define the amount of memory we need to represent the problem and do not worry about hash overheads like index collisions, resizing and extra memory space.
- The work distribution to the different threads can be performed statically: every thread is able to compute its own piece of work by only knowing his thread identifier.
- Array problem representation can better fit caches for small problem instances, but for larger problems any data structure will suffer memory contention as reutilization of data structure items are increasingly scattered in time.

We present a parallel implementation optimised for shared-memory computer systems. Using a shared memory system, application logic is simplified since there is no need for explicit communication between computational nodes. However, we need to carefully study algorithm data dependencies to avoid possible synchronization problems.

There have been previous attempts to parallelize CSG problems. For instance, it is worth mentioning the work by Michalak et al. [11]. They proposed a distributed variant of the anytime IP algorithm called D-IP. In this work, a distributed computer system composed of 14 nodes was used to speed up the execution of a particular CSG implementation. Michalak et al. show that the distributed execution of IP reduces its computational time down to 5% in the best scenarios.

In this work, we expose the potential of current multicore architectures, using only a single computational system similar to one node of the cited Michalak work, we can get a performance optimization of 10x. Then, our proposal can be used as the basic application block that can be further distributed using Michalak proposal.

The main contributions of this paper are:

- We identify the most critical operations of both DP and IDP. We evaluate alternative implementation methods to show that they can improve both algorithms' performance by an order of magnitude.

- We parallelize the IDP algorithm on a shared-memory, multi-core processor system.
- We study the main performance bottleneck of both the sequential and parallel implementations. We find that for both implementations the memory access pattern lacks of temporal and spatial locality. Because of the weak support for irregular and scattered accesses provided by current memory hierarchies, this lack of locality hinders performance.
- We find out that the parallel solutions that we propose significantly outperform the state of the art. Our multi-threaded version speeds up IP's execution time up to a factor of 50. In particular, for 27 agents the execution time drops from two and a half days to 1,2 hours. In general, we think that the methodology and techniques presented in this work are useful and applicable to many dynamic programming algorithms and can be used as common guidelines to implement scalable algorithms for combinatorial problems.

The paper is organized as follows. Section 2 introduces the CSG problem and describes the state of the art on dynamic programming techniques. Section 3 analyzes CPU implementation issues such as data representation, most frequent operations and bottlenecks in a single core environment and proposes solutions to reduce execution time. Section 4 studies how to parallelize the IDP algorithm and Section 5 evaluates the performance of single and multi-core implementations. Finally, Section 6 concludes and outlines future work.

2. THE COALITION STRUCTURE GENERATION PROBLEM

In this section we describe what a CSG problem is and how dynamic programming finds an optimal solution to it.

Table I shows an example of the input data of a CSG problem with 4 agents. The goal is to discover the combination of disjoint coalitions that maximise the value of the sum of their respective associated values.

The CSG problem can be formalized defining $A = \{a_1, a_2, \dots, a_n\}$ as a set of agents where $|A| = n$. A subset $C \subseteq A$ is termed a coalition. Then a CSG problem is completely defined by its characteristic function $value : 2^A \rightarrow \mathbb{R}$ (with $value(\emptyset) = 0$) which assigns a real value representing utility to every feasible coalition. The CSG problem is to identify the exhaustive disjoint partition of the agent space into not empty coalitions (or coalition structure) $CS = \{C_1, C_2, \dots, C_k\}$ so that the sum of values $\sum_{i=1}^k value(C_k)$ is maximised.

From table I, the algorithms must find what coalitions belong to the optimal coalition. For example, the coalition formed by $\{a_2, a_3\}$ will not be part of the optimal coalition structure since $value[\{a_2, a_3\}] = 36$ and $value[\{a_2\}] + value[\{a_3\}] = 52$.

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

Table I. Coalition values for a CSG problem among 4 agents.

The DP[7] and IDP[8] algorithms have been applied to solve the CSG problem. DP and IDP can find an optimal solution to the CSG problem. For that, they explore the complete solution space with complexity $O(3^n)$. IDP shows a notable performance improvement and reduces the hardware requirements of DP. As IDP is an extended version of DP, both algorithms share a very similar structure.

Next, we present an initial description of DP and then describe the improvements introduced by IDP. First of all, we are defining some terminology:

- Agents (A): the set of all available agents. $A = \{a_1, a_2, \dots, a_n\}$.
- Agent (a_i): denotes single agent, where i indicates the agent identifier.
- Coalition (C): $C \subseteq A$. C is a non-empty subset of A that contains the agents participating in a coalition. Its size is defined as the number of agents forming the coalition.
- Split : the operation performing a partition of a coalition C in (C_1, C_2) where $|C_1|, |C_2| > 0$, $C_1 \cup C_2 = C$, $C_1 \cap C_2 = \emptyset$.
- Splitting : the result of the split operation. A splitting is a 2-tuple represented by (C_1, C_2) .
- Coalition Structure (CS): a collection of disjoint coalitions whose union yields the entire set of agents.

2.1. The DP algorithm

Given the set of agents and their coalition values as the input, DP starts by evaluating all coalitions of size 2. In general, it proceeds by evaluating all coalitions of size $m + 1$ after completely evaluating all coalitions of size m . This process will be repeated until m is equal to the size of the set of agents ($|A|$).

Once this process is finished, DP will have found the value of the optimal Coalition Structure, and not the optimal Coalition Structure itself. However, as we will show soon after, the Coalition Structure will be easily found thanks to the reuse of intermediate results. This operation will have a low computational cost.

Algorithm 1 and Table II show the specific details of its function and a trace of a DP execution using table I as the input data.

The DP algorithm activity shown in algorithm 1 is defined by three main nested loops

- The outer loop, between lines 1 and 14, where coalition size is determined.
- The intermediate loop, between lines 2 and 13, where coalitions of a fixed size are generated.
- The inner loop, between lines 5 and 11, where each coalition is split and evaluated. First splitting is generated by `getFirstSplit` function in line 4. From there, the rest are generated by the `getNextSplit` function in line 10. Lines 7 to 9 assess the value of the best splitting. Finally, the value is stored in memory in line 12.

As an example of this process, table II shows the size of the selected coalitions of each step along the first column. We start by those coalitions with $size = 2$ and finish when size $|A|$ is met.

For each size m selected, we show in the second column all possible coalitions evaluated. DP knows the associated coalition value of each coalition C that is shown in the third column. From here, DP enumerates all the possible splittings (C_1, C_2) of each coalition C . These are shown in the fourth column of the table. The total number of splittings for a coalition of size m is $2^{m-1} - 1$. For

Algorithm 1 Pseudo-code of the DP algorithm

```

1: for  $m = 2 \rightarrow n$  do
2:   for  $C \leftarrow \text{coalitionsOfSize}(m)$  do  $\triangleright \binom{n}{m}$  iterations
3:      $\text{max\_value} \leftarrow \text{value}[C]$ 
4:      $C_1 \leftarrow \text{getFirstSplit}(C)$ 
5:     while ( $\text{sizeof}(C_1) > 0$ ) do  $\triangleright 2^{m-1} - 1$  iterations
6:        $C_2 \leftarrow C - C_1$ 
7:       if ( $\text{max\_value} < \text{value}[C_1] + \text{value}[C_2]$ ) then
8:          $\text{max\_value} \leftarrow \text{value}[C_1] + \text{value}[C_2]$ 
9:       end if
10:       $C_1 \leftarrow \text{getNextSplit}(C_1)$ 
11:    end while
12:     $\text{value}[C] \leftarrow \text{max\_value}$ 
13:  end for
14: end for

```

m	C	$\text{value}[C]$	Splittings (C_1, C_2)	Value of splittings $\text{value}[C_1] + \text{value}[C_2]$	max
2	$\{a_1, a_2\}$	87	$\{a_1\}, \{a_2\}$	72	87
	$\{a_1, a_3\}$	87	$\{a_1\}, \{a_3\}$	46	87
	$\{a_1, a_4\}$	70	$\{a_1\}, \{a_4\}$	73	73
	$\{a_2, a_3\}$	36	$\{a_2\}, \{a_3\}$	52	52
	$\{a_2, a_4\}$	52	$\{a_2\}, \{a_4\}$	79	79
	$\{a_3, a_4\}$	67	$\{a_3\}, \{a_4\}$	53	67
3	$\{a_1, a_2, a_3\}$	97	$\{a_1\}, \{a_2, a_3\}$	85	126
			$\{a_2\}, \{a_1, a_3\}$	126	
			$\{a_3\}, \{a_1, a_2\}$	100	
	$\{a_1, a_2, a_4\}$	111	$\{a_1\}, \{a_2, a_4\}$	112	127
			$\{a_2\}, \{a_1, a_4\}$	112	
			$\{a_4\}, \{a_1, a_2\}$	127	
	$\{a_1, a_3, a_4\}$	100	$\{a_1\}, \{a_3, a_4\}$	100	127
			$\{a_3\}, \{a_1, a_4\}$	86	
			$\{a_4\}, \{a_1, a_3\}$	127	
	$\{a_2, a_3, a_4\}$	132	$\{a_2\}, \{a_3, a_4\}$	106	132
			$\{a_3\}, \{a_2, a_4\}$	65	
			$\{a_4\}, \{a_2, a_3\}$	92	
4	$\{a_1, a_2, a_3, a_4\}$	151	$\{a_1\}, \{a_2, a_3, a_4\}$	165	166
			$\{a_2\}, \{a_1, a_3, a_4\}$	166	
			$\{a_3\}, \{a_1, a_2, a_4\}$	140	
			$\{a_4\}, \{a_1, a_2, a_3\}$	166	
			$\{a_1, a_2\}, \{a_3, a_4\}$	154	
			$\{a_1, a_3\}, \{a_2, a_4\}$	166	
			$\{a_1, a_4\}, \{a_2, a_3\}$	125	

Table II. Trace of execution of a problem of size 4

each of these splittings, DP must also compute the sum of the splitting member's coalition values. This is shown in the fifth column of the table. Finally, DP must compare each of these values with the value of the coalition C . If the sum of the splitting values $\text{value}[C_1] + \text{value}[C_2]$ is bigger than the coalition value $\text{value}[C]$, this splitting value becomes the best coalition value. The last column of the table shows the maximum value of a coalition.

We also provide the recurrence description of the algorithm. It describes how to find the coalition structure that provides a maximum value for a given number of agents (n). In every recursive step,

we need to fetch the value of all needed coalitions from a table. Best coalition structure for a given set of agents $\{a_1, \dots, a_n\}$ is found by selecting the structure that maximises its value. For that we have to evaluate the following expressions:

$$f(C) = \max(\{V(C)\} \cup \{f(A^k) + f(C - A^k) \mid A^k \in A_{comb}^k, 1 \leq k \leq |C|\})$$

if $|C|=1$, then $f(C) = V(C)$

Where

f is the function that gives us the best coalition value.

V is the original value of the coalition provided in a table.

C is the input coalition of n agents such coalition is called "grand" coalition.

$A_{comb}^k = \binom{|C|}{k}$ combinations of k elements of C

Once all the coalitions of size m have been evaluated, the algorithm increments m by one, so every possible split contains a pair of evaluated coalitions. Then, the DP algorithm must decide whether it is better to split the coalition or keep it to get the optimal value. With this strategy, DP ensures that the maximal value for the optimal coalition structure is found at the end of the execution.

Although Algorithm 1 does not compute the actual optimal coalition structure, Rahwan & Jennings demonstrated in [8] that maintaining a table for storing the Coalition Structures found so far is not required, thus releasing 33% of the memory required by DP. Instead, once a solution is found, it is possible to compute the corresponding Coalition Structure providing that solution.

After finding the maximal value of the Coalition Structure, the corresponding coalition can be found using the *FindCS* function described by Algorithm 2. It has a complexity of $O(2^n)$, where n is the number of agents involved. This cost is very small compared to the complexity of DP algorithm, $O(3^n)$, and, therefore, the time required to find the actual Coalition Structure after running DP has little computational relevance in the overall process of solving the problem.

Algorithm 2 Find Coalition Structure

```

1: function FINDCS(Coalition)
2:    $CS \leftarrow C$ 
3:    $C_1 \leftarrow getFirstSplit(C)$ 
4:   while ( $sizeof(C_1) > 0$ ) do
5:      $C_2 \leftarrow C - C_1$ 
6:     if ( $value[C_1] + value[C_2] = value[C]$ ) then
7:        $CS \leftarrow FINDCS(C_1) \cup FINDCS(C_2)$ 
8:       break;
9:     end if
10:     $C_1 \leftarrow getNextSplit(C_1)$ 
11:  end while
12:  return  $CS$ 
13: end function

```

2.2. The IDP algorithm

As mentioned above, the DP algorithm can find the optimal solution with $O(3^n)$ complexity. Unfortunately, DP performs redundant calculations [16]. IDP tries to avoid the evaluation of as many redundant splittings as possible, without losing the guarantees of finding the optimal coalition

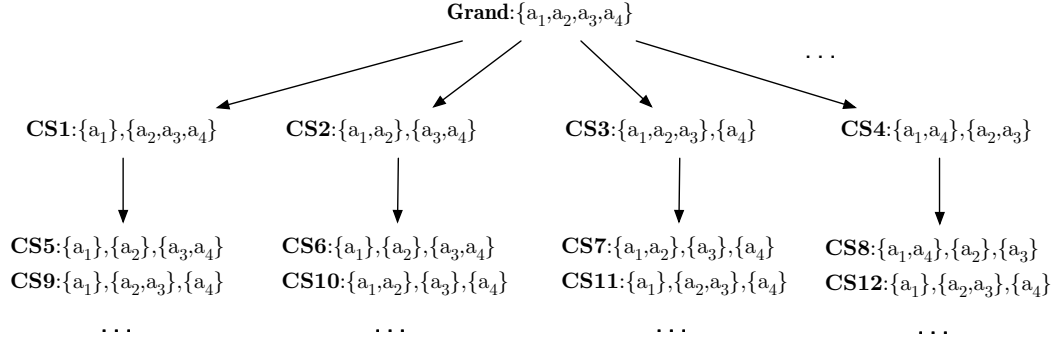


Figure 1. Different splittings leading to redundant work.

structure. It proposes a method to avoid the search paths that will lead us to a state that can also be reached by other paths. As a result, IDP achieves the same functionality as DP but evaluating less combinations. The fraction of splittings evaluated by IDP only depends on the number of agents. For problems between 22 and 28 agents we have found this fraction to range between 38% and 40%.

More specifically, IDP restricts the number of splittings that are generated from a coalition C of size $|C|=m$ in a problem of size n . As we defined before, a splitting of a coalition C is a 2-tuple represented by (C_1, C_2) , where $C_1 \cup C_2 = C$, $C_1 \cap C_2 = \emptyset$, and $|C_1|, |C_2| > 0$. Let s_{max} be the size of the largest member of the splitting, i.e. $s_{max} \leftarrow \max(|C_1|, |C_2|)$. According to the IDP rationale, all splittings satisfying $s_{max} > n - m$ can be safely removed since their exploration leads to redundant work.

We illustrate how DP generates duplicate work by means of figure 1, which shows how different Coalition Structures are generated in a problem of size $n = 4$. In this representation, each level of the tree shows the result of applying a split operation in one of the members of the Coalition Structure of the parent node. A split operation on the Grand coalition on the top of the tree generates 7 different Coalition Structures, but, in the sake of simplicity, figure 1 represents only 4 of them, tagged as CS1, CS2, CS3, and CS4.

A closer look to the figure reveals that after performing two consecutive split operations on the Grand Coalition, the same Coalition Structure, for example CS5 and CS6, can be generated by different ways. CS5 is generated by performing a split on the second member of CS1, while CS6 is the result of splitting the first member of CS2, and they are actually the same Coalition Structure. The example shows more repetitions, like CS9-CS11-CS12 and CS7-CS10. IDP evaluates the condition $s_{max} > n - m$ to prune the splitting operation on CS1 and CS3 (with $s_{max}=2$ and $m=3$), and then avoids the redundant generation of CS5, CS7, CS9, and CS11.

Algorithm 3 presents the pseudo-code of IDP. It is very similar to DP but with differences between lines 4 and 6, where splittings to be evaluated are filtered. The function *IDPBounds* at line 4 finds the bounds on the sizes of the members of the splitting tuple according to the IDP description [8] and the rationale described above. It generates a lower and a higher bound for a given number of agents (n) and coalition size (m).

Algorithm 3 Pseudo-code of the IDP algorithm

```

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

```

For the rest of this paper we are going to use IDP as the reference algorithm. We are going to optimize the algorithm performance and propose a new parallel implementation to reduce the execution time.

3. CPU IMPLEMENTATION

In the next sections we analyse the DP and IDP algorithm computational costs. We focus on how the operation of generating and evaluating splittings affects the inner loop of both algorithms, where most of the execution time is spent.

We have analysed the operations of generating and evaluating splittings and have found that they consume about 99% of the execution time. Algorithms 1 and 3 describe how these operations are located in the inner loop of the algorithms. In this section, we are going to analyse how these operations are executed in a current CPU and present new algorithm design choices to improve the performance in terms of number of instructions executed and requested accesses to memory.

The way data is represented and stored is important, especially in algorithms where loops are dedicated mainly to fetching data from memory and doing some small number of calculations. Selecting the right data structure will substantially reduce the memory requirements of the algorithm. This is the case of Algorithms 1 and 3 where in every iteration of the innermost loop two memory requests are issued and then a small computation is performed; for instance, two additions and one comparison in the case of Algorithm 1.

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

To give an example, in table III we show the representations of the coalition $C = \{a_1, a_2, a_4\}$ with four agents $A = \{a_1, a_2, a_3, a_4\}$. In the table, it is represented as $C = 1101$ in the binary system and

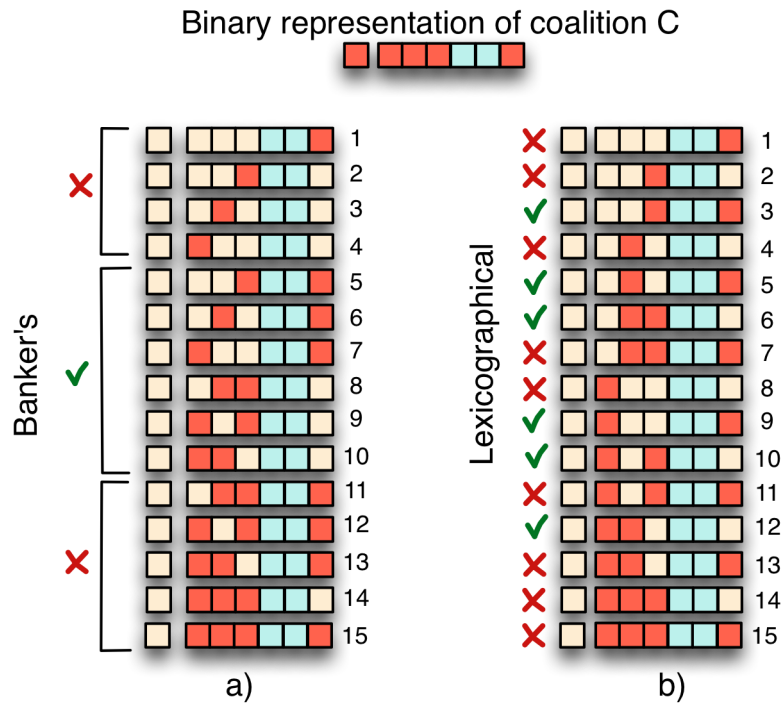


Figure 2. a) Banker's sequence versus b) lexicographical order.

13 in the decimal system. Therefore, if the coalition is represented as an integer it can implicitly be stored as an index to its coalition value.

Order	Encoding	Coalitions
(k)	Bin	Dec
2	1011	11
3	1101	13
4	1110	14

Table III. Example of coalition representation.

3.1. Implementation of the splitting generation operation

Since each coalition splitting is composed by a non-empty subset C_1 and its complement (C_2), generating all the subsets C_1 from a given coalition C will produce the same splitting twice. One for each splitting subset. Instead, we remove one element from the coalition when performing the subset enumeration, so that the removed element is never part of the enumerated subset and always belongs to its complement.

There exist several ways of enumerating subsets [17] like banker's sequence, lexicographical order and gray codes. For the case of DP and IDP, we find banker's sequence in the original algorithms and we propose and justify the adoption of a lexicographical order splitting generation to reduce the execution time of the algorithm.

Based on the description of IDP [8] the banker's sequence seems to be the enumeration technique in use since it generates the splittings in growing order of size of $|C_1|$. Then, simplifying the filtering

of splittings by its member's size. Figure 2a shows an example of the banker's sequence operation for $C=\{a_1, a_4, a_5, a_6, a_7\}$ assuming that only coalitions with $|C_1| = 2$ need to be evaluated. In this case, element a_7 is always assigned to the complementary subset (shown in light colour). Splitting generation starts from the first splitting of size $|C_1| = 2$, followed by the remaining $\binom{4}{2}-1$ subsets of the same size. Then, stops before generating the first subset of size 3. In this way, the code does execute only those instructions required to generate the needed subset.

When generating splittings in lexicographical order (shown in Figure 2b) we need to add an extra computation required to check whether the size of the actual splitting member sizes ranges between a given pair of bounds in order to fit IDP specification. In our example of Figure 2, 8 out of 14 splittings are actually discarded (expressed by a crossed sign).

The intuition drive us to think that banker's sequence should be the best strategy since only the splittings that are going to be evaluated are generated, meanwhile using a lexicographical order approach, we have to generate more splittings and then evaluate all of them in order to keep them or discard them. However we will show that lexicographical order fast generation of splittings combined with their fast evaluation will lead to better results.

Both methods were implemented using recurrent functions that calculate the next splitting from the previous one. The lexicographical order was implemented with a reduced number of very simple operations:

$$C_1(i+1) \leftarrow (C_1(i) + C^{**}) \text{ AND } C,$$

where C^{**} is the two's complement of C , that can be precalculated for all the splittings of a given coalition. The presented implementation containing all splitting generation and filtering requires only 7 instructions in a current x86 ISA CPU. This is far less than an optimized implementation of banker's sequence [17] that required, on average, 6 times more instructions. More implementation details, like the effective use of a special population count instruction for computing $|C_1|$, can be found in the published code [18]. Although it is hard to say if this function can be performed with less than 7 CPU instructions, any possible improvements in reducing the computation of the successor will not have a relevant impact in the overall execution time, as execution performance is mostly limited by memory performance, as we will show in the next subsection.

3.2. Memory hierarchy usage

The management of large state-spaces has already been described as high memory demanding [19] when a distributed memory system is generating a set of state-spaces. It shows that an strategy of balancing memory load can be useful to reduce the variation of memory utilization among the processors of the system. Our approach shares the same view when defining a strategy to balance the memory requests between the threads that access to the shared memory system.

Here, we want to describe the use of memory done by the IDP algorithm. We are focusing on locating opportunities for exposing memory level parallelism. Also, we are evaluating how well the algorithm can reuse the data structures already allocated in memory during the execution. Finally, we use this analysis to describe which is the effective use of the cache memory and the main problems found in its use.

All memory accesses requested by the algorithm correspond to reads from the one-dimensional vector of coalition values issued by the inner loop of the algorithm. The few writes issued in the

intermediate loop can be neglected. The total number of data read operations done by the DP algorithm is around 2×3^n . As explained above, IDP evaluates only a limited number of the splittings, corresponding to 38-40% of the read operations done by DP.

When evaluating the potential for memory-level parallelism in the code we find it to be high. The inner loop iterations are independent (can be potentially executed in parallel), meaning that an out-of-order processor core can issue multiple independent read operations to the memory system in parallel. The actual number of reads that can be issued is subject to the storage availability for pending memory requests and the number of instructions that are blocked waiting for the data.

The data reuse degree of the algorithm is high. There are 2^n elements in the value vector and the average number of reads to the same data item is $\approx 2 \times (3/2)^n$ ($\approx 100,000$ for $n = 27$). However, accesses to the same item are scattered in time when evaluating medium or large size coalitions, so, in practice, the temporal locality of the data accesses is scarce.

The memory access pattern is irregular. All the possible combinations of coalitions have to be evaluated. That means that every time the split operation is performed the CPU needs to fetch the values of the two members of the splitting from the memory system. In most cases, distant positions in the memory system are requested, leading to bad performance. Unfortunately, we cannot improve the spatial data locality of IDP: as all combinations have to be tested in pairs, all possible pairs need to be explored. This means that every possible reorganisation of the data in memory improves locality (and performance) for some pairs but at the same time decreases locality for others.

We have found a bad performance behavior of the memory access pattern when vectors do not fit into the cache memory. The vector size is 2^{n+2} bytes, which sums up 16 MBytes for $n = 22$. For large values of n an important amount of vector accesses will miss the cache and will request a full 64-Byte cache block to be read from DRAM. The high memory-level parallelism described before helps hiding part of the DRAM latency problem, but, due to the lack of spatial locality, there is still an important amount of latency exposed in execution time. The final result is that the effective memory bandwidth consumed by the application is substantially lower than the potential DRAM bandwidth in the system.

4. MULTI-THREAD IMPLEMENTATION

In this section we describe an efficient parallelisation of the IDP Algorithm. Our parallelisation reaches a balanced distribution of tasks on the different computational units. We obtain this work-balance by using a novel algorithm able to select disjoint set of tasks (i.e. coalition set to split) to be performed by every computational unit. As we will show, the new parallel algorithm takes advantage of the multiple computational resources and consequently obtains significant speedups compared to the sequential IDP.

4.1. Identifying sources of parallelism

The first attempt to improve the performance of an iterative program is to first parallelize the outer loop of the program. In our case, DP and IDP are not good candidates: the optimal values for coalitions of size m must be generated before using them for generating the optimal values for coalitions of size $m + 1$.

Alternatively, if we consider the intermediate loop, we see that it generates all coalitions of a specific size and, for each coalition, it analyzes all the splittings of certain smaller sizes. Then, if we could assign each coalition to a different task, we find a large source of potential parallelism as there are no dependencies between each of them. Our work here is to transform a sequential algorithm where each new coalition is generated as the next in lexicographic order into a concurrent source of independent coalition analysis tasks.

4.2. Speeding up work distribution among threads

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

Table IV. Coalitions generated using lexicographical order.

Our starting point is having to deal with a number of threads t where each of them is in charge of analysing a disjoint set of coalitions. We must distribute work to assure good load balance between the threads in a fast and efficient way. Table IV illustrates the generation of all the possible coalitions of size 3 from a set of 6 agents. The sequential algorithm generates all 20 coalitions, $\binom{6}{3}$, represented as bitmaps in the binary encoding columns of Table IV.

There are different algorithms for enumerating coalitions of a given size. As discussed in [20], coalition enumeration can be performed in different orders: using lexicographical or co-lexicographical order, using the cool-lex algorithm [21], using gray codes, using the Eades-McKay strong minimal-change order [22], or endo/enup moves. Some of the mentioned enumeration techniques present some useful characteristics in fields like data communications or error correction. In our case, as we will show immediately, we need to compute the k -th element of the sequence and this can be achieved using lexicographical order.

It is important to state that this operation is not crucial in terms of performance since the aggregated computational time of this enumeration represents less than 1% of the total computational time. Also, it has no impact in memory system performance since all the elements have to be visited anyway, and there is no locality benefits of generating two consecutive coalitions that are close in memory: a massive number of intermediate memory requests are performed by the Split operation between the generation of one coalition and the generation of the next coalition.

In practice, we must calculate $cnt = \binom{n}{m}$, where n is the number of agents and m is the coalition size, and then assign cnt/t coalitions to each thread. Once a thread receives a starting coalition, provided by its position k in the table, it can generate any arbitrary number of coalitions cnt just by following the sequential algorithm. Then, all we need is an efficient strategy to generate the k^{th} coalition without having to compute all precedent coalitions from the beginning.

Algorithm 4 pseudocode of $getCoalition(n, m, k)$

```

1: if  $((m == 0) \text{ OR } (k == 0))$  then
2:   return 0
3: end if
4:  $h \leftarrow \binom{n-1}{m-1}$ 
5: if  $(k \leq h)$  then
6:   return  $1 + 2 \times getCoalition(n-1, m-1, k)$ 
7: end if
8: return  $2 \times getCoalition(n-1, m, k-h)$ 

```

Algorithm 4 describes $getCoalition(n, m, k)$, a function that generates the k^{th} coalition of m elements from a set of n in lexicographical order. The description provided is recursive to help understand how it works, although the actual implementation is iterative in order to improve its performance. The coalition is created recursively, bit by bit, starting from the least significant bit and considering $\binom{m}{n}$ possibilities. Around the first half of the possible coalitions have the least significant bit set to 1. Concretely, if the requested rank, k , is lower than or equal to $h = \binom{n-1}{m-1}$, then the bit is set to 1, and m is decremented by one. Otherwise, the bit is set to zero, and the rank k is reduced to $k - h$. Each recursive call decrements the number of bits to consider to $n - 1$.

Defining the set of coalitions as a list of elements stored in a table, like shown in table IV, all threads receive a k starting disjoint position in the table and has to call $getCoalition$ function. In contrast with the serial implementation, every thread has to compute which is its first coalition to be evaluated.

However, although such calculation introduces an extra overhead, this overhead is negligible. The $getCoalition$ function has a complexity of $O(n^2)$ and is executed always with small values of n (i.e. from 2 to 30). Every time a new coalition size is selected, every thread calls once the $getCoalition$ function and then perform a huge amount of splitting evaluations. For example in problems of size 22 this operation will be performed a total of 21 times per thread. This computation has negligible impact on the performance compared to the 15,686,335,501 splitting evaluations evenly distributed across the threads for the same problem of size 22.

5. EXPERIMENTAL RESULTS: SERIAL AND MULTICORE

We have evaluated the different implementations of the presented algorithms in a single-thread execution, and in a multi-thread execution on a multi-core Intel CPU. DP and IDP were executed using both the banker's and lexicographical splitting generation methods. Our objective is to provide a comprehensive view of the scalability of the algorithms showing important performance improvements on the state-of-the-art implementations of DP and IDP. Additionally, we also describe how the lack of locality in the pseudo-random memory access pattern of the algorithms seriously affects their scalability in multi and many-core platforms.

Note that the data configuration has no impact in the overall performance of either DP or IDP. This is due to the fact that all the possible combinations are explored regardless of the actual input

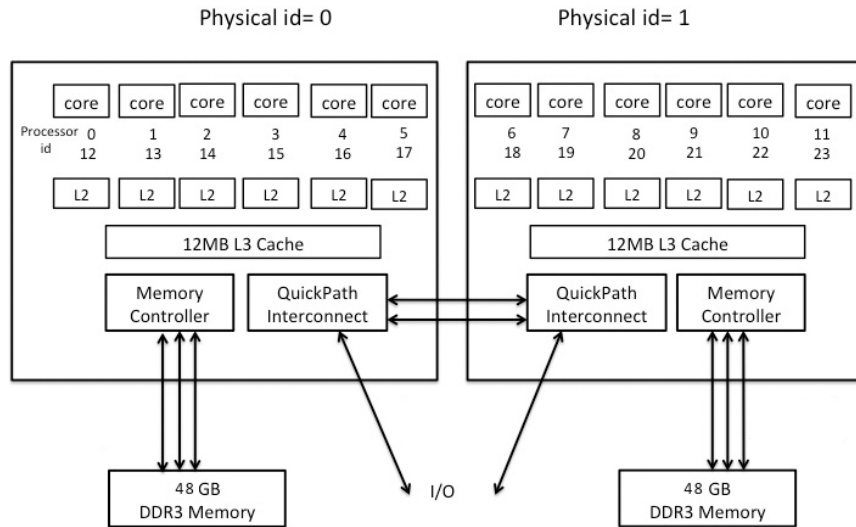


Figure 3. Hardware configuration

data values. Two problems of the same size solved with the same algorithm but different data will perform the same exact amount of operations.

In our work, we have developed C implementations of the mentioned algorithms, using standard GCC (version 4.7) compilers and standard openMP (version 3.1) libraries.

The computer system used in our experiments is a dual-socket Intel Xeon E5645, each socket containing 6 Westmere cores at 2.4 GHz, and each core executing up to 2 hardware threads using hyperthreading (it can simultaneously execute up to 24 threads by hardware). The Level 3 Cache or Last Level Cache (LLC) provides 12 MB of shared storage for all the cores in the same socket. 96 GiB of 1333-MHz DDR3 RAM is shared by the 2 sockets, providing a total bandwidth of 2×32 GB/sec. The Quickpath interconnection (QPI) between the two sockets provides a peak bandwidth of 11.72 GB/sec per link direction.

Figure 3 shows a diagram of the Hardware configuration used. Note, that this two-socket system has a Non-Uniform Memory Access (NUMA) configuration where half of the global memory is directly attached to one socket, and the other half to the other. Although for the programmer there is only one global memory system, some memory requests have to travel from one socket to the other, having its corresponding penalization in terms of CPU cycles.

Input data was created using a uniform distribution as described by [5] for problem sizes $n = 18 \dots 27$. In all the experiments, data is stored in the first memory system socket. This allows fast data access for all the threads in the single-core execution and for half of the threads in the multi-core execution.

5.1. Single-thread execution

Figure 4 plots the execution time in logarithmic scale for the four algorithmic variants. The solution is computed around 7x to 11x faster using lexicographical order rather than banker's sequence. This is due to the fact that although the generation of the splittings using a banker's sequence generates less splittings, the lexicographical order generation is considerably faster. In fact, the number of

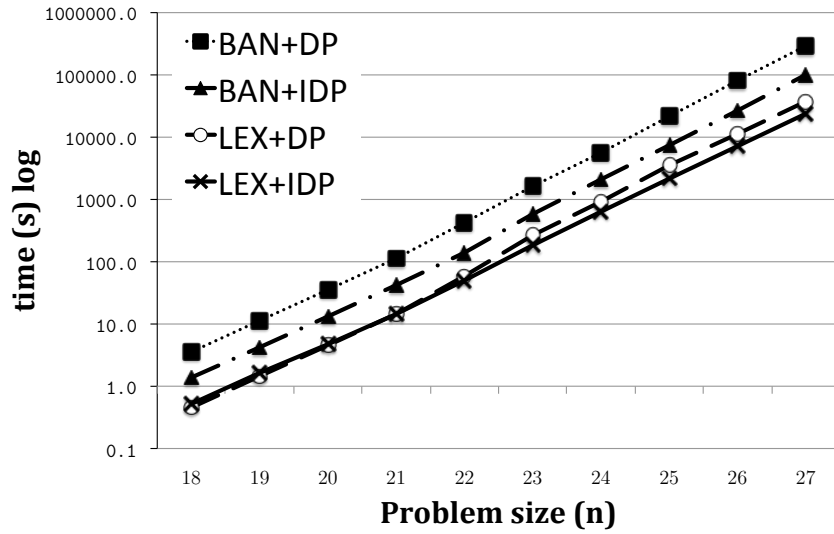


Figure 4. Execution time (log).

instructions executed by the processor is around 6 times lower when using the lexicographical order technique.

Figure 5 represents the execution time of DP and IDP divided by 3^n (algorithmic complexity). This metric evaluates the average time taken by the program to execute a basic algorithmic operation, in this case a splitting evaluation. It is similar to the CPI (Cycles Per Instruction) metric, but at a higher level. The metric helps identifying performance problems at the architecture level. Figure 5 shows two different problem size regions: those that fit into the last level cache ($n < 22$), and those that do not. A small problem size determines a computation-bound scenario, where DP slightly outperforms IDP, even when it executes around 20% more instructions. The reason is that IDP is penalized by a moderate number of branch mispredictions.

Large problem sizes determine a memory-bound scenario, where IDP amortizes its effort on saving expensive memory accesses to outperform DP by 40-50%. Figure 6 shows the effective memory bandwidth consumption seen by the programs. The shape of the curves can be deduced from Figure 5, but we are interested on the actual values. The effective bandwidth ranges between 0.5 and 1.0 GB/sec. A small fraction of this bandwidth comes from the last level and lower-level caches, and the remaining fraction comes from DRAM. Even considering the worst case that only 4 bytes out of the 64-Byte cache block are effectively used, it is still a very small value compared to the peak 32 GB/sec. We conclude, then, that DRAM latency is the primary performance limiter. The multi-thread execution results in the next subsections corroborate these conclusions.

5.2. Multi-thread execution

We focus our multi-thread analysis on IDP, which outperforms DP for interesting larger problem sizes. We run IDP using $t = 6, 12$, and 24 threads. The case $t = 6$ corresponds with using a single processor socket. The case $t = 12$ uses only one socket but also exploits its hyperthreading capability. Finally, $t = 24$ is an scenario where all 2 sockets have their 6 cores running 2 threads each, using hyperthreading. Figure 7 show the speedup results compared to the single-thread execution.

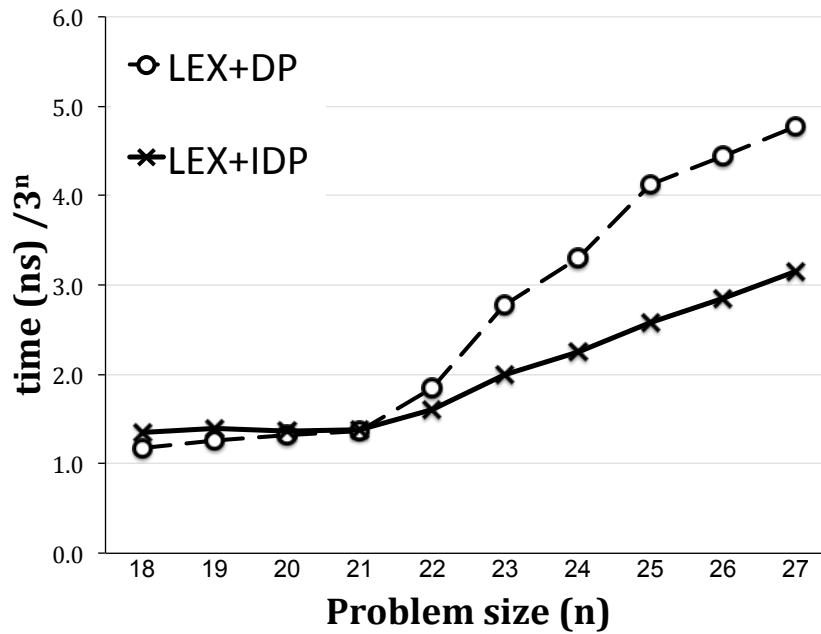
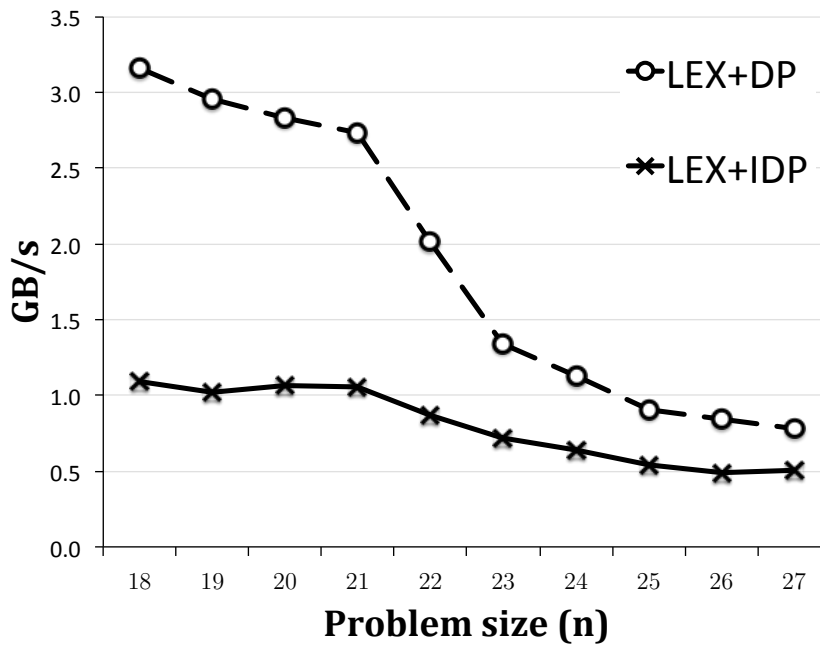
Figure 5. Time / Complexity $\Theta(3^n)$.

Figure 6. Effective Memory Bandwidth (GB/s).

The $t=6$ configuration provides a speedup of 5 for small problems, and lower than 4 for large problems. The $t=12$ configuration further increases performance around 60% for small problems, and 30% for bigger problems. The fact that executing two threads per core does improve performance corroborates previous latency limitations, since hyperthreading is a latency-hiding

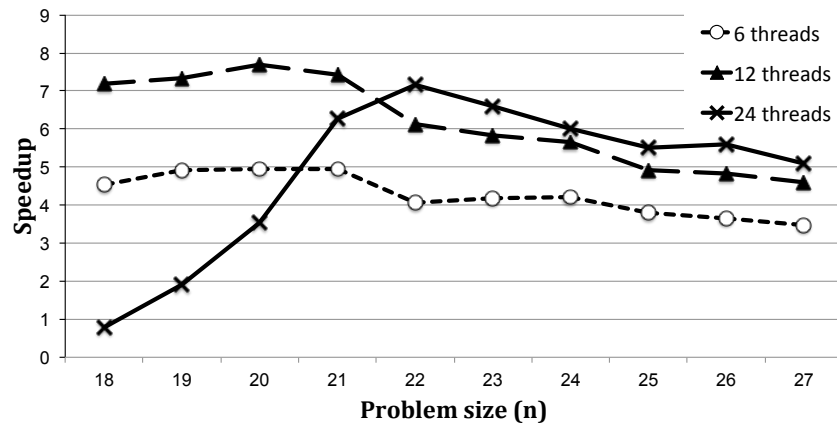


Figure 7. Single-thread IDP versus 6-, 12- and 24-thread IDP execution

mechanism. It also indicates that 6 threads do not generate enough cache memory and DRAM requests to fully exploit the available cache and DRAM bandwidth.

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

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

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

To store coalitions and their associated values we use a one-dimensional array, namely the values vector. The bad performance behavior of the memory access pattern arises for values vectors that do not fit in the processor's cache. For coalitions larger than $n=22$ agents, memory accesses will miss the cache and will request a full 64 Byte cache blocks to DRAM. Also, although data-reuse of the algorithm is high, the accesses to the same item are scattered in time, specially for medium/large problems.

Serial and parallel DP/IDP implementations have the same pseudo-random access pattern to memory. The effect on performance is more evident when problem sizes are large and we start to use a larger number of threads. The global number requests to pseudo random positions in memory makes these algorithms to be latency bound.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an optimized serial implementation and a novel parallel multi-threaded implementation of state-of-the-art dynamic programming algorithms for random problems. Our best serial implementation speeds up previously reported IDP's execution time[8] by an order of magnitude. Furthermore, our multi-threaded version speeds up our sequential version execution time up to 7.5x in a 12 core machine. Combining both techniques, our parallel and optimized version executed using 24 threads is able to find the solution around 50 times faster than the original and sequential IDP version.

Our parallel IDP is the first CSG algorithm designed to run in a multicore scenario. Although it is not the first algorithm able to solve the CSG in parallel, our implementation is the first one able to exploit a multicore scenario efficiently. The previous remarkable parallel algorithm is the distributed anytime algorithm [11], designed to run in a distributed environment. Besides the differences on the hardware required to run both algorithms, our parallel IDP has lower worst-case time complexity which is $O(3^n)$, than the worst-case time complexity of the mentioned distributed anytime algorithm which is $O(n^n)$.

We have analyzed the bottlenecks of DP and IDP finding that their memory access pattern lacks locality. This causes that both algorithms exploit the memory system capabilities very inefficiently. Nonetheless, we have observed that the increased latency tolerance ability exhibited by multi-threading improves performance on a multicore processor.

Finally, we have made publicly available the source code of our serial and multi-threaded implementations [18].

The use of new data structures and alternative algorithms can help to increase the locality of the memory accesses performed. In this way, the objective is to make a more predictable number of memory requests. Then, as part of the future work, we need to study more complex algorithms with higher locality at the expense of doing more calculations than IDP. Then, new solutions will scale better on current multi-core computing platforms.

7. ACKNOWLEDGEMENTS

This research has been supported by MICINN-Spain under contracts TIN2012-38876-C02-01, TIN2014-53234-C2-1-R and the Generalitat of Catalunya (2009-SGR-1434).

REFERENCES

1. Shehory O, Kraus S. Methods for task allocation via agent coalition formation. *Artif. Intell.* 1998; **101**(1-2):165–200.
2. Rassenti S, Smith V, Bulfin R. A combinatorial auction mechanism for airport time slot allocation. *The Bell Journal of Economics* 1982; :402–417.
3. Voice T, Ramchurn SD, Jennings NR. On coalition formation with sparse synergies. *AAMAS*, 2012; 223–230.
4. Sandholm TW, Lesser VR. Coalitions among computationally bounded agents. *Artificial Intelligence* 1997; **94**:99–137.

5. Larson KS, Sandholm TW. Anytime coalition structure generation: an average case study. *J. of Experimental & Theoretical Artificial Intelligence* 2000; **12**(1):23–42, doi:10.1080/095281300146290. URL <http://www.tandfonline.com/doi/abs/10.1080/095281300146290>.
6. Cerquides J, Farinelli A, Meseguer P, Ramchurn SD. A tutorial on optimization for multi-agent systems. *The Computer Journal* 2013; :bxt146.
7. Yun Yeh D. A dynamic programming approach to the complete set partitioning problem. *BIT Numerical Mathematics* 1986; **26**:467–474. URL <http://dx.doi.org/10.1007/BF01935053>, 10.1007/BF01935053.
8. Rahwan T, Jennings NR. An improved dynamic programming algorithm for coalition structure generation. *AAMAS* (3), 2008; 1417–1420.
9. Rahwan T, Ramchurn SD, Dang VD, Giovannucci A, Jennings NR. Anytime optimal coalition structure generation. *AAAI*, 2007; 1184–1190.
10. Rahwan T, Jennings N. Coalition structure generation: dynamic programming meets anytime optimisation. *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, 2008; 156–161.
11. Michalak T, Sroka J, Rahwan T, Wooldridge M, McBurney P, Jennings N. A distributed algorithm for anytime coalition structure generation. *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, International Foundation for Autonomous Agents and Multiagent Systems, 2010; 1007–1014.
12. Sen S, Sen I, Dutta PS. Searching for optimal coalition structures. *Proceedings of the Fourth International Conference on Multiagent Systems*, IEEE, 2000; 286–292.
13. Rahwan T, Michalak TP, Wooldridge M, Jennings NR. Coalition structure generation: A survey. *Artificial Intelligence* 2015; **229**:139–174.
14. González D, Almeida F, Roda J, Rodríguez C. From the theory to the tools: parallel dynamic programming. *Concurrency: practice and experience* 2000; **12**(1):21–34.
15. Stivala A, Stuckey PJ, de la Banda MG, Hermenegildo M, Wirth A. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing* 2010; **70**(8):839–848.
16. Rahwan T, Jennings NR. Coalition structure generation: Dynamic programming meets anytime optimization. *AAAI*, vol. 8, 2008; 156–161.
17. Loughry J, van Hemert J, Schoofs L. Efficiently enumerating the subsets of a set. <http://www.applied-math.org/subset.pdf> 2000. URL <http://www.applied-math.org/subset.pdf>.
18. <https://github.com/CoalitionStructureGeneration/DPIDP>. URL <https://github.com/CoalitionStructureGeneration/DPIDP>.
19. Nicol DM, Ciardo G. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing* 1997; **47**(2):153–167.
20. Arndt J. *Matters Computational: Ideas, Algorithms, Source Code*. 1st edn., Springer, 2011. URL <http://www.jjj.de/fxt/fxtbook.pdf>.
21. Ruskey F, Williams A. Generating combinations by prefix shifts. *Computing and Combinatorics*. Springer, 2005; 570–576.
22. Eades P, McKay B. An algorithm for generating subsets of fixed size with a strong. *Minimal Change Property*, *Information Processing Letters*, 1984; 131–133.