A Hybrid CPU-GPU-MIC Algorithm for the Hitting Set Problem

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

[1]Centro de Matemática, Computação e Cognição
Universidade Federal do ABC (UFABC)
Santo André, Brazil

[2]Instituto de Matemática e Estatística
Universidade de São Paulo (USP)
São Paulo, Brazil

{danilo.santos, david.martins, luiz.rozante}@ufabc.edu.br

song@ime.usp.br, raphael.camargo@ufabc.edu.br

Abstract. *We present a hybrid exact algorithm for the Hitting Set Problem (HSP) 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 solve large HSP 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 exact HSP solutions for instances with tens of thousands of variables in less than 5 hours. These results reinforce the statement that heterogeneous clusters of CPUs, GPUs and MICs can be used efficiently for high-performance computing.*

1. Introduction

Many important theoretical or practical problems can be modeled, in part or as a whole, as an instance of the Hitting Set problem (HSP) whose main objective, informally, is to find a minimum set of variables 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 [Kolman and Walen 2007], classification models [Hvidsten et al. 2003], polymerase chain reaction experiments [Pearson et al. 1996] and gene regulatory networks (GRN) inference [Ideker et al. 2000, Ruchkys and Song 2003]. A common characteristic of these problems is that the input size of the modeled HSP instances is often large, making the retrieval of the exact solutions impracticable for traditional algorithms.

Since the Hitting Set problem is NP-hard [Garey and Johnson 1999], there exist some solutions that try to circumvent this problem, either by imposing restriction on the problem, such as with non-polynomial exact algorithms [Shi and Cai 2010], or by using heuristics and approximations [Cendic 2014, Hochbaum 1997]. Most recent works in HSP algorithms focus only on multithread parallelism, with different approaches to tackle HSP such as hypergraphs [Murakami and Uno 2014] and MapReduce [Cardoso and Abreu 2013]. Gainer-

Dewar and Vera-Licona [Gainer-Dewar and Vera-Licona 2017] present an extensive review of recent HSP algorithms. Due to the absence of accelerator-aided HSP algorithms, we have previously proposed a highly parallel and efficient algorithm for GPU (Graphics Processing Unit) [Owens et al. 2008] clusters that finds exact solutions for HSP instances with thousands of variables [Carastan-Santos et al. 2015, Carastan-Santos et al. 2017]. In these previous works, however, we explored only a single architecture for finding exact HSP solutions.

With the advent of new accelerator technologies such as Many Integrated Core (MIC) architectures [Duran and Klemm 2012], 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, that is, applications capable of using CPU, GPU and MIC processors together [Andrade et al. 2014, Wolfe et al. 2014, Reza et al. 2015, Sîrbu and Babaoglu 2016]. 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 this regard, in this paper we explore ways to circumvent these hindrances in the scope of the Hitting Set Problem, and thus we present the following contributions:

- We propose a hybrid exact Hitting Set 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 HSP implementation. The experimental results show that if (i) the load balancing is properly set, (ii) each individual architecture advantage is exploited and (iii) the node to node communication cost is minimized, we can effectively use heterogeneous CPU-GPU-MIC clusters, which allow us to solve unprecedentedly large HSP instances with tens of thousands of variables in reasonable time.

The remaining parts of this manuscript are organized as follows: Section 2 presents a formal definition of HSP, followed by our exact hybrid algorithm design, including the load balancing procedure. In Section 3 our proposed MIC HSP implementation is presented. The experimental results showing the performance evaluation are presented and discussed in Section 4 and in Section 5 we present the main conclusions.

2. Hybrid CPU-GPU-MIC Hitting Set Algorithm

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 $H \subseteq X$ with the smallest cardinality 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 $\mathcal{H} = \{H_1, H_2, ..., H_{|\mathcal{H}|}\}$ the collection of possible solutions. In this work our goal is to obtain all possible solutions $H \in \mathcal{H}$.

2.1. Exact Hybrid HSP Algorithm

We adopted an enhanced exhaustive search algorithm [Carastan-Santos et al. 2017] 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. 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 the 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 processing 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. For CPUs and GPUs we adopted the exact HSP algorithm developed by Carastan-Santos *et. al* [Carastan-Santos et al. 2017]. For MICs we adopted the exact MIC HSP algorithm presented in Section 3.

Algorithm 1 Exact Hybrid Parallel Algorithm for Hitting Set Problem (Master Process).

Input: 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

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

1: $i \leftarrow 1$
2: H $\leftarrow \emptyset$
3: SortS $\leftarrow$ `SortClauses(`$\mathcal{S}$`)`
4: `BroadcastToSlaves(`SortS`)`
5: **while** $i \leq k$ **do**
6: $u \leftarrow \binom{|X|}{i}$
7: $\tau = \lceil u/(b \cdot v) \rceil$
8: $sp \leftarrow 0$
9: **for** $j = 0$ to τ **do**
10: $sPid \leftarrow$ `RecvAvailableSlave()`
11: `SendStartPoint(`sp`,` $sPid$`)`
12: $sp \leftarrow sp + (b \cdot v)$
13: **end for**
14: `WaitForSlavesToFinish()`
15: H $\leftarrow$ `GatherSolutions()`
16: **if** H is not empty **then**
17: **break**
18: **end if**
19: $i = i + 1$
20: **end while**
21: **for** $j = 0$ to n_s **do**
22: $sPid \leftarrow$ `RecvAvailableSlave()`
23: `SendStartPoint(`$STOP$`,` $sPid$`)`
24: **end for**
25: **return** H

Algorithm 2 Exact Hybrid Parallel Algorithm for Hitting Set Problem (Slave Process).

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

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

```
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
```

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 high, 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 2.2.

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 instance of HSP. 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 $C_i = \{c_i, c_{i-1}, ..., c_2, c_1\}$, and an integer N, $0 \leq N \leq \binom{|X|}{i}$ [Knuth 2005]. 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.

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

Figure 1 illustrates this distribution of task batches among PUs. Found solutions

are stored on local solution vectors localH by each slave process. 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.

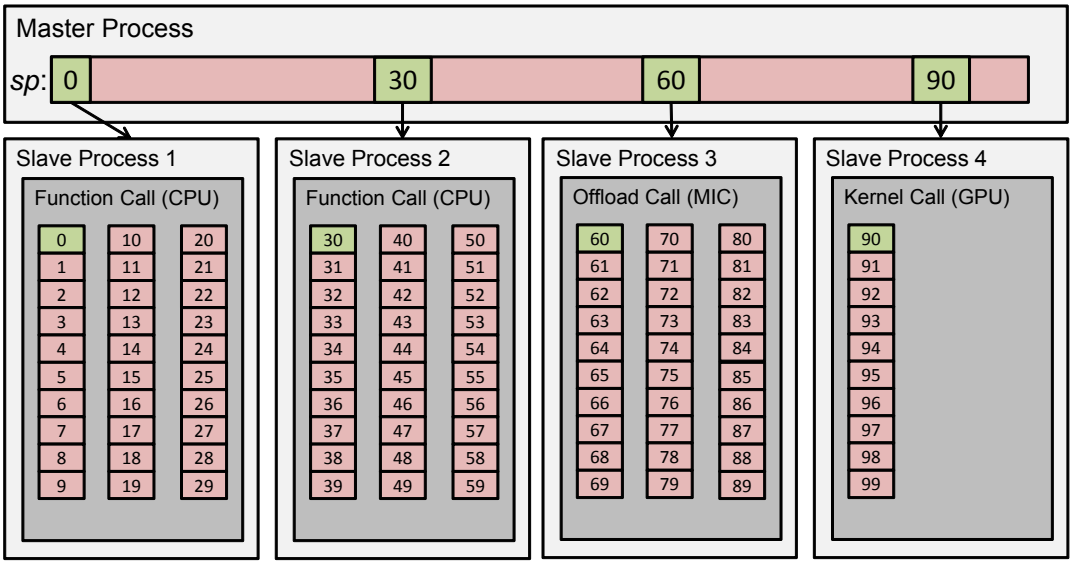


Figure 1. An example of the task batch distribution process among PUs, for 100 candidate solutions and task batch size $b \cdot v = 30$. Note that only information about four candidate solutions (the green ones) is sent through the network.

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.

2.2. 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, i.e., 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's 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 HSP instance. Once the $bMin$ value is set, multiple executions in the same platform, using different inputs, can be performed using the same $bMin$ value.

3. Exact MIC HSP Algorithm

We now describe the main characteristics of the proposed exact MIC HSP algorithm. As presented in Section 2.1, the part of our algorithm that is assigned to the processing units is the checking of the candidate solutions, which in the case of MICs, is executed in an offload call. Algorithm 3 shows the pseudo-code of the MIC version of the function `EvalSolutionsOnPU`.

Algorithm 3 Offload function EvalSolutionsOnPU (MIC version)

Input: Array SeeD, array SortS, solution vector localH, number of clauses w and number of candidates per MIC thread κ
Output: 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:    for j = 1 to w do
10:     if C^tr and SortS^j are disjoint sets then
11:       solution ← false
12:     end if
13:   end for
14:   if solution = true then
15:     atomically add C^tr in localH
16:   end if
17: end for
```

Table 1. Cluster Configuration

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

Each thread tr of the MIC is responsible to evaluate κ candidate solutions. The first candidate solution to be evaluated by each thread is present in the vector SeeD that is sent to the MIC during offload. This candidate solution is assigned to the variable C^{tr} and the next steps is to check if C^{tr} is not disjoint with all of the clauses present in the vector SortS. If such condition holds true, C^{tr} is atomically added to a MIC local solution vector localH. Once a candidate solution is evaluated, the respective thread of the MIC generates the subsequent candidate solution to be evaluated, using the previous candidate solution as base.

One characteristic of the Xeon Phi is the 512-bit vector processors that are present in each core of the Xeon Phi co-processors, which constitutes a huge advantage [Jeffers and Reinders 2013]. 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 HSP input. 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 9-13 of the Algorithm 3 once a candidate solution is tagged as $false$. However, we implemented this loop in a way that we avoid 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.

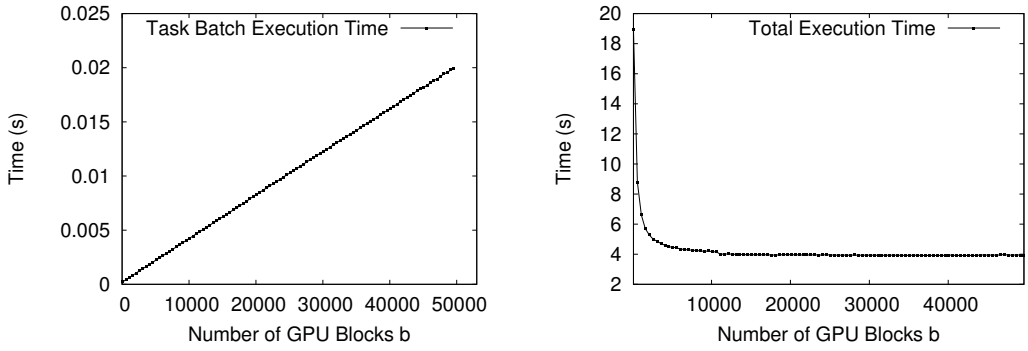
4. Experimental Results

Here we present the experimental results obtained with our hybrid HSP implementation, using a two-node heterogeneous cluster described in Table 1. 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.

For the experiments we generated HSP instances with $|X| = 8192$ (number of variables), collection of clauses $\mathcal{S}$ with $|\mathcal{S}| = 1024$ and $k = 3$. These HSP instances were randomly generated using the `rand()` function of the 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 HSP instance and finish when the results are returned.

4.1. 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 2.2, smaller task batch sizes permit the generation of more task batches and a better load-distribution with heterogeneous platforms. Figure 2(a) 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 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 2(b) 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 [NVIDIA 2016]. Therefore, the number of HSP solution candidates to be evaluated per each task batch that will be processed in the PUs is $bMin{\cdot}v = 20000{*}128 = 2560000$.



(a) Task Batch Execution Time

(b) Total Execution Time

Figure 2. Task batch execution times and total execution times of our proposed algorithm in function of the number of GPU blocks parameter b.

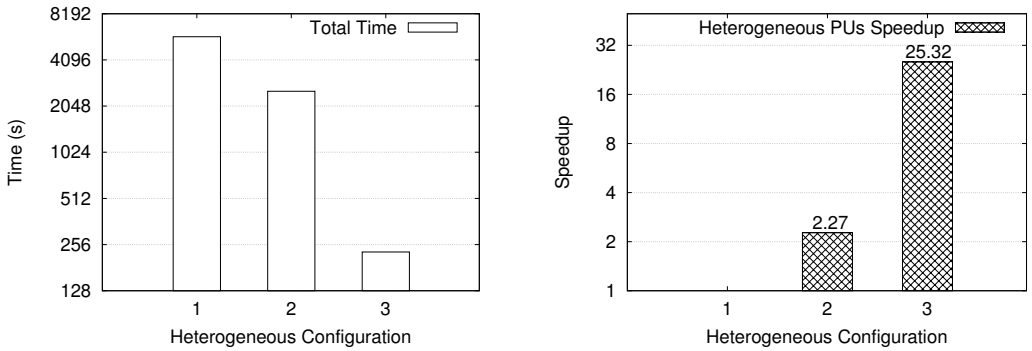
4.2. Performance Evaluation

We evaluated the scalability of our hybrid HSP implementation using a heterogeneous cluster composed of CPUs, GPUs and MICs. Figure 3(a) 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 solve this HSP instance in only 4.1 hours with the four PUs (configuration 3 of Table 2. Result not shown in Figure 3(a)). With only two CPUs, the execution time would take about 25.32 times longer, or 103 hours.

Figure 3(b) shows the obtained speedups, measured as the ratios between the execution time of the HSP algorithm using a configuration shown in Table 2 and the execution time of the HSP algorithm using the dual socket CPUs (configuration 1 of Table 2). With

Table 2. Heterogeneous configuration used in the performance experiments.

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



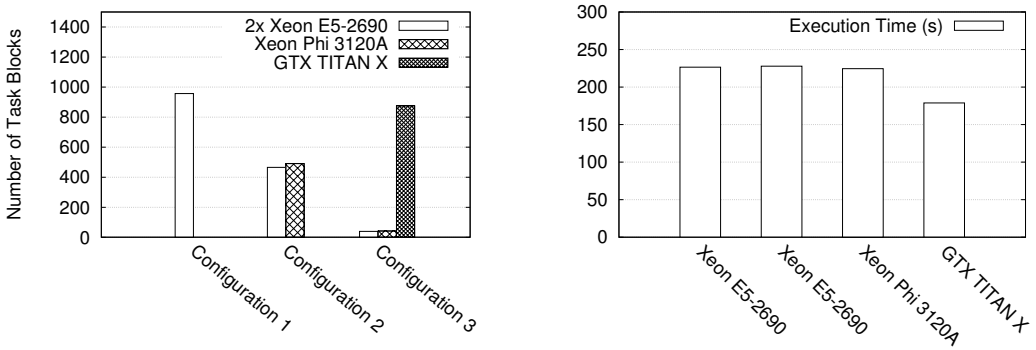
(a) Execution Time (log_2 scale on y axis) (b) Speedup (log_2 scale on y axis)

Figure 3. Execution time and consequent speedups of proposed hybrid algorithm in function of the number of processing units.

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 the HSP algorithm can at least double by adding only one Xeon Phi device, therefore attesting the HPC capabilities of the Xeon Phi in our HSP 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 HSP 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 HSP 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 HSP 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.

To get a better understanding of these results, we monitored the number of task batches assigned to each processing unit (Figure 4(a)). 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 HSP algorithm. Figure 4(b) 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.



(a) Number of task batches assigned to each PU.

(b) Total time that each processing unit has spent solving HSP.

Figure 4. Load balance evaluation results of our hybrid exact HSP algorithm.

5. Conclusions

In this paper we presented an exact hybrid CPU-GPU-MIC algorithm for the Hitting Set Problem suited for large input size applications. We solved instances of HSP 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 device 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 HSP 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. Furthermore, with this cluster, we unprecedentedly solved HSP instances in order of tens of thousands of variables in less than 5 hours, which characterizes a formidable performance increase with regard to exact HSP algorithms. 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 HSP 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. But for applications where the cardinalities are not usually high, such as gene regulatory networks inference [Carastan-Santos et al. 2015, Carastan-Santos et al. 2017], the solution proposed here is suitable.

For future works, we plan to further optimize our algorithm to increase the efficiency of finding HSP solutions with very high cardinalities. Moreover, we also plan to provide a full software package for solving HSP, with a friendly interface and an autotuning procedure, capable to automatically discover the best tuning parameters for a specific cluster.

Acknowledgment

The authors would like to thank UFABC, FAPESP (Procs. n. 2013/26644-1, 2014/50937-1 and 2015/01587-0), CNPq (Procs. n. 559955/2010-3, 302620/2014-1 and 465446/2014-0) and CAPES for the financial support.

References

Andrade, G., Ferreira, R., Teodoro, G., Rocha, L., Saltz, J. H., and Kurc, T. (2014). Efficient execution of microscopy image analysis on CPU, GPU, and MIC equipped cluster systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 89–96. IEEE.

Carastan-Santos, D., de Camargo, R. Y., Martins, D. C., Song, S. W., and Rozante, L. C. (2017). Finding exact hitting set solutions for systems biology applications using heterogeneous gpu clusters. *Future Generation Computer Systems*, 67:418–429.

Carastan-Santos, D., Yokoingawa De Camargo, R., Correa Martins, D., Song, S. W., Silva Rozante, L., and Ferreira Borelli, F. (2015). A multi-gpu hitting set algorithm for grns inference. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 313–322.

Cardoso, N. and Abreu, R. (2013). Mhs2: A map-reduce heuristic-driven minimal hitting set search algorithm. In *International Conference on Multicore Software Engineering, Performance, and Tools*, pages 25–36. Springer.

Cendic, B. L. (2014). A genetic algorithm for the minimum hitting set. *Scientific Publications of the State University of Novi Pazar*, 6(2):107.

Duran, A. and Klemm, M. (2012). The intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE.

Gainer-Dewar, A. and Vera-Licona, P. (2017). The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100.

Garey, M. R. and Johnson, D. S. (1999). *Computers and Intractability - A guide to the Theory of NP-completeness*. W. H. Freeman and Company.

Hochbaum, D. S. (1997). *Aproximation Algorithms for NP-Hard Problems*. PWS Publishi ng Company.

Hvidsten, T. R., Lægreid, A., and Komorowski, J. (2003). Learning rule-based models of biological process from gene expression time profiles using gene ontology. *Bioinformatics*, 19(9):1116–1123.

Ideker, T. E., Thorsson, V., and Karp, R. M. (2000). Discovery of regulatory interactions through perturbation: inference and experimental design. *Pacific symposium on biocomputing*, 5:302–313.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

Knuth, D. E. (2005). *The Art of Computer Programming, Fascicle 3: Generating All Combinations and Partitions*, volume 4. Addison-Wesley, Reading.

Kolman, P. and Walen, T. (2007). Reversal distance for strings with duplicates: Linear time approximation using hitting set. *The Electronic Journal of Combinatorics*, 14(1):11.

Murakami, K. and Uno, T. (2014). Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83–94.

NVIDIA (2016). Maxwell: The Most Advanced CUDA GPU Ever Made. `https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`. Online; last access 18 july 2016.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.

Pearson, W. R., Robins, G., Wrege, D. E., and Zhang, T. (1996). On the primer selection problem in polymerase chain reaction experiments. *Discrete Applied Mathematics*, 71(1):231–246.

Reza, H., Aguilar, M., and Jalal, S. F. (2015). Regression testing of gpu/mic systems for hpcc. In *Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science*, pages 30–37. IEEE Press.

Ruchkys, D. P. and Song, S. W. (2003). A parallel solution to infer genetic network architectures in gene expression analysis. *International Journal of High Performance Computing Applications*, 17(2):163–172.

Shi, L. and Cai, X. (2010). An exact fast algorithm for minimum hitting set. *Int. Joint Conference on Computational Science and Optimization*, 1:64–67.

Sîrbu, A. and Babaoglu, O. (2016). Power consumption modeling and prediction in a hybrid cpu-gpu-mic supercomputer. In *European Conference on Parallel Processing*, pages 117–130. Springer.

Wolfe, N., Liu, T., Carothers, C., and Xu, X. G. (2014). 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*, pages 49–52. IEEE Press.