Optimizing Performance for Coalition Structure Generation Problems' IDP Algorithm

Francisco Cruz-Mencía
Computer Architecture Department.
Universitat Autònoma de Barcelona. Spain
Institut d'Investigació en
Intel·ligència Artificial
IIIA-CSIC. Spain

Jesús Cerquides
Institut d'Investigació en
Intel·ligència Artificial
IIIA-CSIC. Spain

Antonio Espinosa
Computer Architecture Department.
Universitat Autònoma de Barcelona.
Spain

Juan Carlos Moure
Computer Architecture Department.
Universitat Autònoma de Barcelona.
Spain

Juan Antonio Rodriguez-Aguilar
Institut d'Investigació en
Intel·ligència Artificial
IIIA-CSIC. Spain

***Abstract*—The Coalition Structure Generation (CSG) problem is well-known in the area of Multi-Agent Systems. Its goal is establishing coalitions between agents while maximizing the global welfare. Between the existing different algorithms designed to solve the CSG problem, DP and IDP are the ones with smaller temporal complexity. After analyzing the performance of the DP and IDP algorithms, we identify which is the most frequent operation and propose an optimized method. Then, we analyze the memory access pattern and find that its irregular behavior represents a potential performance bottleneck. In addition, we study and implement a method for dividing the work in different threads. We show that selecting the best algorithmic options can improve performance by 10x or more. Furthermore, the execution in a dual-socket, six-core processor computer may increase performance by an additional 5x-6x.**

I. Introduction

In the multi-agent systems area, coalition formation is one of the central types of collaboration. It involves the creation of disjoint groups of autonomous agents that collaborate in order to satisfy their individual or collective goals. One of the major research challenges in the field is the search for an effective set of coalitions that maximises the global satisfaction [1] of the agents.

Coalition formation is applied to many actual-world problems such as distributed vehicle route planning [2], task allocation [1], and airport slots allocation [3]. More recently, it has been considered in the realm of social networks [4].

According to [2], the coalition formation process is divided into three activities. In this paper we focus on the first one, namely coalition structure generation (CSG). Notice that finding the optimal coalition structure is $\mathcal{NP}$-complete [2]. The search space handled by CSG is very large since the number of possible coalition structures grows exponentially with the number of agents.

Several algorithms in the literature tackle the CSG problem. In particular, we distinguish three approaches: (i) optimal algorithms based on dynamic programming (e.g. DP [5], IDP [6]), which offer guaranteed run-times over arbitrary coalition value distributions; their complexity is $\Theta(3^n)$, where n is the number of agents; (ii) optimal algorithms with anytime properties whose convergence time to a solution largely depends on the coalition value distribution, which present a complexity $\Theta(n^n)$; and (iii) heuristic approximate algorithms (e.g. [1]), which aim at computing solutions faster than optimal algorithms without offering quality guarantees. Unfortunately, as widely noticed in the literature, the computational costs of optimal algorithms are highly demanding even for a moderate number of agents.

Against this background, in this paper we propose to optimize the algorithms based on dynamic programing. The implementation can be used as a building block for heuristic algorithms as a means to explore complete subspaces in an effective way.

As proposed in D-IP [7], where a distributed anytime algorithm is presented, in this paper we present an algorithm able to exploit the power of distribution but using a different paradigm. Our proposal is building a IDP based algorithm able to run in a shared memory scenario, which is common in nowadays computers [8]. Using a shared memory paradigm simplifies the communication between computation nodes, since there is no need to send messages between them, but it requires a data dependence study, because of possible synchronization.

As far as we are concerned, no reference implementation neither of DP nor IDP algorithms has been published. When studying and evaluating different implementation alternatives, we have found, though, non-negligible issues on the algorithmic details that have a considerable impact on the overall performance. The contributions of this work can be summarized as:

- We analyze and evaluate fast methods for generating splittings, the most critical operation, establishing that a bad choice can degrade performance by $10x$ or more.

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

The 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 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-threaded implementations. The paper ends summarizing the conclusions and presenting future work in Section 6.

II. The Coalition Structure Generation (CSG) Problem

In this section we describe what a Coalition Structure Generation (CSG) problem is and how dynamic programming algorithms have addressed it to find an optimal solution. To do so, we will use the following terminology:

- **Agent** (a_x): A single agent. E.g. Ann or Bob.
- **Agent Set** (A): The set of all available agents. $A = \{a_1, a_2, \ldots, a_n\}$.
  E.g. A= {Ann, Bob, Chris, Dave}.
- **Coalition** (C): $C \subseteq A$. C is a subset of A that contains the agents participating in a coalition.
  E.g. C= {Ann, Chris, Dave}.
- **Split** : Is the operation performing a binary partition of a coalition.
  E.g. {Ann,Chris,Dave} $\rightarrow$ ({Ann},{Chris,Dave}).
- **Splitting** : Is the result of the split operation. A splitting is a 2-tuple represented by (C_1, C_2). C=$C_1 \cup C_2$ where $|C_1|$,$|C_2|$ >0, $C_1 \cap C_2$=$\emptyset$.
  E.g. ({Ann},{Chris,Dave}) or ({Ann,Chris},{Dave}).
- **Coalition Structure** (CS): Is a collection of disjoint Coalitions such that their union constitute the Agent Set.
  E.g. ({Ann},{Bob},{Chris,Dave}).

Consider a group of n agents A=$\{a_1$,a_2,..., $a_n\}$. Agents can establish coalitions with other agents in order to perform a task. Each agent has its own preferences, meaning that some coalitions are preferred. These preferences are expressed by a value assigned to each possible coalition, denoted $value[C]$. It can be predefined or can be computed by every agent on the basis of its view of the world. In any case, coalition values are inputs known before solving the CSG problem. They can be represented by a table of size 2^n, one per coalition. Table I shows an example of the input data for a CSG Problem of size 4.

The goal of the CSG problem is to find the coalition structure providing maximum global satisfaction. From Table I one can notice that the coalition formed by $\{a_2$,$a_3\}$ has lower value than the sum of $value[\{a_2\}]$ and $value[\{a_3\}]$, meaning that agents a_2 and a_3 prefer to work alone rather than collaborate.

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 of size 4.

A. DP Algorithm

The DP[5] algorithm uses Dynamic Programming to find the optimal solution of the problem. For a given input data, DP first evaluates all the possible coalitions of size 2. For each possible pair of agents a_x and a_y, DP evaluates if it is better to form a coalition or not. This is done by comparing $value[\{a_x, a_y\}]$ with $value[\{a_x\}]$+ $value[\{a_y\}]$. The maximum value represents the preferred formation and substitutes the previous $value[\{a_x,\ a_y\}]$.

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

Algorithm 1 Pseudo-code of the DP Algorithm

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

The DP algorithm (see Algorithm 1) is composed of three nested loops: (i) the outer loop (line 1), where coalition size

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

B. IDP Algorithm

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

Algorithm 2 Pseudo-code of the IDP Algorithm

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

III. Single-Thread Implementation

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

A. Data representation

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

B. Splitting generation

The splitting generation problem can be reduced to the subset enumeration problem, since each coalition splitting is composed by a subset, C_1, and its complementary, C_2. Generating all the subsets C_1 from a coalition C and then calculating the complementary $C_2 = C - C_1$, though, would produce the same splitting twice: once for each of the splitting subsets.

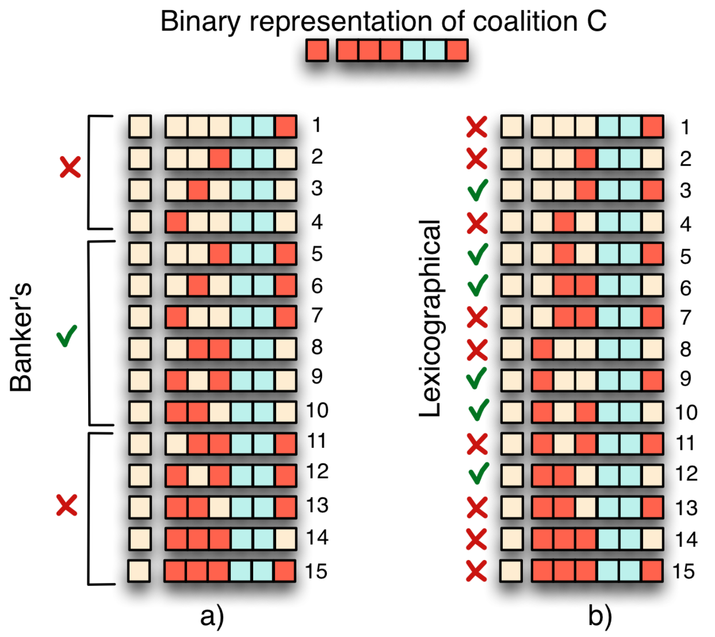


Fig. 1: a) Banker's sequence versus b) lexicographical order.

We remove one element from the coalition (the agent with the highest rank) when performing the subset enumeration, so that the removed element is never part of the enumerated subset and always belongs to its complementary.

There exist several ways of enumerating subsets [9], like banker's sequence, lexicographical order, and gray codes. The banker's sequence seems a suitable option for IDP, since it generates the splittings in growing order of $|C_1|$, and then simplifies the filtering of splittings by its size. Figure 1a shows a scheme of the banker's sequence operation for C=$\{a_1$,a_4,a_5,a_6,$a_7\}$, and assuming that only coalitions with $|C_1|$=2 need to be evaluated. Note that element a_7 is always assigned to the complementary subset (lighted colour). The generation starts directly from the first splitting of size $|C_1| = 2$, follows with the remaining $\binom{4}{2} - 1$ subsets of the same size, and stops before generating the first subset of size 3. The code does not waste instructions generating useless subsets.

When generating splittings in lexicographical order (see Fig. 1b), some filtering code is required to check that the size of the splitting ranges between a given pair of bounds. Execution resources are wasted to generate splittings that are then discarded, and to perform the filter check. In Fig. 1, only 6 out of 14 splittings are actually needed (note the check and discard crossed signs).

Both methods were implemented using recurrent functions that calculate the next splitting from the previous one. The lexicographical order was implemented with a few number of very simple operations: $C_1 \leftarrow (C_1 + C^{**})$ AND C, where C^{**} is the two's complement of C, that can be precalculated for all the splittings of a given coalition. The whole splitting code requires only 7 machine code instructions in a current x86 ISA. On the other hand, our implementation of banker's sequence, an improved version of the algorithm published in [9], required, on average, 6 times more instructions. More details about the implementation, like the usage of a special population count instruction for computing $|C_1|$, can be found in the published code.

C. Memory accesses

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

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

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

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

IV. Multi-thread Implementation

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

A. Identifying sources of TLP

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

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

B. Speeding up Work distribution among threads

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

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

TABLE II: Coalitions generated using lexicographical order.

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

C. Potential Parallel Performance Hazards

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

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

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

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

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

V. Experimental Results

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

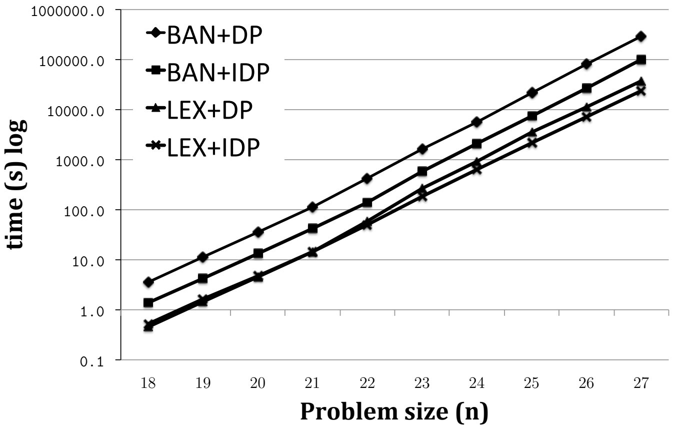
Input data was created using a uniform distribution as described by [10] for problem sizes $n = 18 \ldots 27$.

A. Single-thread Execution

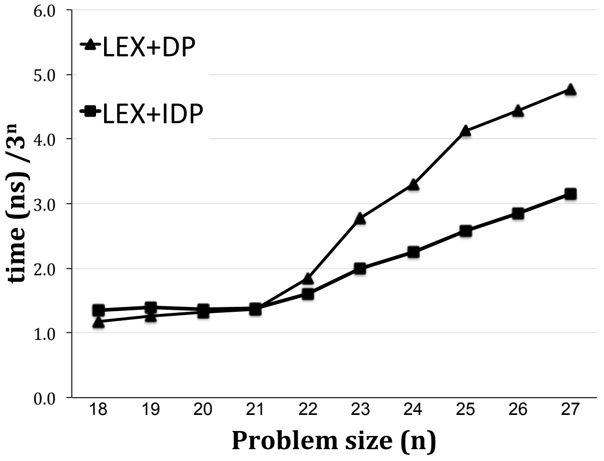
DP and IDP were executed using both the banker's and lexicographical splitting generation methods. Figure 2a plots the execution time in logarithmic scale for the four algorithmic variants. Lexicographic order is around $7x$ to $11x$ faster than banker's and, therefore, in the remaining of the paper we will use the first splitting method.

Figure 2b 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 2b shows two different problem size regions: those that fit into the LLC (n<22), and those that do not. A small problem size determines a computation-bound scenario, where DP slightly outperforms IDP, even when it executes around 20% more instructions. The reason is that IDP is penalized by a moderate number of branch mispredictions.

Large problem sizes determine a memory-bound scenario, where IDP amortizes its effort on saving expensive memory



(a) Execution time (log).

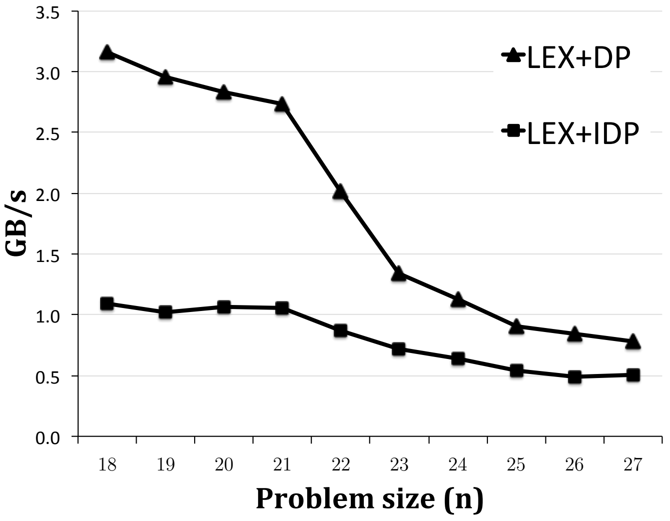


(b) Time / Complexity $\Theta(3^n)$.

accesses to outperform DP by 40-50%. Figure 2c shows the effective memory bandwidth consumption seen by the programs. The shape of the curves can be deduced from Figure 2b, but we are interested on the actual values. The effective bandwith ranges between 0.5 and 1.0 GB/sec. A small fraction of this bandwidth comes from the LLC and lower-level caches, and the remaining fraction comes from DRAM. Even considering the worst case described in section 3.3, that only 4 bytes out of the 64-Byte cache block are effectively used, it is still a very small value compared to the peak 32 GB/sec. The conclusion is that DRAM latency is the primary performance limiter. Results on the next subsection corroborate this conclusion.

B. Multi-thread Execution

We focus our multi-thread analysis on IDP, which outperforms DP for interesting problem sizes. We run IDP using t= 6, 12, and 24 threads. The case t=6 corresponds with using a single processor socket. The case t=12 uses only one socket but also exploits its hyperthreading capability. Finally, t= 24 is an scenario where all 2 sockets have their 6 cores running 2 threads each, using hyperthreading. Figure 3 shows



(c) Effective Memory Bandwidth (GB/s).

Fig. 2: Experimental data (BAN: Banker's sequence; LEX: Lexicographical order).

the speedup compared to the single-thread execution. Again, distinguishing between small and big problem sizes is useful.

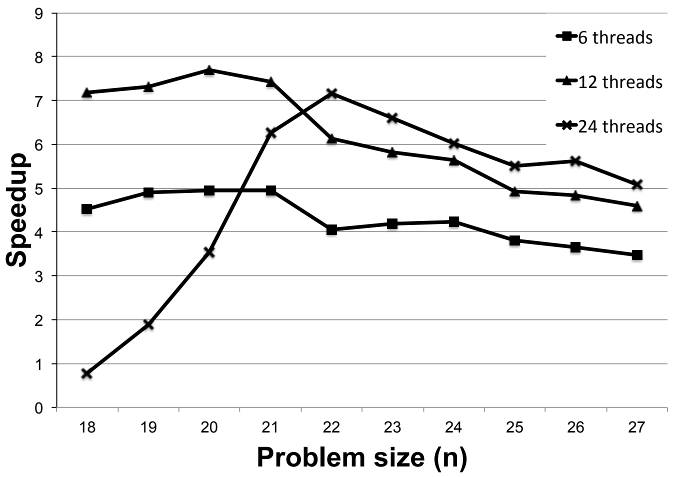


Fig. 3: Single-thread IDP versus 6-, 12- and 24-thread IDP execution

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

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

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

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

VI. Conclusions and future work

This paper presents an optimized implementation of the DP and IDP algorithm and a novel contribution describing the first parallel version of DP and IDP.

Our implementations clearly outperform the results found in the literature. According to [11], they need 2.5 days to solve a CSG problem with 27 agents, in some unspecified computer, and using a code implementation that is not provided. Our best single-thread implementation solves a same sized CSG problem in 5.8 hours. The multi-core implementation reduces execution time to 1.2 hours. Therefore, we claim that our implementation is the fast implementation of IDP published so far. We have made available to the community our source code.

We have analyzed the bottlenecks of DP and IDP. The pseudo-random memory access pattern lacks locality, and exploits the memory system capabilities very inefficiently. The latency tolerance ability of multi-threading improves performance on a multi-core processor. However, a dual-socket NUMA system is not appropriate for solving neither small nor big problems. The use of GPUs or accelerators with massive thread parallelism will be analyzed in the future.

We also want to study alternatives for coalition indexing and storage that provide higher locality, even at the expense of increasing instruction count, which is not a performance limiter for large problems.

Acknowledgment

This research has been supported by MICINN-Spain under contracts TIN2011-28689-C02-01, TIN2012-38876-C02-01 and the Generalitat of Catalunya (2009-SGR-1434).

References

[1] O. Shehory and S. Kraus, "Methods for task allocation via agent coalition formation," *Artif. Intell.*, vol. 101, no. 1-2, pp. 165–200, 1998.

[2] T. W. Sandholm and V. R. Lesser, "Coalitions among computationally bounded agents," *Artificial Intelligence*, vol. 94, pp. 99–137, 1997.

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

[4] T. Voice, S. D. Ramchurn, and N. R. Jennings, "On coalition formation with sparse synergies," in *AAMAS*, 2012, pp. 223–230.

[5] D. Yun Yeh, "A dynamic programming approach to the complete set partitioning problem," *BIT Numerical Mathematics*, vol. 26, pp. 467–474, 1986, 10.1007/BF01935053. [Online]. Available: http://dx.doi.org/10.1007/BF01935053

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

[7] T. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N. 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*. International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 1007–1014.

[8] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[9] J. Loughry, J. van Hemert, and L. Schoofs, "Efficiently enumerating the subsets of a set," http://www.applied-math.org/subset.pdf, 2000. [Online]. Available: http://www.applied-math.org/subset.pdf

[10] K. S. Larson and T. W. Sandholm, "Anytime coalition structure generation: an average case study," *J. of Experimental & Theoretical Artificial Intelligence*, vol. 12, no. 1, pp. 23–42, 2000. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/095281300146290

[11] T. Rahwan, S. D. Ramchurn, V. D. Dang, A. Giovannucci, and N. R. Jennings, "Anytime optimal coalition structure generation," in *AAAI*, 2007, pp. 1184–1190.