Department of Computer Science

# Coalition Structure Generation with the Graphic Processor Unit

**Krzysztof Pawłowski[*], Karol Kurach[*], Tomasz Michalak, Talal Rahwan**

[*] Both Krzysztof Pawłowski and Karol Kurach are the main authors of this paper.

CS-RR-13-07

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD

# Coalition Structure Generation with the Graphic Processor Unit

Krzysztof Pawłowski[1*], Karol Kurach[1*], Tomasz Michalak[2,1], Talal Rahwan[3]
[1] Institute of Informatics, University of Warsaw, Poland
[2] Department of Computer Science, University of Oxford, UK
[3] School of Electronics and Computer Science, University of Southampton, UK
[*] Both Krzysztof Pawłowski and Karol Kurach are the main authors of this paper.

**Abstract**

Coalition Structure Generation—the problem of finding the optimal set of coalitions—has received considerable attention in recent AI literature. The fastest exact algorithm to solve this problem is IDP-IP$^*$, due to Rahwan et al. (2012). This algorithm is a hybrid of two previous algorithms, namely IDP and IP. As such, 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 speed ups. We show that IDP$^G$ is faster than IDP by two orders of magnitude.

## 1. Introduction

Coalitional games have been studied in various areas of artificial intelligence and multi-agent systems [3, 11, 16]. By cooperating, agents are often able to enhance their performance and achieve tasks otherwise unachievable. It is applicable 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 [7], or reduce the uncertainty that green-energy generators have about their own production [4], or allow buyers to obtain cheaper prices through bulk purchasing [8].

In general, the effectiveness of a coalition can be influenced by other co-existing coalitions. Such settings are known as *partition function games* [9]. 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 [4, 7, 21, 19]. 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., [10, 17, 13], while others are tailored for certain classes or alternative representations of CFGs, e.g., [3, 22, 16, 2, 23]. 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$^*$, due to Rahwan et al. (2012). As the name suggests, this algorithm is a hybrid of two previous algorithms, namely IDP [14] and IP [15]. 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 talking all cores into consideration) is more than that of a CPU. This, people have realised, meant that almost any algorithm can run faster on a GPU than a CPU, provided that this algorithm can be efficiently parallelised. Such capabilities gave rise to a new line of research known as GPGPU (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 [5, 12], computational biology [6], linear algebra [20], signal processing [25], 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 computationl power) over CPUs has been growing in recent years [1]. 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 prob-

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

lem 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 centralised *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. In other words, when breaking the optimization problem into smaller subproblems that are each solved by a separate thread, one might consider it optimal to have as many threads as there are cores on the GPU. However, this will result in an inefficiency due to *memory latency*. In particular, whenever a thread needs to access global memory, the core that is executing it would remain idle. 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 instruction 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 summarised 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 points. We then prove a certain property of IDP, and exploit it when reducing global memory access in IDP$^G$.

[2]This is due to hardware specifications of GPUs, which are beyond the scope of this paper. More can be found in NVIDIA's *"CUDA C Best Practice Guide"*, `http://docs.nvidia.com`

- 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. Finally, Section 5 concludes the paper and outlines future directions.

## 2. Background

In this section we formally state the coalition structure generation problem, and then describe IDP 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 numerical value, known as *the value of* $C$, and denoted by $v(C)$. Formally, $v : 2^A \to \mathbb{R}$. Now, let $\Pi^A$ 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 $CS^*$.

Now, we turn our attention to IDP. To help the reader understand how it works, we need to first explain a preliminary version, called DP [24]. To this end, we will refer to any set of disjoint coalitions as a *"partition"*, denoted $P$. Only when such a partition contains all agent will we ever use the term *"coalition structure"*. Now, for any coalition $C \subseteq A$, let $\Pi^C$ 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 $V(P)$ be the value of partition $P$, where $V(P) = \sum_{P_i \in P} v(P_i)$. 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 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 the one that maximises $V(C') + V(C'')$, 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, DP 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 summarise, 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 $n$.

Moving back to IDP, Rahwan and Jennings (2008b) 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 splitting some coalition of size $c$ into two coalitions of sizes $c'$ and $c''$, where $(c', c'')$ is in:

$$dep(c) = \Big\{(c', c'') \in \mathbb{N}^2 : (c' \geq c'') \lor (c' + c'' = n) \lor (c' \leq n - c' - c'')\Big\}$$

Based on this, for any given coalition $C$ such that $|C| = c$, IDP only evaluates the splits in $\Pi^C$ that involve splitting $C$ 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 $c \in \left[\left\lfloor \frac{2n}{3} \right\rfloor + 1, n - 1\right]$. However, the link between the value of $c$ and the elements in $dep(c)$ was not formalised for cases where $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$.

## 3. IDP$^G$

In this section, we present IDP$^G$—a parallelised version of IDP, designed to meet the desiderata outlined in the introduction. 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 paralellized 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 lower than $\lceil n/2 \rceil - 1$.
- Section 3.2 presents focuses on the other desired properties that we set earlier in the introduction, and presents the relevant pseudo codes.

### *3.1. Handling Synchronization Points*

In order to parallelise 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., 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.

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

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

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 computed as follows:

$$l(c) = \begin{cases} c - 1 & \textbf{if } 1 \le c \le \left\lfloor \frac{n+1}{2} \right\rfloor \\ n - c & \textbf{if } \left\lceil \frac{n}{2} \right\rceil + 1 \le c \le \left\lfloor \frac{2n}{3} \right\rfloor \\ 0 & \textbf{if } \left\lceil \frac{2n+1}{3} \right\rceil \le c \le n-1 \\ \left\lceil \frac{n}{2} \right\rceil & \textbf{if } c = n \end{cases} \tag{2}$$

To prove that the above assignment of subproblems to stages is correct, we need to prove that, whenever a subproblem depends on another, 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 any $c = 1, \ldots, n$, and for any $(c', c'') \in dep(c)$, the following holds:*

$$\Big(l(c') < l(c)\Big) \textbf{\textit{ and }} \Big(l(c'') < l(c)\Big) \tag{3}$$

*Proof.* Since $(c', c'')$ is in $dep(c)$, then from (1) ...x

We will prove that (3) holds for each

***Case 1:*** $c = n$.

From $l$ definition (Equation 2):

$$l(c) = \left\lceil \frac{n}{2} \right\rceil$$

But

$$\forall_{s \neq n} l(s) < \left\lceil \frac{n}{2} \right\rceil$$

And

$$c' < c \text{ and } c'' < c$$

Thus

$$l(c') < l(c) \textbf{ and } l(c'') < l(c)$$

$$\square$$

***Case 2:*** $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$ ***and*** $c \leq \left\lfloor \frac{n+1}{2} \right\rfloor$.

From $l$ definition (Equation 2) we have:

$$l(c) = c - 1$$

And since $c', c'' < c \leq \frac{n+1}{2}$ also:

$$l(c') = c' - 1 \textbf{ and } l(c'') = c'' - 1$$

Thus:

$$l(c') < l(c) \textbf{ and } l(c'') < l(c)$$

$$\square$$

***Case 3 ($c'$):*** $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$ ***and*** $\left\lceil \frac{n}{2} \right\rceil + 1 \leq c \leq \left\lfloor \frac{2n}{3} \right\rfloor$.

From the definition of $l$ (Equation 2) we have:

$$l(c) = n - c$$

From $c' \leq n - c$ (Equation 1) and $\left\lceil \frac{n}{2} \right\rceil + 1 \leq c$ (case):

$$c' \leq \left\lfloor \frac{n}{2} \right\rfloor - 1 = \left\lceil \frac{n-2}{2} \right\rceil \leq \left\lfloor \frac{n+1}{2} \right\rfloor$$

Thus from the definition of $l$ (Equation 2):

$$l(c') = c' - 1$$

Again from $c' \leq n - c$ (Equation 1) and above:

$$l(c') < n - c$$

But we already established that $l(c) = n - c$, thus:

$$l(c') < l(c)$$

$$\square$$

***Case 3 ($c''$):*** $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$ ***and*** $\left\lceil \frac{n}{2} \right\rceil + 1 \leq c \leq \left\lfloor \frac{2n}{3} \right\rfloor$.

Starting from $\left\lceil \frac{c}{2} \right\rceil \leq c'$ (Equation 1):

$$-c' \leq -\left\lceil \frac{c}{2} \right\rceil$$

$$c - c' \leq c - \left\lceil \frac{c}{2} \right\rceil$$

$$c'' \leq \left\lfloor \frac{c}{2} \right\rfloor \leq \left\lfloor \frac{n}{2} \right\rfloor \leq \left\lfloor \frac{n+1}{2} \right\rfloor$$

From the definition of $l$ (Equation 2) and above:

$$l(c'') = c'' - 1 < \left\lfloor \frac{c}{2} \right\rfloor \leq \frac{c}{2}$$

From the assumption $c \leq \left\lfloor \frac{2n}{3} \right\rfloor$:

$$l(c'') < \frac{\left\lfloor \frac{2n}{3} \right\rfloor}{2} \leq \frac{n}{3}$$

From the same assumption:

$$-\left\lfloor \frac{2n}{3} \right\rfloor \leq -c$$

$$n - \left\lfloor \frac{2n}{3} \right\rfloor \leq n - c$$

$$\left\lceil \frac{n}{3} \right\rceil \leq l(c)$$

Thus $l(c'') < \frac{n}{3}$ while $\left\lceil \frac{n}{3} \right\rceil \leq l(c)$. This yields:

$$l(c'') < l(c)$$

□

□

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

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

*Proof.* Let $Dep$ be a function that maps a subproblem 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:

$$C \rightsquigarrow D \textbf{ iff } (C, D \setminus C) \in Dep(D)$$

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

$$\forall_{i=\{1,\dots,k\}} C_{i-1} \rightsquigarrow C_i \textbf{ and } \text{computation for } C_{i-1} \text{ is needed} \tag{4}$$

```
Input: $f$, $n$
Output: $f$
Copy $f$ from host to the device ;
for $s \leftarrow 2$ to $n$ do
    if $s > \lceil n/2 \rceil$ and $s \neq n$ then
        continue;
    // Call the GPU code, see algorithm 2
    spawn $\binom{n}{s}$ threads on GPU with params ($f$, $n$, $s$) $s' \leftarrow n - s + 1$ ;
    if $\neg(2n < 3s'$ and $s' < n)$ and $s \neq n$ then
        // Run another stage in parallel.
        spawn $\binom{n}{s'}$ extra threads with params ($f$, $n$, $s'$)
    waits for all threads to finish
end
Copy $f$ from the device to host;
```

**Algorithm 1**: HOST CODE THAT MANAGES THE GPU CODE

We construct such a $\lceil \frac{n}{2} \rceil$-element sequence:

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

Condition (4) is met because we have:

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

which follows directly from the definition of *dep* (see Equation 1). This concludes the proof that there exists a $(k+1)$-element sequence of coalitions that satisfies (4). Since every element of the sequence requires a computation and needs to be in separate stage (because it depends on the previous element in the sequence), 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 required synchronization points.

$\square$

### *3.2. Handling Other Design Requirements*

One of the innovations of IDP$^G$ over IDP is how the splits are evaluated. As stated in desiderata, minimizing global memory access if of utmost performance and translates to direct performance gain. The number of executed instructions that do not access global memory is less important. To this end, IDP$^G$ checks every possible split of subproblems it analyzes. However, only the necessary cause slow memory access. To keep an overhead small, enumeration using fast bitmask operations (as described by Francesco etc) is used. For each potential split, IDP$^G$ strives to keep the number of the execution cost as small as possible. In general, it checks conditions developed by IDP to see if a given split is needed. As it turns out, some checks can be avoided altogether, due to the following result.

**Theorem 3.** *All splits of coalitions of size c, where $|c| \leq \frac{n}{2}$, need to be evaluated.*

```
  Input: $f$, $n$, $s$
  Output: $f$
1 // Every thread has unique index $\in [0, \dots, \binom{n}{s} - 1]$
2 id ← index of current thread;
3 if $id < \binom{n}{s}$ then
4     $C \leftarrow (id + 1)$-th $k$-element subset of $A$;
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$(f[C], x)$;
9 end
```

**Algorithm 2**: CODE THAT IS RUN ON THE GPU

```
  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 $|C'| \geq 1/2 * |C|$ and $|C| = n$;
```

**Algorithm 3**: CHECKSPLIT CONDITION (SMALL)

*Proof.* Recall how eq. 1 specifies when a split is needed. IDP$^G$ iterates efficiently over all the potential splits that meet cases 1-3 of the aforementioned equation. The only thing left to prove is that the condition $c' \leq n - c$ holds, thus that the split is necessary. The assumptions in this case are:

- $c \leq \frac{n}{2}$ **and**
- $c' + c'' = c$ **and**
- $1 \leq c' \leq c - 1$.
- $\lceil \frac{c}{2} \rceil \leq c'$.

By transforming the above inequalities we get:

$$-\frac{n}{2} \leq -c$$

$$c' \leq c \leq \frac{n}{2} = n - \frac{n}{2}$$

And then by substituting the right side of former for $-\frac{n}{2}$ in latter we get the desired:

$$c' \leq n - c$$

Thus, for coalitions of sizes at most $\frac{n}{2}$, above check can be safely avoided – we just proved that the condition is always met and therefore cannot lead to a reduced numer of global memory accesses. □

```
    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 $|C| = n)$;
```
**Algorithm 4**: CHECKSPLIT CONDITION (LARGE)

## 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 paralellisation, and so it only utilizes a single CPU core.

Given different numbers of agents, Figure 1 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. As can be seen, IDP$^G$ is significantly faster than IDP. This speed up 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 [15].

## 5. 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, important 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[3].

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

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

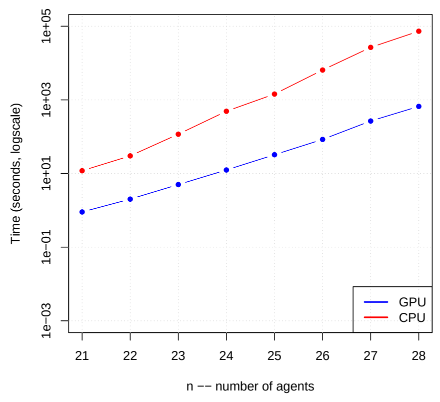

Figure 1: Running time (measured in seconds) of IDP and IDP$^G$ given different numbers of agents. Results are plotted on a *log scale*.

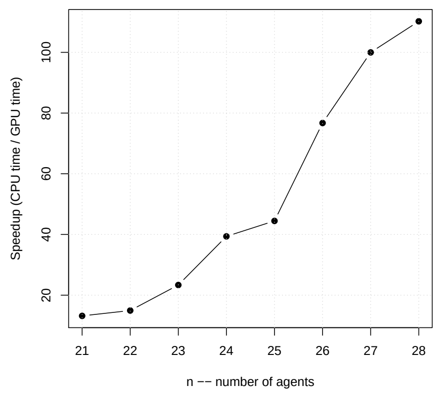

Figure 2: The speed up of IDP$^G$ over IDP, represented as the ratio between the run time of both algorithms given different numbers of agents.

## References

[1] Manish Arora. *The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing*. Research survey, Department of Computer Science and Engineering, University of California, San Diego, 2012.

[2] Haris Aziz and Bart de Keijzer. Complexity of coalition structure generation. In *AAMAS '11: Tenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 191–198, 2011.

[3] Yoram Bachrach and Jeffrey S. Rosenschein. Coalitional skill games. In *AA-*

*MAS'08: Seventh International Conference on Autonomous Agents and Multi-Agent Systems*, pages 1023–1030, 2008.

[4] E.Y. Bitar, E. Baeyens, P.P. Khargonekar, K. Poolla, and P. Varaiya. Optimal sharing of quantity risk for a coalition of wind power producers facing nodal prices. In *Proceedings 31st IEEE American Control Conference*, 2012.

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

[6] Z. Du, Z. Yin, and D.A. Bader. A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[7] Robin Glinton, Paul Scerri, and Katia Sycara. Agent-based sensor coalition formation. In *Proceedings of the International Conference on Information Fusion*, number CMU-RI-TR-, July 2008.

[8] Cuihong Li, Katia Sycara, and Alan Scheller-Wolf. Combinatorial coalition formation for multi-item group-buying with heterogeneous customers. *Decis. Support Syst.*, 49(1):1–13, April 2010.

[9] William Lucas and Robert Thrall. $n$-person games in partition function form. *Naval Research Logistic Quarterly*, pages 281–298, 1963.

[10] Tomasz Michalak, Jacek Sroka, Talal Rahwan, Michael Wooldridge, Peter McBurney, and Nicholas R. Jennings. A Distributed Algorithm for Anytime Coalition Structure Generation. In *AAMAS '10: Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 1007–1014, 2010.

[11] Naoki Ohta, Vincent Conitzer, Ryo Ichimura, Yuko Sakurai, Atsushi Iwasaki, and Makoto Yokoo. Coalition structure generation utilizing compact characteristic function representations. In *CP'09: 15th International Conference on Principles and Practice of Constraint Programming*, pages 623–638, 2009.

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

[13] Talal Rahwan and Nicholas R. Jennings. Coalition structure generation: Dynamic programming meets anytime optimisation. In *AAAI'08: Twenty Third AAAI Conference on Artificial Intelligence*, pages 156–161, 2008.

[14] Talal Rahwan and Nicholas R. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS'08: Seventh International Conference on Autonomous Agents and Multi-Agent Systems*, pages 1417–1420, 2008.

[15] Talal Rahwan, Sarvapali D. Ramchurn, Andrea Giovannucci, and Nicholas R. Jennings. An anytime algorithm for optimal coalition structure generation. *Journal of Artificial Intelligence Research (JAIR)*, 34:521–567, 2009.

[16] Talal Rahwan, Tomasz P. Michalak, Edith Elkind, Piotr Faliszewski, Jacek Sroka, Michael Wooldridge, and Nicholas R. Jennings. Constrained coalition formation. In *Twenty Fifth AAAI Conference on Artificial Intelligence (AAAI)*, pages 719–725, 2011.

[17] Talal Rahwan, Tomasz P. Michalak, and Nicholas R. Jennings. Minimum search to establish worst-case guarantees in coalition structure generation. In *IJCAI'11: Twenty Second International Joint Conference on Artificial Intelligence*, pages 338–343, 2011.

[18] Talal Rahwan, Tomasz Michalak, and Nicholas R. Jennings. A hybrid algorithm for coalition structure generation. In *Twenty Sixth Conference on Artificial Intelligence (AAAI-12)*, Toronto, Canada, 2012.

[19] T. W. Sandholm and V. R. Lesser. Coalitions among computationally bounded agents. *Artificial Intelligence*, 94(1):99–137, 1997.

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

[21] M. Tsvetovat, K. P. Sycara, Y. Chen, and J. Ying. Customer coalitions in the electronic marketplace. In *Proceedings of the Fourth International Conference on Autonomous Agents (AA-01)*, pages 263–264, 2000.

[22] Suguru Ueda, Atsushi Iwasaki, Makoto Yokoo, Marius Calin Silaghi, Katsutoshi Hirayama, and Toshihiro Matsui. Coalition structure generation based on distributed constraint optimization. In *Twenty Fourth AAAI Conference on Artificial Intelligence (AAAI)*, pages 197–203, 2010.

[23] Suguru Ueda, Makoto Kitaki, Atsushi Iwasaki, and Makoto Yokoo. Concise characteristic function representations in coalitional games based on agent types. In *IJCAI'11: Twenty Second International Joint Conference on Artificial Intelligence*, pages 393–399, 2011.

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

[25] K. Zhang and J.U. 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.