Coalition Structure Generation with the Graphics Processing Unit

Krzysztof Pawłowski*
University of Warsaw
kpidpg@gmail.com

Karol Kurach†
University of Warsaw
kkidpg@gmail.com

Kim Svensson
University of Southampton
ks6g10@ecs.soton.ac.uk

Sarvapali Ramchurn
University of Southampton
sdr@ecs.soton.ac.uk

Tomasz P. Michalak
University of Oxford
University of Warsaw
tomasz.michalak@cs.ox.ac.uk

Talal Rahwan
Masdar Institute
trahwan@gmail.com

ABSTRACT

Coalition Structure Generation—the problem of finding the optimal division of agents into coalitions—has received considerable attention in recent AI literature. The fastest exact algorithm to solve this problem is IDP-IP* [17], which is a hybrid of two previous algorithms, namely IDP and IP. Given this, it is desirable to speed up IDP as this will, in turn, improve upon the state-of-the-art. In this paper, we present IDP^G—the first coalition structure generation algorithm based on the *Graphics Processing Unit (GPU)*. This follows a promising, new algorithm design paradigm that can provide significant speedups. We show that IDP^G is faster than IDP by two orders of magnitude.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General

General Terms

Theory, Design, Performance

Keywords

Coalition Structure Generation, Dynamic Programming, GPU

1. INTRODUCTION

Coalitional games have been studied in various areas of artificial intelligence and multi-agent systems [4, 14, 18]. By cooperating, agents are often able to enhance their performance and achieve tasks otherwise unachievable. The formation of coalitions is relevant both in cases where agents are *cooperative* (*i.e.*, they maximize the social welfare) as well as cases where they are *selfish* (*i.e.*, each agent maximizes its own reward, regardless of the consequences on others). Coalition formation techniques can be used, for example, to improve the surveillance of an area using autonomous sensors [9], or reduce the uncertainty that green-energy generators have about their own production [5], or allow buyers to obtain cheaper prices through bulk purchasing [10].

In general, the effectiveness of a coalition can be influenced by other co-existing coalitions. Such settings are known as *partition function games* [11]. On the other hand, in *characteristic function games* (CFGs), a coalition's effectiveness depends solely on the identities of its members. This assumption simplifies the research questions significantly, and holds in many realistic settings [5, 9, 24, 22]. Thus, as common practice in the literature, we focus in this paper on characteristic function games (CFGs).

Generally speaking, there are settings where merging any two coalitions is always beneficial. In such settings, all the agents should work together in one big coalition. There are other settings, however, where there are coordination and/or communication costs that often increase with the size of the coalition. In such settings, the agents may find it more profitable to partition themselves into multiple, disjoint coalitions. Such a partition is called a coalition structure, and the problem of identifying the best such partition is known as the *coalition structure generation* problem. Many algorithms have recently been developed to solve this problem, some of which use the classical representation of CFGs, e.g., [1, 13, 19], while others are tailored for certain classes or alternative representations of CFGs, e.g., [4, 25, 18, 3, 26]. We focus on the former type of algorithms and, in particular, those that are *exact*. In this context, the state-of-the-art algorithm is IDP-IP* [17]. As the name suggests, this algorithm is a hybrid of two previous algorithms, namely IDP [16] and IP [20]. While IP, given n agents, runs in $O(n^n)$ time, combining it with IDP reduces the complexity to $O(3^n)$. Furthermore, the hybrid has been tested against several problem instances, and has been shown to be faster in practice than both its constituent parts. Based on all the above, any improvements to IDP will naturally result in improvements to the state-of-the-art.

Against this background, in this paper we set to develop a faster version of IDP following a promising, new algorithm design paradigm that builds upon *Graphics Processing Units (GPUs)*. In more detail, a GPU is a piece of hardware (consisting of multiple computational entities, or "cores", just like a standard CPU) designed mainly for rendering 3D computer graphics. It differs from a standard CPU, though, in that the number of "cores" is larger, while the speed per core is lower. Importantly, however, the total computational power on a GPU (when taking all cores into consideration) is much greater than that of a CPU. This, many have realized, meant that almost any algorithm can run faster on a GPU than a CPU, provided that this algorithm can be efficiently parallelized. Such capabilities gave rise to a new line of research known as GPGPU

*First author together with Karol Kurach.

†First author together with Krzysztof Pawłowski.

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

(General Purpose computing on the GPU), which studies the science of overcoming the challenges imposed by the transition from traditional CPUs to GPUs. To date, this remains a green field, with many areas yet to be explored, and many widely-used algorithms waiting to be redesigned for GPUs. This line of research spreads over different fields such as artificial intelligence [6, 15], computational biology [8], linear algebra [23], signal processing [28], among others. Speedups of one or even two orders of magnitude are widely reported. Another appealing feature for using GPUs is that its advantage (in terms of total computational power) over CPUs has been growing in recent years [2]. While high-performance alternatives to GPU exist, such as super-computing clusters and field-programmable gate arrays (FPGAs), these are much more expensive at similar performance levels compared to GPUs.

We bring to the attention of the Computational Coalition Formation community some of the desiderata that can significantly enhance the performance when developing GPU algorithms:[1]

- **Minimize the number of synchronization points**, i.e., the points in the algorithm to which all cores must arrive before the algorithm can proceed. The problem with synchronization points is that they cause delays. This happens when some cores arrive to a synchronization point before others, which means they have to remain idle while waiting for the others. This is inevitable in practice even when the cores have identical computational power, especially if the number of cores is in the hundreds as is the case with GPU (since one core can cause all others to wait).

- **Minimize the number of global memory accesses**. In a nutshell, a GPU contains a centralized *global memory* that can be shared by different threads. It is by far the largest on a GPU, meaning it is the only option when storing large amounts of data (e.g., the characteristic function table with 2^n values). However, global memory is slow. Thus, when designing GPU algorithms, one should try and keep global memory accesses to a minimum.

- **Maximize the number of threads** that are scheduled at the same time. At first glance, when breaking the optimization problem into smaller subproblems that are each solved by a separate thread, it may seem ideal to have as many threads as there are cores on the GPU. However, this could lead to inefficiency due to *memory latency*. In particular, whenever a thread needs to access global memory, the core that is executing that thread would remain idle until memory access is granted. This is precisely why it is more efficient to have as many threads on a single core as possible; it allows the core to switch to a thread while another is waiting to be granted memory access. This optimization is known as *latency hiding*.

- **Minimize the number of instructions per "branch"**. This is due to the way GPUs operate. Basically, cores on a GPU are divided into groups; one group on every *"streaming multiprocessor"*. Whenever cores in the same groups happen to be executing the same line of code simultaneously, the GPU can take advantage of this and speed up the execution.[2] Based on this, when designing GPU-based algorithms, one should try and have as few instructions as possible in *"branches"*, which in the GPU context mean blocks of code, execution of which depends on conditional instruction such as the "if" instruction. This optimization is known as reducing *warp divergence*.

Against this background, our contributions in this paper can be summarized as follows:

- We develop IDP^G—the reformulation of IDP for GPUs. This is the first GPU-based algorithm for coalition structure generation.

- We prove that the number of synchronization points in IDP^G is optimal, i.e., it is not possible to parallelize IDP with a fewer number of such points. We then prove a certain property of IDP, and exploit it when reducing global memory access in IDP^G.

- We evaluate our algorithm experimentally, and show that it outperforms IDP by two orders of magnitude.

The remainder of the paper is organized as follows. Section 2 provides the necessary background. Section 3 presents IDP^G. Section 4 presents empirical evaluations. Section 5 reviews the related work. Finally, Section 6 concludes the paper and outlines future directions.

2. BACKGROUND

In this section we formally state the coalition structure generation problem, and then describe IDP—which stands for Improved Dynamic Programming—since we will build on it later on in the paper.

Let $A = \{a_1, \ldots, a_n\}$ denote the set of n agents. We focus on characteristic function games, where the efficiency of any coalition, $C \subseteq A$, is represented by a real number, known as *the value of* C, and denoted by $v(C)$. Formally, $v : 2^A \rightarrow \mathbb{R}$. Now, let Π^A be the set of possible coalition structures (i.e., partitions over A) and, for any coalition structure CS, let $V(CS)$ denote the *value* of CS, where: $V(CS) = \sum_{C \in CS} v(C)$. Furthermore, let CS^* denote an *optimal* coalition structure. That is, $CS^* \in \arg\max_{CS \in \Pi^A} V(CS)$. The coalition structure generation problem is then the problem of finding one such CS^*.

Now, we turn our attention to IDP. To help the reader understand how it works, we need to first explain a preliminary version of this algorithm, called DP [27]—which stands for Dynamic Programming; this is basically the first dynamic programming algorithm proposed to solve the coalition structure generation problem. Before we can explain how the DP algorithm works, we need to introduce some additional definitions. More specifically, we will refer to any set of disjoint coalitions as a *"partition"*, denoted P. Only when such a partition contains all agents will we use the term *"coalition structure"*. Now, for any coalition $C \subseteq A$, let Π^C be the set of possible *partitions of* C, where a partition $P = \{P_1, \cdots, P_{|P|}\} \in \Pi^C$ is a set of disjoint coalitions of which the union equals C. In the same way that we defined the value of a coalition structure, we now define the *value of a partition*. Formally, let $V(P) = \sum_{P_i \in P} v(P_i)$ be the value of partition P. Now, let $f(C)$ be the value of the optimal partition of C, i.e., $f(C) = \max_{P \in \Pi^C} V(P)$. Then, DP is based on the following

[1] We will be using some standard terms commonly used in research on GPU, and in the documentations of NVIDIA—the world's leading GPU developer: `http://docs.nvidia.com`

[2] This is due to hardware specifications of GPUs. Additional information can be found in NVIDIA's *"CUDA C Best Practice Guide"*, `http://docs.nvidia.com`

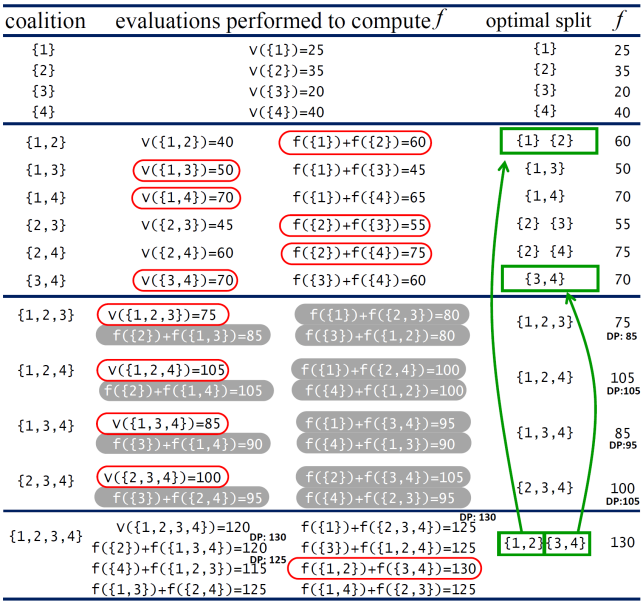

Figure 1: IDP vs. DP, given $A = \{a_1, a_2, a_3, a_4\}$ and function v as shown in Example 1. Highlighted in green is the path leading to the optimal result. Locally optimal results are highlighted in red. Gray indicates splits that are considered by DP, but not IDP.

recursive formula:

$$f(C) = \begin{cases} v(C) & \text{if } |C| = 1 \\ \max\left\{v(C),\ \max_{\{C',C''\}\in\Pi^C}\left(f(C') + f(C'')\right)\right\} & \text{otherwise.} \end{cases}$$

In other words, for any coalition C, if we know the optimal partition of every strict subset of C, then we can easily (relatively speaking) find an optimal partition of C: Instead of examining all partitions in P^C, it suffices to examine only those containing exactly two coalitions (i.e., examine every $\{C', C''\} \in \Pi^C$ as in the equation above) and then find one that maximizes $V(C') + V(C'')$, denoted $\{C^*, C^{**}\}$. Once we have identified $\{C^*, C^{**}\}$, the optimal partition of C can be found straight away; it is the union of the optimal partitions of C^* and C^{**}, unless $v(C) > f(C^*) + f(C^{**})$, in which case it is $\{C\}$.

Based on the above idea, the DP algorithm iterates over all the coalitions of size 1, and then over all those of size 2, and then size 3, and so on until size n. For every such coalition C, it computes $f(C)$ using the above equation. So, to summarize, let us call every partition containing exactly two coalitions a *"split"*. Then, DP works by evaluating every possible split of every possible coalition, starting with the coalitions of size 2, then moving to those of size 3, and so on until size n.

Example 1. *Let $A = \{a_1, a_2, a_3, a_4\}$ and $v(\{a_1\}) = 25$, $v(\{a_2\}) = 35$, $v(\{a_3\}) = 20$, $v(\{a_4\}) = 40$, $v(\{a_1, a_2\}) = 40$, $v(\{a_1, a_3\}) = 50$, $v(\{a_1, a_4\}) = 70$, $v(\{a_2, a_3\}) = 45$, $v(\{a_2, a_4\}) = 60$, $v(\{a_3, a_4\}) = 70$, $v(\{a_1, a_2, a_3\}) = 75$. $v(\{a_1, a_2, a_4\}) = 105$, $v(\{a_1, a_3, a_4\}) = 85$, $v(\{a_2, a_3, a_4\}) = 100$, $v(\{a_1, a_2, a_3, a_4\}) = 120$. The splits computed by DP to generate f are shown in Figure 1.*

Let us now move back to IDP, the improved version of DP [16]. In particular, the authors showed that certain splits can safely be skipped without losing the guarantee of finding an optimal coalition structure. In particular, they showed that it is sufficient to evaluate the splits that involve partitioning a coalition of size c into two coalitions of sizes c' and c'', where (c', c'') is in:

$$dep(c) = \Big\{(c', c'') \in \mathbb{N}^2 : (c' \geq c'') \ \wedge\ (c' + c'' = c) \\ \wedge\ [(c' \leq n - c' - c'') \vee\ (c = n)]\Big\}.$$

Based on this, for any given coalition C such that $|C| = c$, IDP only evaluates the splits in Π^C that involve partitioning C into two coalitions of sizes c' and c'', where $(c', c'') \in dep(c)$. We chose the notation dep as it indicates the *dependencies* between different coalition sizes. Rahwan and Jennings proved that $dep(c) = \emptyset$ for all $c \in \left\{\left\lfloor \frac{2n}{3} \right\rfloor + 1, \ldots, n-1\right\}$. However, the link between the value of c and the elements in $dep(c)$ was not formalized for cases where $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$.

Example 2. *Coming back to the example in Figure 1, IDP evaluates no splits of coalitions of size 3. In other words, it evaluates 12 splits less compared to DP.*

3. IDP^G

In this section, we present IDP^G—a parallelized version of IDP, designed to meet the desiderata outlined in the introduction. The open-source implementation of IDP^G is made publicly available [3].

To simplify notation, we will denote by c, c', c'' the cardinalities (i.e., sizes) of coalitions C, C', C'', respectively. Furthermore, the problem of evaluating every $\{C', C''\} \in \Pi^C : (c', c'') \in dep(c)$ for a given C will be called the *"subproblem of C"*, or simply a *"subproblem"* when there is no risk of confusion. The section is divided as follows.

- Section 3.1 focuses on minimizing the number of synchronization points. In particular, we show how IDP's operations can be parallelized with $\lceil n/2 \rceil - 1$ synchronization points. We then prove the correctness of the proposed synchronization scheme, and prove it is optimal, that is, it is not possible to parallelize the operation of IDP with a number of synchronization points smaller than $\lceil n/2 \rceil - 1$.
- Section 3.2 explains an efficient way for enumerating the subsets of size k out of a set of size n. This operation is used to assign a different subproblem to each GPU thread.
- Section 3.3 describes the pseudo codes of IDP^G.
- Section 3.4 focuses on the other desired properties that we set earlier in the introduction.

3.1 Handling Synchronization Points

In order to parallelize IDP, we need to analyze the dependencies between the different subproblems. Here, our aim is to group the subproblems into *"stages"* that are solved sequentially (i.e., all subproblems in stage 1 are solved first, then all of those in stage 2, then stage 3 and so on). We aim to do this in such a way that guarantees every subproblem is solved before any of its dependents, i.e., guarantees that every subproblem depends solely on subproblems belonging to earlier stages. This way, the dependencies between subproblems are reduced to dependencies between stages. With this, subproblems within the same stage can be computed in parallel without any need for synchronization. To be more precise, synchronization would be needed between stages, but not within a stage.

[3] For the IDP^G implementation developed part of this research, see: *https://github.com/idpg/idpg*

c	l(c)	dep(c)	ld(c)
1	0	$\emptyset$	$\emptyset$
2	1	$\{(1,1)\}$	$\{0\}$
3	2	$\{(2,1)\}$	$\{0,1\}$
4	3	$\{(2,2),(3,1)\}$	$\{0,1,2\}$
5	4	$\{(3,2),(4,1)\}$	$\{0,1,2,3\}$
6	4	$\{(3,3),(4,2)\}$	$\{1,2,3\}$
7	0	$\emptyset$	$\emptyset$
8	0	$\emptyset$	$\emptyset$
9	0	$\emptyset$	$\emptyset$
10	5	{(5, 5), (6, 4), (7, 3), (8, 2), (9, 1)}	$\{0,1,2,3,4\}$

Table 1: Stage assignments and dependencies for 10 agents

To this end, observe that the definition of $dep(c)$ implies the following:

$$(c', c'') \in dep(c) \textbf{ iff } \begin{cases} 1 \leq c' \leq n-1 \textbf{ and} \\ c' + c'' = c \textbf{ and} \\ (c \leq \lfloor \frac{2n}{3} \rfloor \textbf{ or } c = n) \textbf{ and} \\ (\lceil \frac{c}{2} \rceil \leq c' \leq n - c \textbf{ or } c = n). \end{cases} \quad (1)$$

Based on this, we will show how to group subproblems into stages so as to minimize the number of synchronization points. Specifically, in our algorithm, determining the stage of a given subproblem depends solely on the size of the coalition whose splits are evaluated in that subproblem. More formally, the subproblem of coalition $C : |C| = c$ is assigned to stage $l(c)$, which is defined as follows:

$$l(c) = \begin{cases} c-1 & \textbf{if } 1 \leq c \leq \lfloor \frac{n+1}{2} \rfloor \\ n-c & \textbf{if } \lceil \frac{n}{2} \rceil + 1 \leq c \leq \lfloor \frac{2n}{3} \rfloor \\ 0 & \textbf{if } \lceil \frac{2n+1}{3} \rceil \leq c \leq n-1 \\ \lceil \frac{n}{2} \rceil & \textbf{if } c = n. \end{cases} \quad (2)$$

Example 3. *Table 1 illustrates how the assignment of subproblems to stages works in practice for a problem with* 10 *agents. The first column, c, denotes the size of the subproblem. The second column, $l(c)$, describes the stage that the subproblem has been assigned to, according to Equation 2. Third column, $dep(c)$, is the set of dependencies as defined by Equation 1. Finally, the last column shows the value of an expression $ld(c) = \bigcup\{\{l(x), l(y)\} : (x, y) \in dep(c)\}$, which is simply a set of stage numbers that the subproblem of C depends on, where $|C| = c$. As can be seen, all $ld(c)$ elements are smaller than $l(c)$. This basically means we can order the execution according to $l(c)$ without violating any dependencies.*

To prove that the above assignment of subproblems to stages is correct, we need to prove that whenever a subproblem depends on another subproblem, the former will always be assigned to an earlier stage compared to the latter. In order to do so, it is sufficient to prove the following theorem:

THEOREM 1. *For every $c \in \{1, \ldots, n\}$, and every $(c', c'') \in dep(c)$, the following holds:*

$$\Big(l(c') < l(c)\Big) \textbf{ and } \Big(l(c'') < l(c)\Big) . \quad (3)$$

Proof. We will prove that (3) holds for each case presented in (2). In particular:

Case 1: $c = n$.
From (2) we know that: $l(c) = \lceil \frac{n}{2} \rceil$, and that $l(s) < \lceil \frac{n}{2} \rceil$ for all $s \neq n$. Thus, having $c' < c$ and $c'' < c$ imply that $l(c') < l(c)$ and that $l(c'') < l(c)$.

Case 2: $c \leq \lfloor \frac{2n}{3} \rfloor$ ***and*** $1 \leq c \leq \lfloor \frac{n+1}{2} \rfloor$.
From (2) we know that $l(c) = c - 1$. Now if $c = 1$, then of course $dep(c) = \emptyset$. On the other hand, if $c > 1$, then $1 \leq c'$ and $c'' < c \leq \frac{n+1}{2}$. Therefore, $l(c') = c' - 1$ and $l(c'') = c'' - 1$. Trivially then, $l(c') < l(c)$ and $l(c'') < l(c)$.

Case 3: $\lceil \frac{n}{2} \rceil + 1 \leq c \leq \lfloor \frac{2n}{3} \rfloor$.
First, let us deal with c', i.e., let us prove that $l(c') < l(c)$. To this end, from (2) we have $l(c) = n - c$. Moreover, (1) yields $c' \leq n - c$ (since obviously $c' \neq n$). Also, since $\lceil \frac{n}{2} \rceil + 1 \leq c$, we have $c' \leq n - (\lceil \frac{n}{2} \rceil + 1) = \lfloor \frac{n}{2} \rfloor - 1 = \lceil \frac{n-2}{2} \rceil \leq \lfloor \frac{n+1}{2} \rfloor$. Based on this, (2) implies that $l(c') = c' - 1$. Moving on, $c' \leq n - c$ yields $l(c') \leq n - c - 1$ and therefore $l(c') < n - c$. In effect, we have shown that $l(c') < l(c)$.

Now, let us deal with c'', i.e., let us prove that $l(c'') < l(c)$. To this end, we know from (1) that $\lceil \frac{c}{2} \rceil \leq c'$. Based on this, $-c' \leq -\lceil \frac{c}{2} \rceil \Rightarrow c - c' \leq c - \lceil \frac{c}{2} \rceil$. Now since $c' + c'' = c$, we have $c'' \leq \lfloor \frac{c}{2} \rfloor \leq \lfloor \frac{n}{2} \rfloor \leq \lfloor \frac{n+1}{2} \rfloor$. This, as well as (2) imply that $l(c'') = c'' - 1 \leq \lfloor \frac{c}{2} \rfloor - 1 < \frac{c}{2}$. From the assumption that $c \leq \lfloor \frac{2n}{3} \rfloor$ we get that $l(c'') < \frac{\lfloor \frac{2n}{3} \rfloor}{2} \leq \frac{n}{3}$. This assumption also yields the following: $-\lfloor \frac{2n}{3} \rfloor \leq -c \Rightarrow n - \lfloor \frac{2n}{3} \rfloor \leq n - c \Rightarrow \lceil \frac{n}{3} \rceil \leq l(c)$. Thus $l(c'') < \frac{n}{3}$ and $\lceil \frac{n}{3} \rceil \leq l(c)$, which is what we wanted to prove.

□

Our synchronization scheme divides the computation into $\lceil \frac{n}{2} \rceil + 1$ stages. However, the first stage contains subproblems for coalitions of size 1. This means the algorithm starts from stage 2. Based on this, our synchronization scheme requires exactly $\lceil \frac{n}{2} \rceil - 1$ synchronization points. Next, we prove that this synchronization scheme is optimal, i.e., IDP cannot be parallelized with a number synchronization points smaller than $\lceil \frac{n}{2} \rceil - 1$.

THEOREM 2. *There exists no parallelization scheme that requires less than $\lceil \frac{n}{2} \rceil - 1$ synchronization points.*

Proof. Let $S = 2^A$ be the set of coalitions (subsets) of a set of agents A. Let $Dep : S \longrightarrow 2^{S \times S}$ be a function that maps a subproblem (coalition) into a set of its dependants. That is, let: $(C', C'') \in Dep(C)$ **iff** IDP evaluates a split of C into (C', C''). Furthermore, let $\rightsquigarrow$ be a binary relation on $P(A)$ such that: $C \rightsquigarrow D$ **iff** $(C, D \setminus C) \in Dep(D)$.

To prove that at least k synchronization points are necessary, it is sufficient to construct a $(k+1)$-element sequence $C_0, C_1, ..., C_k$ such that

$$\forall_{i=\{1,\ldots,k\}} C_{i-1} \rightsquigarrow C_i \textbf{ and } \text{computation for } C_{i-1} \text{ is needed.} \quad (4)$$

We construct such a sequence with $\lceil \frac{n}{2} \rceil$ elements:

$$\{a_1, a_2\} \rightsquigarrow \{a_1, a_2, a_3\} \rightsquigarrow ... \rightsquigarrow \{a_1, ..., a_{\lceil \frac{n}{2} \rceil}\} \rightsquigarrow A .$$

Condition (4) is met because we have:

$$\forall_{i=\{1,\ldots,k\}} (C_{i-1}, C_i \setminus C_{i-1}) \in Dep(C_i) ,$$

which follows directly from the definition of dep in (1). This concludes the proof that there exists a $(k+1)$-element sequence of coalitions that satisfies (4). Every element of the sequence requires a computation and, since it depends on the previous element in the

Algorithm 1: HOST CODE THAT MANAGES THE GPU

Input: f, n
Output: f
1 Copy f from host to the device ;
2 **for** $s \leftarrow 2$ ***to*** n **do**
3 **if** $s > \lceil n/2 \rceil$ *and* $s \neq n$ **then**
4 **continue**;
5 *// Call the GPU code, see Algorithm 2*
6 spawn $\binom{n}{s}$ threads on GPU with parameters (f, n, s)
7 $s' \leftarrow n - s + 1$;
8 **if** $\neg(2n < 3s'$ *and* $s' < n)$ *and* $s \neq n$ **then**
9 *// Solve subproblems of size s' in parallel.*
10 spawn $\binom{n}{s'}$ extra threads with parameters (f, n, s')
11 wait for all threads to finish
12 Copy f from the device to host;

Algorithm 2: CODE THAT IS RUN ON THE GPU

Input: f, n, s
Output: f
1 *// Every thread has unique index* $\in \{1, \ldots, \binom{n}{s}\}$
2 id $\leftarrow$ compute index of current thread;
3 **if** $id \leq \binom{n}{s}$ **then**
4 $C \leftarrow g(n, k, id)$;
5 *// For CheckSplit definition, see Algorithms 3 and 4*
6 $\mathfrak{C} = \{C' | C' \subset C$ and CheckSplit$(n, C', C)\}$
7 $x \leftarrow \max\{f[C'] + f[C \setminus C'] : C' \in \mathfrak{C}\}$;
8 $f[C] \leftarrow \max(f[C], x)$;

sequence, it needs to be in a separate stage. Because of that it is not possible to design an algorithm with fewer than $\lceil \frac{n}{2} \rceil - 1$ synchronization points. Thus, IDP^G is optimal in the number of synchronization points.

□

3.2 Computing the i^{th} subset of size k

In order to support parallel execution, we need to adopt an ordering of the subsets of a given size k out of the set of n agents. As we will show later on, such an ordering allows us to distribute all coalitions of a certain size among the different threads. We propose an ordering that we formally define as follows:

Definition 1. *Our ordering of k-element subsets, out of a set of n elements, is:*
$X \prec Y$ ***iff*** $\exists_i \forall_{j>i} a_i \notin X$ ***and*** $a_i \in Y$ ***and*** $(a_j \in X$ ***iff*** $a_j \in Y)$.

Now, for any given size $k \in \{1, \ldots, n\}$, we need an efficient method to compute the i^{th} subset according to our ordering. For this purpose, we define the following function (which will be evaluated by every thread to find a subset for which the thread is responsible):

$$g(n,k,i) = \begin{cases} \emptyset & n < 0 \vee k = 0 \\ \{a_n\} \cup g(n-1, k-1, i - \binom{n-1}{k}) & k > 0 \wedge i \geq \binom{n-1}{k} \\ g(n-1,k,i) & k > 0 \wedge i < \binom{n-1}{k} \end{cases}$$

Example 4. *All 2-element subsets of 4 agents in order $\prec$ are:* $\{a_1, a_2\} \prec \{a_1, a_3\} \prec \{a_2, a_3\} \prec \{a_1, a_4\} \prec \{a_2, a_4\} \prec \{a_3, a_4\}$. *The first subset calculated by function g (i.e., the one at position $i = 0$) is:* $g(4,2,0) = g(3,2,0) = g(2,2,0) = \{a_2\} \cup g(1,1,0) = \{a_1, a_2\} \cup g(0,0,0) = \{a_1, a_2\}$. *Similarly, the fourth subset in relation $\prec$ (i.e., the one at position $i = 3$) is calculated as follows:* $g(4,2,3) = \{a_4\} \cup g(3,1,0) = \{a_4\} \cup g(2,1,0) = \{a_4\} \cup g(1,1,0) = \{a_1, a_4\} \cup g(0,0,0) = \{a_1, a_4\}$.

THEOREM 3. *The function $g(n,k,i)$, defined for $k \leq n$ and $0 \leq i < \binom{n}{k}$, returns the i-th k-element subset (with relation $\prec$, 0-based) from a set of n agents.*

Proof. We use structural induction.
Basis: $g(0,0,0) = \emptyset$, since there is only one 0-element subset of 0 elements, which is the empty set.
Inductive step: Let us assume that $\forall_{l \leq n-1, j < \binom{n-1}{l}} g(n-1, l, j)$ meets Theorem 3 , which can be written as $\forall_{j' > j} g(n-1,l,j) \prec g(n-1,l,j')$. We will show that $\forall_{k \leq n, i < \binom{n}{k}} g(n,k,i)$ also meets Theorem 3 .

Let us consider the result of $g(n,k,i)$. If $k = 0$, then we know that no more agents can be added, so the only possible result is $\emptyset$, which is returned by g. When $k > 0$ we can either add a_n (the last agent) or not.

Case *1: $i < \binom{n-1}{k}$.*
If $i < \binom{n-1}{k}$, we skip agent a_n because there are $\binom{n-1}{k}$ subsets of the remaining $n-1$ agents that do not have a_n. All of them are smaller (with relation $\prec$) than any subset that contains a_n. We return the result of $g(n-1,k,i)$, which is correct due to the induction hypothesis.

Case *2: $i \geq \binom{n-1}{k}$.*
Since we skipped all $\binom{n-1}{k}$ subsets that do not contain a_n, we add agent a_n to the result. Instead of looking at the k-element subset at position i, now we are looking for a $(k-1)$-element subset at position $i - \binom{n}{k}$ with the remaining $n-1$ agents. This set is exactly $g(n-1, k-1, i - \binom{n-1}{k})$, which can be calculated using the induction hypothesis (since $i - \binom{n-1}{k} \leq \binom{n-1}{k-1}$). Thus, the final result is $\{a_n\} \cup g(n-1, k-1, i - \binom{n-1}{k})$.

Since a_n is in the result, all subsets returned for $i \geq \binom{n-1}{k}$ are greater in relation $\prec$ than subsets from case 1. Also, based on the induction hypothesis: $\forall_{i' > i} g(n-1, k-1, i - \binom{n}{k}) \prec g(n-1, k-1, i' - \binom{n}{k})$.

□

3.3 Pseudo code

The code executed on the host by the CPU is presented in Algorithm 1. First, in line 1, the CPU transfers data from the host to the device (GPU). Then, lines 2 to 11 involve a loop over all $\lceil n/2 \rceil$ stages. Specifically in this loop, line 3 ensures that no more stages than necessary are considered. In each stage, at least $\binom{n}{s}$ kernel threads spawn[4] on the device. This is done in line 6. Optionally, lines 7-10 spawn an extra $\binom{n}{s'}$ threads responsible for solving subproblems of size s', where $s' \geq \lceil n/2 \rceil$. All threads execute in parallel on the GPU, while the CPU remains idle at the synchronization point (line 11) until all threads complete. This concludes the processing of a single stage. Finally, after all stages have been processed, the newly-computed f is copied from the device back to the host (line 12).

Algorithm 2 presents the pseudo code of a single kernel that is executed on the GPU. First, the index of the thread is computed (line 2). At this stage, id represents the index of the current thread in the group of threads responsible for subproblems of size s. Due to rounding up, this may slightly be greater than $\binom{n}{s}$ (the total number of subproblems in the stage). If that is the case, line 3 ensures

[4]The term "spawning" is standard in parallel computing; it simply means creating and starting a new thread.

Algorithm 3: CHECKSPLIT CONDITION (SMALL)

Input: n, C', C where $C' \subset C$ and $|C| \leq \frac{n}{2}$
Output: True iff split $(C', C \setminus C')$ should be evaluated
1 **return** $|C'| \geq 1/2 * |C|$;

Algorithm 4: CHECKSPLIT CONDITION (LARGE)

Input: n, C', C where $C' \subset C$ and $|C| > \frac{n}{2}$
Output: True iff split $(C', C \setminus C')$ should be evaluated
1 **return** $|C'| \geq 1/2 * |C|$ ***and*** $(|C'| \leq n - |C|$ ***or*** $|C| = n)$;

that no further code will be executed. However, if a thread's id is in the proper range, execution continues from line 4. The algorithm generates a bitmask that represents the id-th k-element subset of an n-element set and stores it as C. See Section 3.2 for an explanation of how the id-th k-element subset is generated. In the next step, the algorithm finds the best split of C into two complementary subsets (line 7). If the split is worth more than $v(C)$, the entry $f[C]$ is updated (line 8). Subsets of C are enumerated efficiently using a constant number of arithmetic operations per split. *CheckSplit* conditions are tested (Algorithms 3 and 4) to determine if a given split should be evaluated. Those important procedures are described in more detail in the following section.

3.4 Handling Other Design Requirements

One of the innovations in IDP^G is how splits are evaluated. As stated in the desiderata, limiting global memory access is of utmost importance and translates to significant performance gains. In contrast, minimizing the number of executed instructions that do not access global memory is of less importance. To reflect this, while IDP^G calculates every possible split of every subproblem it analyzes, it only performs an expensive global memory lookup when absolutely necessary. In other words, coalition values stored in global memory are retrieved when it cannot be avoided.

To keep an overhead small, subproblems are enumerated by very fast bitmask operations (as described by [12]). Additionally, for each potential split, IDP^G strives to keep the execution cost as low as possible.

In general, IDP^G checks the logical conditions (similarly to IDP) to see if a given split should be considered. As it turns out, some of that checks can be avoided due to the following result.

THEOREM 4. *All splits of coalitions of size c, where $|c| \leq \frac{n}{2}$, need to be evaluated.*

Proof. Recall how Equation (1) specifies when a split is needed. IDP^G iterates over all the potential splits that meet the first three cases in the aforementioned equation. The only thing left to prove is that the condition $c' \leq n - c$ holds, thus making the split necessary. The assumptions in this case are:

- $c \leq \frac{n}{2}$ (Theorem 4 assumption) **and**
- $c' + c'' = c$ (since $C = C' \cup C''$) **and**
- $1 \leq c' \leq n - 1$ (follows from two conditions above) **and**
- $\lceil \frac{c}{2} \rceil \leq c'$ (because C' is never smaller than C'').

By transforming the above inequalities we get: $-\frac{n}{2} \leq -c$ **and** $c' \leq c \leq \frac{n}{2} = n - \frac{n}{2}$. Then, by substituting $-\frac{n}{2}$ we get the desired: $c' \leq n - c$. Thus, for coalitions of sizes at most $\frac{n}{2}$, the check can be safely avoided – we just proved that the condition is always met and therefore cannot lead to a reduced number of global memory accesses[5]. □

[5]Compare Algorithms 3 and 4. For larger coalition sizes an extra check is performed as only in that case it can prevent unnecessary global memory accesses.

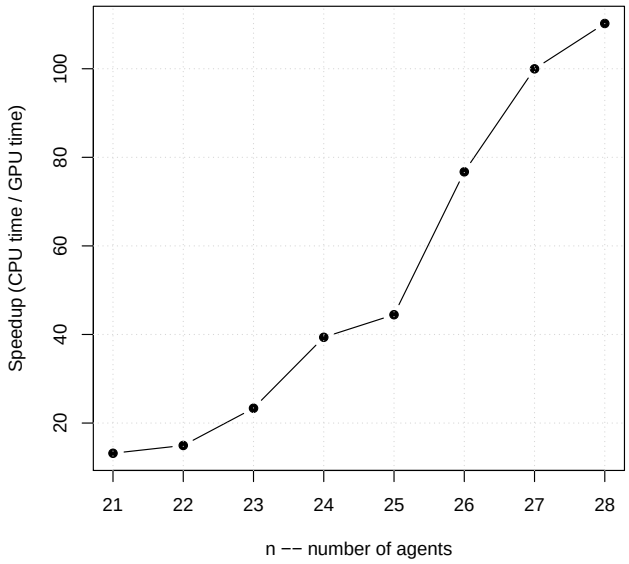

Figure 2: The speedup of IDP^G over IDP, represented as the ratio between the run time of both algorithms given different numbers of agents.

4. PERFORMANCE EVALUATION

In this section, we benchmark IDP^G against IDP to evaluate the effectiveness of using GPUs. We perform our experiments on a PC machine equipped with Intel Pentium G620 (2.60GHz) CPU, 4GB of RAM and the NVIDIA GeForce GTX 660 GPU with 960 cores and 2GB of on-board memory. Our implementation of IDP does not involve any parallelization—it only utilizes a single CPU core.

We randomly generated (based on a uniform distribution) multiple problem instances that vary in the number of agents involved. While we do report the distribution from which the values where sampled, replacing this with any other set of values would not affect the run time. This is because the number of operations performed by IDP^G depends solely on the number of agents involved, i.e., it is not influenced by the other factors, such as the value distribution for example.

Given different numbers of agents, Figure 3 shows on a *log scale* the total run time of both IDP and IDP^G (measured in clock time), while Figure 2 shows the ratio between the two running times. The exact numbers are provided in Table 2. As can be seen, IDP^G is significantly faster than IDP. This speedup increases with the number of agents, and reaches two orders of magnitude (110 times faster to be more precise) as soon as the number of agents reaches 28. This is despite the fact that our implementation of IDP is highly optimized. In fact, it is more optimized than Rahwan et al.'s own implementation of IDP. For example, given 27 agents, our implementation took 7.5 hours, while theirs took 2.5 days on a processor with almost identical computational capabilities compared to ours [20].

n	s. points	GPU	CPU	speedup
21	10	1	12	13
22	11	2	30	15
23	11	5	117	23
24	12	12	491	39
25	12	32	1439	44
26	13	84	6471	77
27	13	267	26647	100
28	14	665	73260	110

Table 2: The number of synchronization points of IDP^G, running times in seconds of IDP^G vs IDP and the speedup factor for different problem sizes.

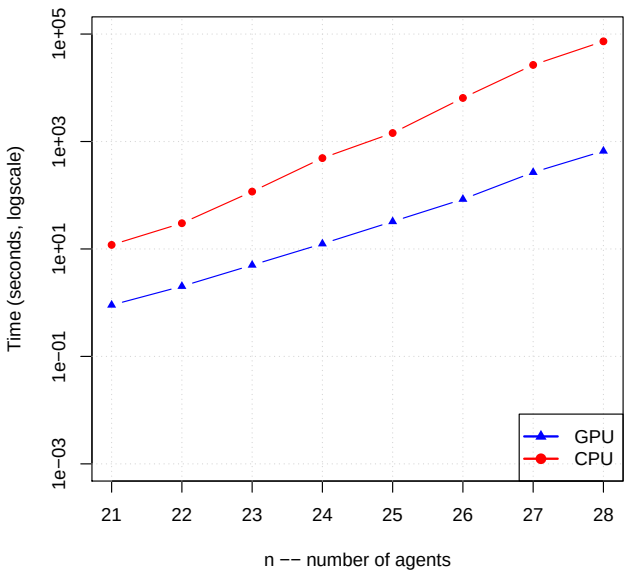

Figure 3: Running time (measured in seconds) of IDP and IDP^G given different numbers of agents. Results are plotted on a *log scale*.

5. RELATED WORK

The computational power of GPUs has been utilized with various degrees of success. For instance, for certain computations related to MRI scanners, GPUs provided a speedup factor of 431, while a problem of generating hashes was solved 11 times faster, compared to a CPU [21]. More generally, whenever an algorithm has data-independent sub-routines which may be run concurrently, a GPU will most likely perform better than a CPU. Combining the GPU together with dynamic programming has been used before to solve similar combinatorial optimization problems. Boyer, et al. [7] successfully implemented and solved the knapsack problem with a speedup factor of 26.

Now, turning to the CSG problem, in terms of parallel programming approaches, we note the work of Michalak et. al. [13]. In particular, the authors proposed an algorithm called D-IP—a distributed version of another coalition structure generation algorithm called IP [20]. D-IP incorporates a number of techniques, one of which involves the distribution of the search space among the agents. The implementation was based on 14 dual-core workstations that shared the computational burden. An empirical evaluation over different problem instances showed that D-IP takes around 5% to 10% of the time taken by the centralized IP algorithm. However, the problem with D-IP (as far as GPUs are concerned) is that it requires data to be shared between computational nodes over potentially slow Ethernet links, and also sharing redundant copies of the characteristic function table across those nodes. Furthermore, D-IP inherits the same weaknesses that IP has (compared to IDP and IDP^G); the worst-case runtime is $O(n^n)$, while the worst-case runtime of IDP or IDP^G is $O(3^n)$.

6. CONCLUSIONS

Graphics Processing Units (GPUs) promise speedups of several orders of magnitude, but demand re-designing existing algorithms to meet certain requirements imposed by the GPU framework. We bring those challenges to the attention of the Computational Coalition Formation community, and develop IDP^G—the first GPU-based coalition structure generation algorithm. Our algorithm is a GPU-based reformulation of a previous algorithm called IDP. We prove a certain property of that previous algorithm, and show how this property can be useful when reducing global memory access. Furthermore, we prove that our algorithm minimizes the number of synchronization points—a property desired in GPU algorithms. Finally, we test our GPU version against the original one, and show that ours is faster by two orders of magnitude. The community can benefit from the open-source implementation, which is made publicly available[6].

Future directions include developing GPU versions of other coalition structure generation algorithm, such as, IDP-IP*.

7. ACKNOWLEDGMENTS

Tomasz Michalak was supported by the European Research Council under Advanced Grant 291528 ("RACE").

8. REFERENCES

[1] J. Adams and T. Service. Constant factor approximation algorithms for coalition structure generation. *Autonomous Agents and Multi-Agent Systems*, 23(1), 2011.

[2] M. Arora. *The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing*. Research survey, University of California, San Diego, 2012.

[3] H. Aziz and B. de Keijzer. Complexity of coalition structure generation. In *AAMAS*, 2011.

[4] Y. Bachrach and J. S. Rosenschein. Coalitional skill games. In *AAMAS*, 2008.

[5] E. Bitar, E. Baeyens, P. Khargonekar, K. Poolla, and P. Varaiya. Optimal sharing of quantity risk for a coalition of wind power producers facing nodal prices. In *ACC*, 2012.

[6] A. Bleiweiss. Gpu accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS*, pages 65–74. Eurographics Association, 2008.

[7] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on gpu. *OR*, 39(1), 2012.

[8] Z. Du, Z. Yin, and D. Bader. A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda. In *IPDPS - IEEE*, pages 1–8. IEEE, 2010.

[9] R. Glinton, P. Scerri, and K. Sycara. Agent-based sensor coalition formation. In *Information Fusion*, number CMU-RI-TR-, 2008.

[10] C. Li, K. Sycara, and A. Scheller-Wolf. Combinatorial coalition formation for multi-item group-buying with heterogeneous customers. *Decision Support Systems*, 49(1):1–13, 2010.

[6]For the IDP^G implementation developed as part of this research, see: *https://github.com/idpg/idpg*

[11] W. Lucas and R. Thrall. n-person games in partition function form. *Naval Research Logistic Quarterly*, pages 281–298, 1963.

[12] F. C. Mencia. Optimizing performance for coalition structure generation problems in multicore systems. Master's thesis, Universitat Autonoma de Barcelona, Spain, 2012.

[13] T. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N. Jennings. A distributed algorithm for anytime coalition structure generation. In *AAMAS*, 2010.

[14] N. Ohta, V. Conitzer, R. Ichimura, Y. Sakurai, A. Iwasaki, and M. Yokoo. Coalition structure generation utilizing compact characteristic function representations. In *CP*, 2009.

[15] J. Pan, C. Lauterbach, and D. Manocha. g-planner: Real-time motion planning and global navigation using gpus. In *AAAI*, pages 1245–1251, 2010.

[16] T. Rahwan and N. R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS*, 2008.

[17] T. Rahwan, T. Michalak, and N. R. Jennings. A hybrid algorithm for coalition structure generation. In *AAAI*, 2012.

[18] T. Rahwan, T. P. Michalak, E. Elkind, P. Faliszewski, J. Sroka, M. Wooldridge, and N. R. Jennings. Constrained coalition formation. In *AAAI*, 2011.

[19] T. Rahwan, T. P. Michalak, and N. R. Jennings. Minimum search to establish worst-case guarantees in coalition structure generation. In *IJCAI*, 2011.

[20] T. Rahwan, S. D. Ramchurn, A. Giovannucci, and N. R. Jennings. An anytime algorithm for optimal coalition structure generation. *JAIR*, 34:521–567, 2009.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *ACM SIGPLAN*, 2008.

[22] T. W. Sandholm and V. R. Lesser. Coalitions among computationally bounded agents. *Artifcial Intelligence*, 94(1), 1997.

[23] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[24] M. Tsvetovat, K. P. Sycara, Y. Chen, and J. Ying. Customer coalitions in the electronic marketplace. In *AA*, 2000.

[25] S. Ueda, A. Iwasaki, M. Yokoo, M. C. Silaghi, K. Hirayama, and T. Matsui. Coalition structure generation based on distributed constraint optimization. In *AAAI*, 2010.

[26] S. Ueda, M. Kitaki, A. Iwasaki, and M. Yokoo. Concise characteristic function representations in coalitional games based on agent types. In *IJCAI*, 2011.

[27] D. Y. Yeh. A dynamic programming approach to the complete set partitioning problem. *BIT Numerical Mathematics*, 26(4):467–474, 1986.

[28] K. Zhang and J. Kang. Real-time 4d signal processing and visualization using graphics processing unit on a regular nonlinear-k fourier-domain oct system. *Optics express*, 18(11):11772–11784, 2010.