Received: 9 April 2018 | Revised: 26 October 2018 | Accepted: 29 October 2018
DOI: 10.1002/cpe.5087

WILEY

SPECIAL ISSUE PAPER

A hybrid CPU-GPU-MIC algorithm for minimal hitting set enumeration

Danilo Carastan-Santos[1,2] | David C. Martins-Jr[1] | Siang W. Song[3] | Luiz C.S. Rozante[1] | Raphael Y. de Camargo[1]

[1]Center of Mathematics, Computing and Cognition, Universidade Federal do ABC, Santo André, Brazil,
[2]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France
[3]Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, Brazil

Correspondence
Danilo Carastan-Santos, Center of Mathematics, Computing and Cognition, Universidade Federal do ABC, Santo André, Brazil; or Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France.
Email: danilo.santos@ufabc.edu.br

Funding information
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq, Grant/Award Number: 465446/2014-0, 559955/2010-3, and 302620/2014-1; CAPES, Grant/Award Number: 88887.136422/2017-00; FAPESP, Grant/Award Number: 14/50937-1, 15/24485-9, 13/26644-1, 14/50937-1, and 15/01587-0

Summary

We present a hybrid exact algorithm for the Minimal Hitting Set (MHS) Enumeration Problem for highly heterogeneous CPU-GPU-MIC platforms. With several techniques that permit an efficient exploitation of each architecture, low communication cost, and effective load balancing, we were able to enumerate MHSs for large instances in reasonable time, achieving good performance and scalability. We obtained speedups of up to 25.32 in comparison with using two six-core CPUs and we also enumerated MHSs for instances with tens of thousands of variables in less than 5 hours. We also evaluated our algorithm with a real-world driven dataset, and with a large CPU-GPU cluster, we unprecedentedly enumerated in parallel large minimal hitting sets of this dataset in less than 8 hours. These results reinforce the statement that heterogeneous clusters of CPUs, GPUs, and MICs can be used efficiently for high-performance computing.

KEYWORDS

GPU, high performance computing, hitting set, hybrid parallel programming, MIC

1 | INTRODUCTION

Many industrial or scientific applications involve, in part or as a whole, solving the Minimal Hitting Set (MHS) enumeration problem. The MHS enumeration problem is an arguably more constrained variant of the traditional Hitting Set Problem (HSP) whose main objective, informally, is to find a set of variables with smallest size that satisfies every element in a finite set of constraints. This satisfiability reasoning is present, for example, in problems from Systems Biology, such as genomic reversal distance,[1] classification models,[2] polymerase chain reaction experiments,[3] and gene regulatory networks (GRN) inference.[4-8]

A common characteristic of these problems is that the input size of the modeled MHS and HSP instances is often large, making the retrieval of the exact solutions impracticable for traditional algorithms. Moreover, the MHS enumeration problem adds two major difficulties in comparison to the traditional HSP: (i) an MHS solution H must be a hitting set (similarly with HSP), but no other subset of H is allowed to be a hitting set, and (ii) we are not only concerned to enumerate MHSs of minimum size but as well all MHSs from a minimum size where solutions exist to a certain maximum size k. These difficulties lead to a considerably larger search space, and thus, enumerating MHSs of any size even for small instances is often impracticable for exact algorithms.

With the advent of new accelerator technologies such as Many Integrated Core (MIC) architectures,[9] hybrid heterogeneous platforms composed of CPUs, GPUs, and MICs became a common occurrence in research centers. In order to fully exploit the computer power of these novel hybrid platforms, there is a growing effort to develop hybrid applications, ie, applications capable of using CPU, GPU and MIC processors together.[10-13] However, this growing effort is still small mostly due to two major hindrances: (i) these heterogeneous platforms demand hybrid

algorithms to efficiently exploit all co-processor architectures in conjunction and (ii) the efficiency of the usage of CPU-GPU-MIC platforms in a wide range of applications is not clearly known. In regards to HSP, a first attempt performed by our previous works to solve this problem with accelerator devices was to perform exhaustive search with multiple GPUs[6,7] and with hybrid CPU-GPU-MICs platforms,[8] where we could solve synthetic HSP instances constituted by thousands of variables and constraints. To the best of our knowledge, there is no exact MHS enumeration algorithm in the literature that makes use of accelerator devices (CPUs, GPUs, or MICs), either separately or in conjunction.

In this paper, we propose and evaluate a highly parallel and scalable algorithm for the enumeration of Minimal Hitting Sets in hybrid CPU-GPU-MIC clusters. This work has the following contributions:

- We propose a hybrid exact Minimal Hitting Set (MHS) enumeration algorithm for CPU-GPU-MIC heterogeneous platforms, which efficiently exploits the advantages of each individual architecture on the heterogeneous cluster, efficiently minimizes communication among nodes and properly balances the load among the processors.
- We report a performance evaluation of the heterogeneous CPU-GPU-MIC cluster usage with our hybrid MHS enumeration implementation. The experimental results show that by (i) properly setting the load balancing, (ii) exploiting each individual architecture advantages, and (iii) minimizing node to node communication cost, we can effectively use heterogeneous CPU-GPU-MIC clusters to enumerate MHSs of instances with tens of thousands of variables in reasonable time.
- Our proposed MHS algorithm enumerated all minimal hitting sets up to size 10 from the `HER2+ SHORT`[14] cell signaling network in less than 8 hours and with no memory exhaustion, using a cluster constituted by 12 CPUs and 12 GPUs. To the best of our knowledge, this is an unprecedented feat for parallel minimal hitting set enumeration algorithms.

This work extends our previous paper,[8] generalizing the original algorithm to solve the minimal hitting set enumeration problem. We performed new performance evaluations over the generalized algorithm. Finally, we included the evaluation of a real cell signaling network, using the `HER2+ SHORT`[14] dataset.

The remaining parts of this manuscript are organized as follows. In Section 2, we present recent works about the Minimal Hitting Set (MHS) enumeration problem and the Hitting Set Problem (HSP). Section 3 presents a formal definition of HSP and the MHS enumeration problem, followed by our exact hybrid algorithm design, including the load balancing procedure. In Section 4, we present the specific processing modules for CPU-MIC and GPU accelerator devices. The experimental results showing the performance evaluation are presented and discussed in Section 5, and in Section 6, we present the main conclusions.

2 | RELATED WORK

Since the Hitting Set problem is NP-hard,[15] there exist some solutions that try to circumvent this problem, either by imposing restrictions, such as with non-polynomial exact algorithms,[16] or by using heuristics and approximations.[17,18] More recently, Ruchkys and Song[5] proposed sequential and parallel approximation algorithms to solve HSP, tailored for systems biology problems. Steinbach and Postohoff[19] proposed an exact, exhaustive search driven algorithm using single GPU (Graphics Processing Units) to determine the minimum cover of Boolean expressions, which is dual to the HSP. However, such algorithm is not capable of dealing with high dimension input sets (order of thousands of variables) and multiple GPUs. In our previous works, we also followed an exhaustive search driven approach, though with multiple GPUs[6,7] and with hybrid CPU-CPU-MICs platforms,[8] with many improvements in regards to scalability, memory consumption, and processing performance which enabled us to solve synthetic HSP instances with high dimension.

Although closely related to HSP, the MHS enumeration problem remains with its complexity unknown,[20] so far only non polynomial-time algorithms are known and it is a notorious open problem to find polynomial-time algorithms to solve it. Gainer-Dewar and Vera-Licona[20] present an extensive review of recent MHS enumeration algorithms. In the literature many algorithms were proposed either to solve MHS directly or to solve a dual problem, and their approaches usually fall down into three broad categories: (i) clauses iteration,[21-23] in which MHSs are built by iterating in the constraints, (ii) solution buildup,[24,25] where MHSs are built by iterating at each variable and keeping track on satisfied constraints and (iii) divide and conquer,[26,27] where the constraints are partitioned, sub MHSs are built by each partition and then combined together to build the global MHSs.

The common point of all of these categories and their algorithms is that, although these algorithms provide clever approaches to enumerate MHSs, they are either sequential with procedures that are hard to parallelize, or multithreaded parallel with no aid of accelerator devices (notably GPUs and/or MICs), and with procedures that lead to memory exhaustion[20] even for medium sized MHS instances.

3 | HYBRID CPU-GPU-MIC MINIMAL HITTING SET ENUMERATION

Formally, the Hitting Set Problem (HSP for short) can be defined as follows. Given a finite set X, a collection $\mathcal{S}$ of subsets of X, which we call a collection of clauses, and a positive integer k, the goal is to find a subset $H \subseteq X$ such that:

$$|H| \leq k \text{ and } H \cap S \neq \emptyset, \forall S \in \mathcal{S}. \quad (1)$$

More than one subset of X may satisfy the conditions above. We call $\mathcal{H} = \{H_1, H_2, \ldots, H_{|\mathcal{H}|}\}$ the collection of possible solutions.

In contrast with our previous works,[7,8] here, we are concerned about the Minimal Hitting Set (MHS for short) enumeration problem. In this problem, we call a subset $H \subseteq X$ as a Minimal Hitting Set of $\mathcal{S}$ if H is a Hitting Set (ie, H satisfies Equation (1)), but no other subset $G \subset H$ is a Hitting Set as well. Moreover, we are concerned on enumerating all Minimal Hitting Sets $H \in \mathcal{H}$, up to a set cardinality k, that is, all MHSs whose size are on the interval $[1, k]$.

In our previous works,[7,8] we proposed an enhanced exhaustive search algorithm for HSP that enumerates and evaluates all candidate solutions, which are encoded as combinations of variables of X, in increasing order of cardinality $i, 1 \leq i \leq k$ and stops when a solution is found. It is important to note that, in this previous version, the algorithm stops at the smallest cardinality in which solutions exist, thus not necessarily executing up to cardinality k. In this work, we propose an extension of this algorithm that enables the enumeration of all MHSs up to cardinality k.

Extending an HSP algorithm to work for the MHS enumeration problem is not a trivial task. Though similar with HSP, the MHS enumeration problem adds another hindrance which we elucidate with the following proposition:

Proposition 1. *Given a finite set X, a collection $\mathcal{S}$ of subsets of X and a maximum solution size $k < |X|$, if there exists a Hitting Set H for $\mathcal{S}$ with size $|H| = j, j < k$, there exists at least one Hitting Set H' for $\mathcal{S}$ with size $i, j < i \leq k$ that is not a Minimal Hitting Set.*

Proof. From the proposition, we can see that $|H| < |X|$ and hence $H \subset X$. Therefore, we can always construct a Hitting Set $H' = \{H \cup Y, Y \subseteq X \setminus H, Y \neq \emptyset\}$. Since $H \subset H'$, H' is not a Minimal Hitting Set. □

As a consequence of the Proposition 1, any HSP algorithm will not work naturally for the MHS problem, thus demanding clever approaches. We present our novel MHS enumeration algorithm, which addresses the aforementioned hindrance, in the next sections.

3.1 | Enhanced parallel exhaustive search for MHS enumeration

As aforementioned, our MHS enumeration algorithm enumerates and evaluates all candidate solutions, which are encoded as combinations of variables of X, in increasing order of cardinality $i, 1 \leq i \leq k$.

The evaluation process is a disjunction check with the clauses in a sorted collection SortS, which is the collection $\mathcal{S}$ whose clauses are sorted in increasing order of its sizes. Evaluating the candidate solution with SortS allows an efficient discarding of non solution candidates.

We define a heterogeneous CPU-GPU-MIC platform as a set of c machines connected by a network, in which the jth machine has g_j processing units (PUs), which can be CPUs, GPUs or MICs. We adopted a master-slave approach, where the master process is responsible for assigning work to the slave processes and each slave process is responsible for demanding work to be processed in its respective PU. Algorithm 1 shows the pseudo-code for the master process and Algorithm 2 shows the pseudo-code for the slave process. In such a setting, the total number of processes is then $np = (\sum_{j=1}^{c} g_j) + 1$. Each PU architecture has its own evaluation module, represented by the function `EvalSolutionsOnPU()` in Algorithm 2, which is an architecture specific implementation of the code that can be called by the slave processes as a kernel, offload or function call for GPU, MIC, and CPU, respectively. We present such implementations for GPUs and for CPUs/MICs in Section 4.

Algorithm 1 Exact hybrid parallel algorithm for minimal hitting set enumeration (master process)

Require: set X of variables, collection S, integer $k > 0$, number of candidates per task batch $b \cdot v$, stop constant *STOP*, number of slave processes n_s
Ensure: solution vector H where each subvector of H represents a subset $H \subseteq X$ such that $|H| \leq k, H \cap S \neq \emptyset, \forall S \in \mathcal{S}$, and no subset $G \subset H$ can satisfy the same conditions.

```
1: i ← 1
2: H ← ∅
3: SortS ← SortClauses(S)
4: BroadcastToSlaves(SortS)
5: while i ≤ k do
6:     u ← (|X| choose i)
7:     τ = ⌈u/(b · v)⌉
8:     sp ← 0
9:     for j = 0 to τ do
10:        sPid ← RecvAvailableSlave()
11:        SendStartPoint(sp, sPid)
12:        sp ← sp + (b · v)
13:    end for
14:    WaitForSlavesToFinish()
15:    H ← GatherSolutions()
16:    i = i + 1
17: end while
18: for j = 0 to n_s do
19:    sPid ← RecvAvailableSlave()
20:    SendStartPoint(STOP, sPid)
21: end for
    return H
```

For a given cardinality i, $\tau = \left\lceil \binom{|X|}{i} / (b \cdot v) \right\rceil$ *task batches* are created. For typical input sizes, the number of task batches τ is large, and these task batches can be dynamically assigned to the available PUs by the master process and executed as either a kernel, offload, or function call. The key factor is to transfer as little information as possible to each slave process so that they can start their processing immediately. The b and v variables are tuning parameters that are explained with more detail in Section 3.2.

Algorithm 2 Exact hybrid parallel algorithm for minimal hitting set enumeration (slave process)

Require: set X of variables, vector SortS, master process ID *mPid*, stop constant *STOP*.

Ensure: solution vector localH where each subvector of localH represents a subset $H \subseteq X$ with smallest cardinality, such that $|H| \leq k$, $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$, and no subset $G \subset H$ can satisfy the same conditions.

```
1: sPid ← GetProcessID()
2: localH ← ∅
3: w = |SortS|
4: loop
5:     SendAvailableSlave(sPid, mPid)
6:     sp ← RecvStartPoint()
7:     if sp = STOP then
8:         break
9:     end if
10:    SeeD ← NSGetSeeds(sp)
11:    EvalSolutionsOnPU(SeeD, SortS, localH, w, κ)
12: end loop
```

To control which instances will be processed by each slave process, the master process initializes a starting point variable *sp* as 0. When a slave requests new work, the master sends the current *sp* value and increases it by $b \cdot v$. After receiving *sp*, the slave process generates the list of candidates that will help the processing unit to generate its candidate solutions. Let X be the set of variables of an MHS enumeration instance. To generate this list of candidates in an efficient manner, we use a combinatorial numbering system (Equation (2)), which establishes a unique correspondence between a combination of elements of X of cardinality i, denoted by $C_i = \{c_i, c_{i-1}, \ldots, c_2, c_1\}$, and an integer N, $0 \leq N \leq \binom{|X|}{i}$.[28] In this way, the combinatorial numbering system, denoted by the function `NSGetSeeds()` in Algorithm 2, is used in the CPU by the slave process to generate only a very small number of candidate solutions, with $N = sp + (j \cdot \kappa)$, $0 \leq j < \beta$, $\kappa = \lfloor (b \cdot v)/\beta \rfloor$, where β is either the number of threads to be launched (in the case of CPU or MIC), or the number of GPU blocks to be launched (in the case of GPU). These combinations generated by the combinatorial numbering system are stored in a vector called SeeD and this vector is transmitted to the PU in the offload call, kernel call or function call. The other candidate solutions are generated in sequence for each thread in the PU during the offload call, kernel call or function call, having as starting point a unique combination present in the vector SeeD that was generated outside of the processing unit

$$N = \binom{c_i}{i} + \binom{c_i - 1}{i - 1} + \cdots + \binom{c_2}{2} + \binom{c_1}{1}, \quad c_i > c_{i-1} > \cdots > c_2 > c_1 \geq 0. \tag{2}$$

Figure 1 illustrates this distribution of task batches among PUs. Each slave process has a result vector localH where the results found by its respective device are stored. When the master detects that all of the τ tasks batches were assigned to a slave process, it means that all candidate solutions of the current cardinality i are being tested and then it waits (function `WaitForSlavesToFinish()` of the Algorithm 1) for all slave processes to complete its computations. Once the computations of all slave processes are completed, the master gathers the solutions found by them (function `GatherSolutions()` of the Algorithm 1). If no solution is found, it increases the cardinality by 1 and the entire process is repeated, until i reaches the value k.

From the implementation point of view, localH is implemented as a vector whose size is proportional to the number of candidate solutions evaluated per task batch (max of 20MB in our experiments, size is lower for small MHS sizes). Since the number of candidate solutions are

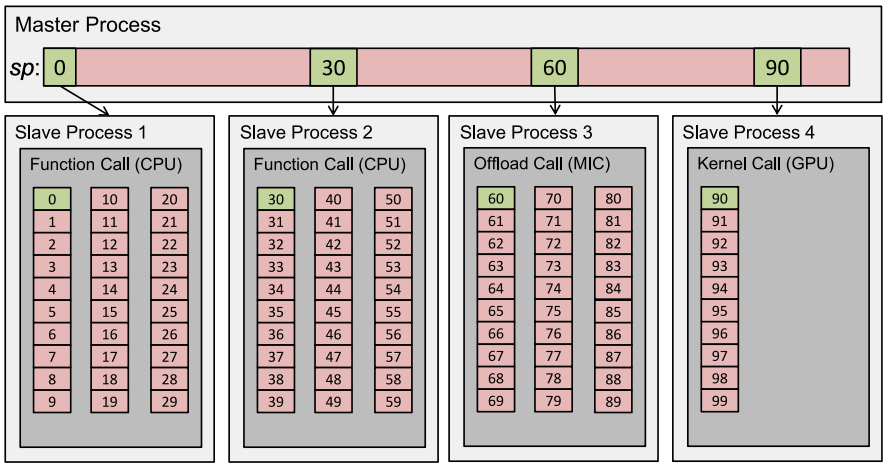

FIGURE 1 An example of the task batch distribution process among PUs, for 100 candidate solutions and task batch size $b \cdot v = 30$. Note that only information about four candidate solutions (the green ones) is sent through the network

computed per PU, it is possible to append the found solutions sequentially in localH of each PU. Hence, after gathering the solutions found by the devices, the master process does not need to go through the whole gathered result vector to retrieve the found solutions, it only needs to look at the appended sections of each PU. Furthermore, the way in which the found solutions are represented in localH can be different, depending on the quality of the network infrastructure:

1. A solution of any size i is represented as a single integer number, whose value is stored in localH. The master process is responsible to retrieve the underlying combination $(c_1, c_2, \dots, c_i)$ using Equation (2). This process can be easily done in parallel for each solution by the master process.
2. A solution of size i is returned to the master directly as a combination $(c_1, c_2, \dots, c_i)$.

As it can be seen, the approach 1 is more suited if the network bandwidth is not satisfactory, with a trade-off of more computation on the master process. Conversely, the approach 2 could be used, thus avoiding the onus of the master process to retrieve the solution's combination.

With this setting, we can observe the following advantages:

- We should note that most of the data required by the algorithm are generated in parallel by different processes and the only large data structure sent using the network is the SortS (see line 4 of the Algorithm 1) structure, which is sent once at the beginning of the algorithm. During the algorithm execution, only small values, such as *sp* values, are transferred. At the end of the execution, the solution list is transferred. Therefore all the remaining calculations, including the enumeration and evaluation of candidate solutions, are done in parallel by the multiple available PUs and MPI processes. Consequently, this algorithm should present a good scalability with increasing numbers of PUs and machines.
- During the iteration loops of the algorithm, only a small subset of candidate solutions are enumerated by each call of the function `EvalSolutionsOnPU()`, making the memory consumption of our algorithm not dependant on the number of candidate solutions, and thus making possible to enumerate MHSs of large cardinalities with no memory exhaustion.
- Having stored the current found solutions, the algorithm can be easily interrupted and restarted to the same point where it was interrupted only with three pieces of information: the set of clauses SortS, the current cardinality i, and the current starting point *sp*. Therefore, checkpointing of the algorithm is easily achievable.

3.2 | Load balancing

With heterogeneous configurations, we need to send task batch sizes according to the computational capabilities of the available PUs. To achieve such task, we use three tuning parameters b, v, and t. The parameter v defines the number of threads per GPU block and its value is statically set according to the GPU architecture of the platform. A typical practice to set this value is to use the number of GPU cores per streaming processor. In its turn, the variable b defines the number of GPU blocks to be created in each kernel call. We defined these b and v variables to address the thread configuration in the case of GPUs, which is significantly different from the other PUs. To address such a configuration for the case of CPUs and MICs, ie, to define the number of threads t to be created on the CPU or MIC, t is set according to the hyperthreading factor (threads per core) multiplied by the number of cores of the CPU or MIC. It is important to note that, since the number of candidate solutions to be evaluated per task batch is defined by the product $b \cdot v$, the number of candidate solutions per GPU thread is set to 1, and for CPUs and MICs is set to $\lfloor (b \cdot v)/t \rfloor$.

A simple though efficient heuristic strategy is to generate many small task batches, which are deployed to the PUs as they become available. In this case, faster PUs will process more task batches than slower ones. One way to obtain good performance and keep all the PUs busy is by setting the aforementioned parameter b with a certain minimum value (*bMin*). This value can be empirically determined by executing the whole algorithm a few times using the same input size, but with increasing values of b. For small values of b, the MPI communication and kernel/offload launch overheads will dominate, but as b increases, this overhead decreases, resulting in smaller execution times. We select *bMin* as the lowest b value for which the total execution time becomes constant with regard to b, which means that the overhead becomes negligible.

By performing this process on the fastest PU, we obtain a *bMin* value that can be used as the value of b by all PUs in the platform. It is important to note that this process needs to be performed only once per cluster, with only a few tests with increasing values of b and with a small instance. Once the *bMin* value is set, multiple executions in the same platform, using different inputs, can be performed using the same *bMin* value.

4 | PROCESSING UNITS EVALUATION MODULES

4.1 | CPU/MIC implementation

We now describe the main characteristics of the proposed exact CPU/MIC Minimal Hitting Set Enumeration algorithm. As presented in Section 3.1, the part of our algorithm that is assigned to the processing units is the checking of the candidate solutions, which in the case of CPUs and MICs, is executed in an offload call and a function call, respectively. Algorithm 3 shows the pseudo-code of the MIC version of the function `EvalSolutionsOnPU`.

Algorithm 3 Function EvalSolutionsOnPU (CPU/MIC version)

Require: Array SeeD, array SortS, solution vector localH, number of clauses w and number of candidates per thread κ

Ensure: solution vector localH where each subvetor of H represents the subset $H \subseteq X$ with smallest cardinality, such that $|H| \leq k$ and $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$.

```
1:  tr ← threadId
2:  for c = 1 to κ do
3:      if c = 1 then
4:          C^tr ← SeeD^tr
5:      else
6:          C^tr ← GetSubsequentComb(C^tr)
7:      end if
8:      solution ← true
9:      uniques ← ∅
10:     for j = 1 to w do
11:         if C^tr and SortS^j are disjoint sets then
12:             solution ← false
13:         end if
14:         intersect ← C^tr ∩ SortS^j
15:         if |intersect| = 1 then
16:             uniques ← uniques ∪ intersect
17:         end if
18:     end for
19:     if solution = true and uniques = C^tr then
20:         atomically add C^tr in localH
21:     end if
22: end for
```

Each thread tr is responsible to evaluate κ candidate solutions. The first candidate solution to be evaluated by each thread is present in the vector SeeD which sent to the MIC or CPU. This candidate solution is assigned to the variable C^{tr}. To confirm if C^{tr} is an MHS, it is necessary to check (i) if C^{tr} is a hitting set and (ii) if no other subset of C^{tr} is a hitting set as well. To ensure the former condition, we verify if C^{tr} is not disjoint with all of the clauses present in the vector SortS. For the latter condition, we make use of the following statements.[24]

Definition 1. For a candidate solution $C \subseteq X$, a variable $c \in C$ is *critical* in C if there exists at least one clause $S \in \mathcal{S}$ which contains c, but no other variables of C.

Proposition 2. *Let C be a hitting set of $\mathcal{S}$. C is a minimal hitting set if and only if every $c \in C$ is critical in C.*

The Proposition 2 can be verified by keeping track of all critical variables of C^{tr}, represented by the set *uniques* (line 9 and lines 15-17 of Algorithm 3), during the clauses evaluation. At the end of this evaluation, the aforementioned proposition holds true for C^{tr} if the *uniques* set is equal to C^{tr}.

If conditions (i) and (ii) hold true, C^{tr} is atomically added to a local solution vector localH. At the implementation level, atomic instructions are only used to address the thread race condition when computing the number of solutions found in a task batch. Alternatively, these atomic instructions could be replaced by a reduction operation. Once a candidate solution is evaluated, the respective thread generates the subsequent candidate solution to be evaluated, using the previous candidate solution as base.

4.1.1 | MIC vectorization

One characteristic of the Xeon Phi is the 512-bit vector processors present in each core of the Xeon Phi co-processors, which constitutes a huge advantage.[29] In this regard, the vectorization of the algorithm can be done automatically by the Intel compiler; hence, it is crucial that the most process demanding parts of the algorithm are implemented in a way that the vectorization can be performed by the compiler. As previously mentioned, the checking of the candidate solutions is performed through a disjunction check with the clauses of the input instance. Since this disjunction check is the most computationally expensive task, we need to make sure that the implementation of this disjunction check exploits the 512-bit vector units present in the MIC. One can notice that we could avoid the execution of the entire loop of the lines 10-18 of the Algorithm 3 once a candidate solution is tagged as *false*. However, we implemented this loop in a way that avoids unnecessary conditional deviations and early loop breaks, which would prevent vectorization. Although it would seem that executing this loop entirely for every candidate solution is a performance loss, the high vectorization capabilities of the Xeon Phi outperform the absence of the early loop break.

4.2 | GPU implementation

For the GPU implementation, we made use of the CUDA[30] programming model. In this model, the GPU is organized by a grid of thread blocks and each thread block can be executed concurrently by the GPU. Algorithm 4 shows the pseudo-code of our GPU implementation of the kernel call `EvalSolutionsOnPU()`. Since the GPU has a much larger number of cores when compared to the CPU or MIC, we designed our algorithm in a way that each GPU thread evaluates only one candidate solution. The vector SeeD is transmitted to the GPU with information for each thread block to generate its candidate solutions. For a GPU thread block, at the beginning of the algorithm the first thread of the block generates all candidate solutions that will be evaluated by the threads of the block and stores them in a vector SharedC in the shared memory, which is a very fast memory that is shared by the threads in a thread block. When all candidate solutions of a thread block are generated, each thread begins

evaluating its corresponding candidate solution, which is performed in the same way as in the CPU/MIC implementation described in Section 4.1. If a solution is found by a thread, the thread atomically adds this solution to a solution vector localH and terminates its execution. Otherwise, the thread just terminates its execution. In the GPU version, replacing the atomic instructions by a reduction operation would naturally require synchronizations after the evaluation of the candidate solutions, and these synchronizations would prevent the early termination of the GPU threads. Therefore, in contrast with the CPU/MIC version (see Section 4.1), it is likely that a reduction operation would provide less performance than atomic instructions.

Although it seems inefficient to enumerate the candidate solutions of a thread block by using just the first thread of this block, this enables us to enumerate candidate solutions sequentially and in lexicographical order, which informally means that the difference of adjacent solution candidates is likely to be just one variable. This lexicographical order is not only computationally easy to be performed, but also helps the threads in the GPU to be executed in a SIMD (Single Instruction-Multiple Data) way as much as possible, which is an essential property to run GPU algorithms efficiently.

Algorithm 4 Function EvalSolutionsOnPU (GPU version)

Require: Array SeeD, array SortS, solution vector localH and number of clauses w

Ensure: solution vector H whose each subvector of H represents the subset $H \subseteq X$, such that $|H| \leq k$ and $H \cap S \neq \emptyset, \forall S \in \mathcal{S}$.

```
1: if threadLocalId = 0 then
2:     r ← blockId
3:     v ← blockSize
4:     SharedC^0 ← SeeD^r
5:     for t = 1 to v − 1 do
6:         SharedC^t ← GetSubsequentComb(SharedC^{t−1})
7:     end for
8: end if
9: SynchronizeThreads()
10: t ← threadLocalId
11: solution ← true
12: uniques ← ∅
13: for j = 1 to w do
14:     if SharedC^t and SortS^j are disjoint sets then
15:         solution ← false
16:         break
17:     end if
18:     intersect ← SharedC^t ∩ SortS^j
19:     if |intersect| = 1 then
20:         uniques ← uniques ∪ intersect
21:     end if
22: end for
23: if solution = true and uniques = SharedC^t then
24:     atomically add SharedC^t in localH
25: end if
```

4.3 | Few notes on other parallelization approaches

In this work, we chose to perform the parallelization by the minimal hitting set candidates. This choice was made due to the fact that each solution candidate can be evaluated concurrently. One can envision, however, to parallelize by the clauses and not by the solution candidates. In this setting, the clauses would be partitioned by the threads and each thread would evaluate all solution candidates against its subset of clauses.

Indeed, this setting could help for a better vectorization of the disjunction check of the clauses, specially for large clauses and large vector processors (notably the MIC). However, threads would no longer be independent when evaluating a MHS candidate solution, since the evaluation is shared among the threads. Once a candidate solution is tagged as false by some thread, it should inform all other threads to discard this candidate, requiring extra communication and synchronization among threads. Moreover, the vast majority of candidates are not solutions and their costly synchronization would the norm, reducing overall performance.

5 | EXPERIMENTAL RESULTS

Here, we present the experimental results obtained with our hybrid MHS enumeration algorithm. We performed two experiments, the *Small Scale* experiment and the *Large Scale* experiment. In the Small Scale experiment we evaluated the performance and scalability of our algorithm in a small, highly heterogeneous cluster and with synthetic, randomly generated datasets. In the Large Scale experiment, we evaluated the performance of our algorithm in a large CPU-GPU cluster, and with a dataset derived from real-world data. In all experiments, the compilers used are Intel (`icc`) 16.0.1, except for the GPU parts, for which we used the NVIDIA compiler (`nvcc`) version 7.5. We also adopted Intel MPI version 5.1.2 for multi-node communication and conjunction of the processing modules. The optimization parameter `-O3` was used in all compilation steps.

5.1 | Small scale experiment

In this experiment we used a two-node heterogeneous cluster described in Table 1. We generated synthetic instances with $|X| = 8192$ (number of variables), collection of clauses $\mathcal{S}$ with $|\mathcal{S}| = 1024$ and $k = 3$. These instances were randomly generated using the `rand()` function of the

TABLE 1 Cluster description for the small scale experiment

Number of Nodes	Processor	Accelerator	Acc. No. Cores
1	2xXeon E5-2620v2 2.1GHz six-core	Xeon Phi 3120A	57
1	Core i7-5930 K 3.50GHz six-core	GTX Titan X	3072

C standard library. For each instance, we generated random clauses with sizes in the interval $[\lceil min\_range * |X| \rceil, |X|]$, with $min\_range = 0.6$. This *min_range* value was empirically determined to permit the generation of instances with valid solutions. Instances with no solutions were discarded. In all experiments, we start to measure the total time at the moment the algorithm receives the instance and finish when the results are returned, which accounts the lines 1-21 of Algorithm 1.

5.1.1 | Load balance tuning

To evaluate the algorithm for CPU-GPU-MIC platforms, the first step is to determine the minimum task batch size that permitted the application to execute with acceptable overhead. As discussed in Section 3.2, smaller task batch sizes permit the generation of more task batches and a better load-distribution with heterogeneous platforms. Figure 2A shows the execution time of a single task batch in a GTX TITAN X. The execution time increases linearly with task batch size, controlled by the parameter *b* (remembering that *b* is also the number of GPU blocks to be created in the kernel call). This indicates that moderate numbers of task batch size are sufficient to fully occupy the accelerator device. However, Figure 2B shows that only for *b* values above 20000 the total execution time becomes close to constant, indicating that below this value the overhead of MPI process communication and task batch launches becomes relevant. Hence, we set the parameter $bMin = 20000$ for the heterogeneous configuration of the experiments. We set the *v* parameter as 128, which is the number of cores per streaming processor of the GTX TITAN X.[31] Therefore, the number of solution candidates to be evaluated per each task batch that will be processed in the PUs is $bMin \cdot v = 20000 * 128 = 2560000$.

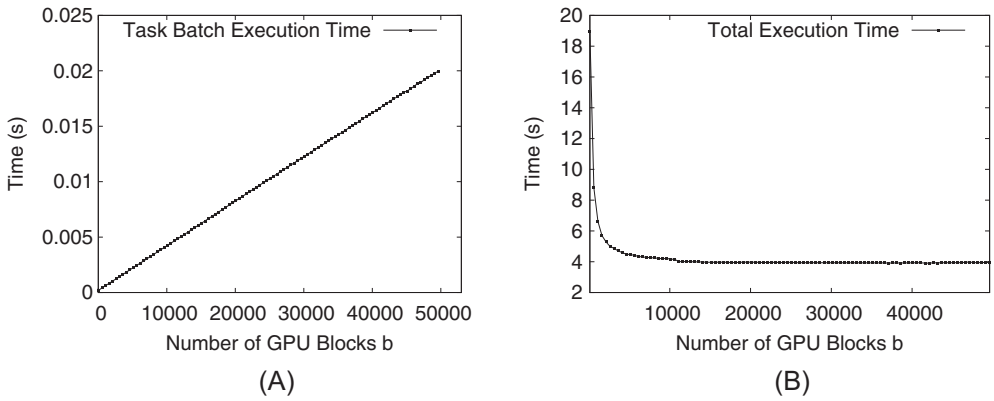

FIGURE 2 Task batch execution times and total execution times of our proposed algorithm in function of the number of GPU blocks parameter *b*. A, Task batch execution time; B, Total execution time

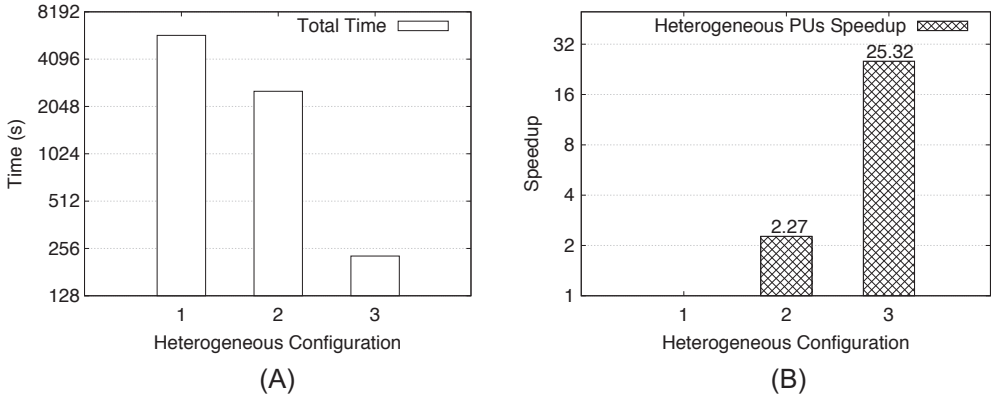

FIGURE 3 Execution time and consequent speedups of proposed hybrid algorithm in function of the number of processing units. A, Execution time (log_2 scale on y axis); B, Speedup (log_2 scale on y axis)

5.1.2 | Small scale performance

We evaluated the scalability of our hybrid MHS enumeration algorithm using a heterogeneous cluster composed of CPUs, GPUs, and MICs. Figure 3A shows the average execution time of ten successive executions as a function of the heterogeneous configurations shown in Table 2. We see a noticeable improvement in the total execution time, from 1.6 hours to only 3.8 minutes. In this regard, we increased the number of variables $|X|$ from 8192 to 32768 and we were able to enumerate all MHSs of this instance in only 4.1 hours with the four PUs (configuration 3 of Table 2, result not shown in Figure 3A). With only two CPUs, the execution time would take about 25.32 times longer, or 103 hours.

Figure 3B shows the obtained speedups, measured as the ratios between the execution time of the algorithm using a configuration shown in Table 2 and the execution time of the algorithm using the dual socket CPUs (configuration 1 of Table 2). With the addition of the Xeon Phi 3120A, the speedup jumped to 2.27, which shows that the Xeon Phi 3120A not only outperformed two Xeon E5-2690 CPUs but also shows that the performance of our algorithm can at least double by adding only one Xeon Phi device, therefore attesting the HPC capabilities of the Xeon Phi in our algorithm. Another result that cannot be overlooked is the price and the performance delivered by the processing units. In a recent price survey (May 2017), two Xeon E5-2690 CPUs would cost around 4000 USD, while one Xeon Phi 3120A would cost around 1500 USD, showing that the Xeon Phi can be in fact an appealing alternative for high performance computing.

In its turn, with the addition of a GTX TITAN X, the speedup has further increased to 25.32. This indicates that the GTX TITAN X had better performance than the Xeon Phi 3120A when processing the task batches. Although the cores of the Xeon Phi 3120A are more complex than those of the GTX TITAN X, the massive number of cores of the GTX TITAN X outperforms the Xeon Phi 3120A in our hybrid MHS enumeration implementation. Since we assign one GPU thread per candidate solution, the GTX TITAN X can concurrently evaluate 3072 candidate solutions. With the Xeon Phi 3120A, however, with four threads per core, there are only 224 concurrent threads. Therefore, we can notice an advantage for the GPU in our hybrid implementation. The price and performance delivered of the GTX TITAN X is also the best of the three configurations we evaluated. In a recent price survey (May 2017), one GTX TITAN X would cost around 2000 USD and it delivered an order of magnitude of performance increase in our algorithm. However, it is important to note that the Xeon Phi 3120A is from the Knights Corner architecture, which is the first non-prototype architecture of Xeon Phi product line. Hence, the GPU *versus* MIC performance difference shown here might be lower with the current Xeon Phi architectures. In addition to the speedup, one may formulate a device efficiency metric, measured as the ratio of the speedup and the number of devices used. Following the speedup values of Figure 3B, adding the Xeon Phi 3120A provided an efficiency of 1.13, taking the 2x Xeon E5-2690 CPUs as a baseline. In its turn, adding the GTX TITAN X provided a combined efficiency of 8.44.

To better understand these results, we monitored the number of task batches assigned to each processing unit (Figure 4A). We can see that with three processing units, the Xeon Phi 3120A received 51% of the tasks, and with four processing units, the Xeon Phi 3120A and GTX TITAN X received 4.3% and 91.4% of the tasks, respectively. We also monitored the time that each PU spent with the algorithm. Figure 4B shows the time spent for the four PUs used. As can be seen, with task batch sizes small enough, all PUs have a tendency of finishing its work at the same time, which is a good load balance measure. This shows that our hybrid algorithm can in fact handle heterogeneous CPU-GPU-MIC platforms, adequately balancing the load among the tasks in function of each processing unit's performance.

TABLE 2 Heterogeneous configurations used in the Small Scale experiment

Configuration	Description
1	2xXeon E5-2690 (dual socket configuration)
2	2xXeon E5-2690 and 1xXeon Phi 3120A (single node)
3	2xXeon E5-2690, 1xXeon Phi 3120A and 1xGTX TITAN X

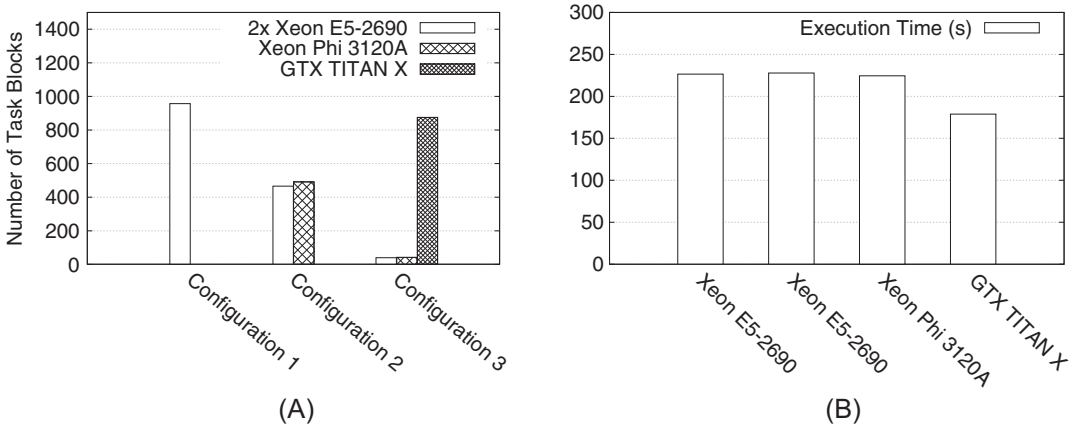

FIGURE 4 Load balance evaluation results of our hybrid exact MHS enumeration algorithm. A, Number of task batches assigned to each PU; B, Total time that each processing unit has spent enumerating MHSs

5.2 | Large scale experiment

This experiment was carried out using the Grid'5000 [32] testbed. We used six nodes of the Grele CPU-GPU cluster (Table 3), with a total of twelve CPUs and twelve GPUs. We also made use of the `HER2+ SHORT`[14] dataset. In this dataset, the clauses represent the shortest intervention pathways of the HER2+ cell signaling network. The minimal hitting sets of this dataset represent optimal combinations of interventions to this signaling network, which are important to better understand and control it. This dataset is constituted by $|X| = 123$ variables and $|S| = 534$ clauses. For this cluster, we assigned $bMin = 50000$. Although the number of variables and clauses does not seem big, the combinatorial nature of the problem makes the MHS enumeration for large cardinalities k very difficult. Indeed, to the best of our knowledge, none of the existing parallel MHS generation algorithms of the literature were capable of enumerating the minimal hitting sets for `HER2+ SHORT` with $k = 10$. The main focus of this experiment is therefore to find all MHS for this configuration.

Table 4 shows the processing time of our MHS enumeration algorithm for the `HER2+ SHORT` dataset for cardinalities $k \in \{5, 7, 10\}$. For cardinalities 5 and 7, our algorithm kept reasonable performances when compared to other algorithms[20] of the literature. The noteworthy result is for $k = 10$. With 24 processing units (12 CPUs and 12 GPUs), we managed to enumerate all MHS up to $k = 10$ in parallel and in 7.7 hours with no memory exhaustion. This result is expected. As mentioned in Section 3.1, only a fraction of the total number of candidate solutions are enumerated in the processing units per task batch, and thus enabling the evaluation of all candidate solutions without exhausting memory.

Figure 5A shows the load balance distribution of task batches among the processing units. Similar to the Small Scale experiment, the GPU presented better performance when compared to the CPU, as our load balancing algorithm tends to assign more task batches to the best performing processing unit and each the GeForce GTX 1080 Ti GPU received an average of 7.5% of task batches, in comparison with only 1.7% in average, for each of the dual-socket Xeon E5-2650 CPU.

We also registered how the minimum hitting sets are spread in the search space. Figure 5B shows the percentage of solutions found by our algorithm in function of the percentage of the search space in lexicographic order. This result shows that 50% of the solutions are concentrated at the first 30% of the search space. This indicates that, if all MHSs are not necessary, one could search in only a small fraction of the total search space and will still be able to find the majority of MHSs. Additionally, although there is a slight peak of solutions at the beginning of the search space, plateaus are uncommon in the graph shown in Figure 5B, indicating that there is no large region of the search space where solutions are nonexistent.

TABLE 3 Cluster description for the Large Scale experiment

Number of Nodes	Processor	Accelerator	Acc. No. Cores
6	2xXeon E5-2650 (dual socket) v4 2.9GHz twelve-core	2xGeForce GTX 1080 Ti	3584

TABLE 4 Performance results of our MHS generation algorithm for the `HER2+ SHORT` dataset

Cardinality k	Processing Time (s)
5	1.87
7	11.7
10	27930 (7.7 hours)

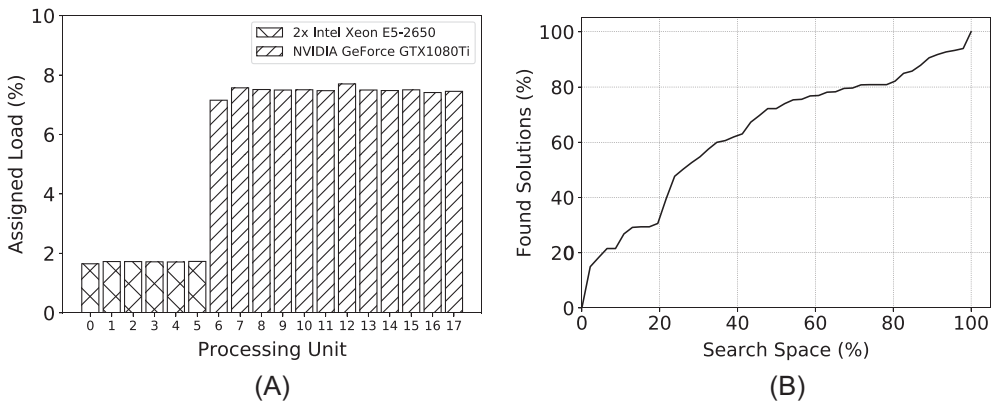

FIGURE 5 Summary of the minimal hitting set enumeration for the HER2+ dataset. A, Load balance distribution among PUs; B, Cumulative number of solutions found over time

6 | CONCLUSIONS

In this paper, we presented an exact hybrid CPU-GPU-MIC algorithm for the Minimal Hitting Set (MHS) Enumeration Problem suited for large input size applications. We enumerated MHSs for problem instances on the order of thousands of elements with reasonable time, achieving good performance and scalability with regard to execution time.

We performed several experiments on a heterogeneous cluster composed of CPUs Intel Xeon E5-2620v2, a MIC Intel Xeon Phi 3120A and a GTX TITAN X GPU. The results show that, in our algorithm, each of the accelerator devices used has resulted in a satisfactory performance increase even for early development accelerator devices such as the Xeon Phi 3120A. Moreover, the results also show that our hybrid algorithm has good performance and scalability by correctly distributing the tasks among the processing units according to their computational efficiency in processing the task batches. This enabled speedups of up to 25.32, when using all processing units instead of two six-core CPUs. With our exhaustive search approach, since the evaluation of the MHS candidate solutions are done in a concurrent manner, devices with a higher level of parallelism were capable of providing better performance. Although the Xeon Phi tries to provide parallelization by means of vectorization, in our experiments, fifty-seven 512-bit vector processors of the Xeon Phi were not enough to outperform the GPU, where more than three thousand of threads can run in parallel. We could envision an increase in performance of the MIC if a more recent Xeon Phi device were used, though not with substantial changes in the results that we have found. Nevertheless, with this cluster, we enumerated MHSs for instances in order of tens of thousands of variables in less than 5 hours, which characterizes a formidable result in terms of input size resolution capability.

When adopting an optimized exhaustive search over candidate solutions approach, we trade higher computational processing requirements for lower memory usage and improved parallelism. This trade-off can be advantageous in several cases. For instance, we enumerated MHSs for the `HER2+ SHORT` cell signaling network, whose MHSs are of special interest to better understand and control this network, with MHS sizes up to 10. We were capable of effectively making use of a large CPU-GPU cluster, with a total of 12 CPUs and 12 GPUs, to enumerate such MHSs in less than 8 hours and being, to the best of our knowledge, the first parallel MHS enumeration algorithm capable of enumerating such MHSs without memory exhaustion.

All of these improvements also reinforce the statement that the adoption of hybrid CPU-GPU-MIC algorithms can characterize a performance improvement if each architecture is adequately exploited, the load balance is properly set and the communication cost is minimized. However, it is important to note that the execution time of our hybrid exact MHS enumeration algorithm can still be unfeasible if the solutions present very high cardinalities. This problem can only be attenuated by adding more processing units to evaluate the solution candidates.

For future works, we plan to further optimize our algorithm to increase the efficiency of enumerating minimal hitting sets with very high cardinalities, as well to provide a performance model to estimate whether our algorithm takes full advantage of the hybrid platform. Moreover, we also plan to provide a full software package for enumerating MHSs, with a friendly interface and an auto-tuning procedure, capable to automatically discover the best tuning parameters for a specific cluster.

ACKNOWLEDGMENTS

This research was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and it is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, 559955/2010-3, 302620/2014-1, CAPES proc. 88887.136422/2017-00, and FAPESP proc. 14/50937-1, 15/24485-9, 13/26644-1, 14/50937-1, 15/01587-0.

The authors also would like to thank UFABC, all institutions (INRIA, CNRS, RENATER, and several Universities as well as other organizations) who support the Grid5000 platform, and Andrew Gainer-Dewar and Paola Vera-Licona for making their datasets publicly available.

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

FINANCIAL DISCLOSURE

None reported.

AUTHOR CONTRIBUTIONS

All authors devised the study and idealized the method. D. C. S. and R. Y. C. planned the implementation of the method and the experiments. D. C. S. implemented the method and performed the experiments. D. C. S. and R. Y. C. wrote the paper. All authors analyzed the results, read, edited and approved the manuscript.

ORCID


Danilo Carastan-Santos https://orcid.org/0000-0002-1878-8137

REFERENCES

1. Kolman P, Walen T. Reversal distance for strings with duplicates: linear time approximation using hitting set. *Electron J Comb*. 2007;14(1):11.
2. Hvidsten TR, Lægreid A, Komorowski J. Learning rule-based models of biological process from gene expression time profiles using gene ontology. *Bioinformatics*. 2003;19(9):1116-1123.
3. Pearson WR, Robins G, Wrege DE, Zhang T. On the primer selection problem in polymerase chain reaction experiments. *Discrete Appl Math*. 1996;71(1-3):231-246.
4. Ideker TE, Thorsson V, Karp RM. Discovery of regulatory interactions through perturbation: inference and experimental design. In: Proceedings of the Pacific Symposium on Biocomputing; 2000; Honolulu, HI.
5. Ruchkys DP, Song SW. A parallel solution to infer genetic network architectures in gene expression analysis. *Int J High Perform Comput Appl*. 2003;17(2):163-172.
6. Carastan-Santos D, de Camargo RY, Martins Jr DC, Song SW, Rozante LCS, Borelli FF. A multi-GPU hitting set algorithm for GRNs inference. Paper presented at: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing; 2015; Shenzhen, China.
7. Carastan-Santos D, de Camargo RY, Martins Jr DC, Song SW, Rozante LCS. Finding exact hitting set solutions for systems biology applications using heterogeneous GPU clusters. *Futur Gener Comput Syst*. 2017;67:418-429.
8. Carastan-Santos D, Martins Jr DC, Rozante LCS, Song SW, de Camargo RY. A Hybrid CPU-GPU-MIC algorithm for the hitting set problem. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD*. 2017;18(1):196-207.
9. Duran A, Klemm M. The Intel® many integrated core architecture. Paper presented at: International Conference on High Performance Computing & Simulation (HPCS); 2012; Madrid, Spain.
10. Andrade G, Ferreira R, Teodoro G, Rocha L, Saltz JH, Kurc T. Efficient execution of microscopy image analysis on CPU, GPU, and MIC equipped cluster systems. Paper presented at: 26th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD); 2014; Paris, France.
11. Wolfe N, Liu T, Carothers C, Xu XG. Heterogeneous concurrent execution of Monte Carlo photon transport on CPU, GPU and MIC. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms; 2014; New Orleans, LA.
12. Reza H, Aguilar M, Jalal SF. Regression testing of GPU/MIC systems for HPCC. In: Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science; 2015; Florence, Italy.
13. Sîrbu A, Babaoglu O. Power consumption modeling and prediction in a hybrid CPU-GPU-MIC supercomputer. In: *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. Cham, Switzerland: Springer International Publishing; 2016:117-130.
14. Vera-Licona P, Zinovyev A, Bonnet E, et al. A pathway-based design of rational combination therapies for cancer. *Eur J Cancer*. 2012;48:S154.
15. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman and Company; 1999.
16. Shi L, Cai X. An exact fast algorithm for minimum hitting set. Paper presented at: Third International Joint Conference on Computational Science and Optimization; 2010; Huangshan, China.
17. Ćendić-Lazović B. A genetic algorithm for the minimum hitting set. *Sci Publ State Univ Novi Pazar Ser A Appl Math Inform Mech*. 2014;6(2):107-117.
18. Hochbaum DS. *Aproximation Algorithms for NP-Hard Problems*. Boston, MA: PWS Publishing Company; 1997.
19. Steinbach B, Posthoff C. Sources and obstacles for parallelization—a comprehensive exploration of the unate covering problem using both CPU and GPU. In: *GPU Computing With Applications in Digital Logic*. Tampere, Finland: Tampere University of Technology; 2012:63-95.
20. Gainer-Dewar A, Vera-Licona P. The minimal hitting set generation problem: algorithms and computation. *SIAM J Discrete Math*. 2017;31(1):63-100.
21. Berge C. *Hypergraphs: Combinatorics of Finite Sets*. Amsterdam, The Netherlands: Elsevier; 1984.
22. Greiner R, Smith BA, Wilkerson RW. A correction to the algorithm in Reiter's theory of diagnosis. *Artif Intell*. 1989;41(1):79-88.
23. Kavvadias DJ, Stavropoulos EC. An efficient algorithm for the transversal hypergraph generation. *J Graph Algorithms Appl*. 2005;9(2):239-264.
24. Murakami K, Uno T. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Appl Math*. 2014;170:83-94.
25. Elbassioni K, Hagen M, Rauf I. A lower bound for the HBC transversal hypergraph generation. *Fundam Informaticae*. 2014;130(4):409-414.
26. Cardoso N, Abreu R. MHS^2: a map-reduce heuristic-driven minimal hitting set search algorithm. In: *Multicore Software Engineering, Performance, and Tools: International Conference, MUSEPAT 2013, St. Petersburg, Russia, August 19-20, 2013. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013:25-36.
27. Toda T. Hypergraph transversal computation with binary decision diagrams. In: *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013;91-102.
28. Knuth DE. *The Art of Computer Programming Fascicle 3: Generating All Combinations and Partitions*. Boston, MA: Addison-Wesley; 2005.
29. Jeffers J, Reinders J. *Intel Xeon Phi Coprocessor High-Performance Programming*. Waltham, MA: Elsevier; 2013.
30. NVIDIA. CUDA C programming guide. 2014.
31. NVIDIA. Maxwell: the most advanced CUDA GPU ever made. 2016. https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/. Accessed July 18, 2016.
32. Balouek D, Amarie AC, Charrier G, et al. Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov II, van Sinderen M, Leymann F, Shan T, eds. *Cloud Computing and Services Science: Second International Conference, CLOSER 2012, Porto, Portugal, April 18-21, 2012. Revised Selected Papers*. Cham, Switzerland: Springer International Publishing; 2013:3-20.

How to cite this article: Carastan-Santos D, Martins-Jr DC, Song SW, Rozante LCS, de Camargo RY. A hybrid CPU-GPU-MIC algorithm for minimal hitting set enumeration. *Concurrency Computat Pract Exper*. 2018;e5087. https://doi.org/10.1002/cpe.5087