Improving Constraint Solving on Parallel Hybrid Systems

Pedro Roque
proque@di.uevora.pt
University of Évora / LISP
Portugal

Vasco Pedro
vp@di.uevora.pt
University of Évora / LISP
Portugal

Daniel Diaz
daniel.diaz@univ-paris1.fr
University Paris-1 / CRI
France

Salvador Abreu
spa@uevora.pt
University of Évora / LISP
Portugal

***Abstract*—Recently, we developed the Parallel Heterogeneous Architecture Constraint Toolkit (PHACT), which is a multi-threaded constraint solver capable of using all the available devices which are compatible with OpenCL, in order to speed up the constraint satisfaction process.**

In this article, we introduce an evolution of PHACT which includes the ability to execute FlatZinc and MiniZinc models, as well as architectural improvements which boost the performance in solving CSPs, especially when using GPUs.

***Index Terms*—Constraint satisfaction, Parallel execution, GPU, Hybrid systems**

I. Introduction

Constraint Programming has been successfully used to model and find solutions for many real-world problems, such as planning and scheduling [2], resources allocation [9] route definition [4], Costas Arrays [8], among several others. When thus formalized, models are called Constraint Satisfaction Problems (CSPs.)

A CSP can be solved with different intents: finding a single solution, finding all solutions, optimizing for the best solution (in which case it is termed a Constraint Optimization Problem, or COP), or just to count the solutions to a given problem instance. The search for solutions for those problems has evolved from being executed sequentially on a single CPU to being executed through distributed solvers using multiple single-threaded CPUs on networked environments [26]. Currently, several solvers already exist which are capable of using multi-threaded CPUs [6], [7], some of them even manage to do so in distributed environments [21]. However, only a few are capable of using multiple computational devices on the same machine, for instance a CPU and massively parallel computing platforms, such as GPUs, to achieve greater performance [1], [5].

PHACT provides its own programming interface for modeling CSPs, and it has been recently extended to be able to load a model description from a MiniZinc or FlatZinc [19] file. Once a CSP has been described, the solver can make use of several different computing resources, ranging from a single thread on a CPU, GPU or MIC, to thousands of threads spread among several of those devices, in order to speed up the solving process. To that end, two levels of load balancing are considered: one between devices, and another between the threads within each device.

To enable this sort of load balancing, the search space is a-priori split into multiple disjoint sub-search spaces that are grouped as sets and distributed among the devices. The number of sub-search spaces that compose each set is dynamically recomputed during the solving process, taking into account the actual observed performance history for each device, when handling previous sets. Within the computing devices, each thread solves one sub-search space at a time until a solution, the best solution or all solutions are found, or until the set has been fully explored.

Work dealing with constraint satisfaction using techniques for complete search is presented in Section II. Section III describes the architecture and features of PHACT, and in Section IV we present and discuss the results achieved when using from a single CPU thread to multiple threads on CPUs and GPUs to solve a set of CSPs. In Section IV, the performance of PHACT in solving those CSPs is also evaluated and compared to the competing general-purpose solver Gecode [27]. In Section V we discuss the overall results and reflect on perspectives for further developments.

II. Related work

Over the past couple of decades, several constraint solvers have been developed, and techniques to improve their performance have proliferated. Some of these are designed to make use of parallel architectures, such as multicore CPUs, and some are even capable of effectively running in distributed environments, as the Parallel Complete Constraint Solver (PaCCS) [15], [21]. PaCCS is a complete constraint solver which uses work stealing techniques to distribute the work among multi-threaded CPUs on a distributed environment.

PaCCS implements work stealing by splitting the search space over multiple agents (workers) and by stealing a new search space from their co-workers after finishing the last one. Each worker is built as a search engine that interleaves rule-based propagation and search.

PaCCS was implemented for Unix, in the C programming language with the objective of providing a back end to a higher level language allowing for constraint modelling constructs and the transparent usage of multithreading CPUs, possibly in distributed environments. PaCCS uses POSIX threads for an easier memory sharing and the MPI standard to distribute

the search space through the workers and for worker pool coordination.

The workers are grouped in teams, each one corresponding to a MPI process, and each worker in the team – including the team controller – is a POSIX thread in that process. The team controller is responsible for managing the communication between the workers of that team and with the other teams controllers.

PaCCS is able to run on multiprocessing systems constituted by multiprocessors, networked computers or both.

Pedro [21] benchmarked the implementation of PaCCS by solving n-Queens, Langford Numbers, Golomb Ruler and Quadratic Assignment problems. The results obtained showed that PaCCS is a very scalable parallel constraint solver which achieved an almost linear performance gain for all the tested problems (when computing all solutions).

Besides work stealing, which needs as much communication and concurrency control between workers and/or masters as the number of workers, some authors like Régin *et al.* [25], split the search-space only at the beginning of the solving process.

Their technique, called Embarrassingly Parallel Search (EPS), consists in filling a queue of sub-search spaces that were not detected as inconsistent by a solver during their generation, and distributing them among workers, for exploration.

These authors' design includes a master worker, responsible for generating these sub-search spaces, maintaining the queue, and collecting results. The authors stated that the optimal number of sub-search spaces that would lead to a best load-balancing between the workers ranged from 30 to 100 per worker. When the queue is full, each worker takes a sub-search space for exploration. After depleting that sub-search each worker takes another one from the queue, until no work remains.

Régin *et al.* [25] tested their technique with the Gecode [27] and the or-tools [11] solvers, using 20 problems modeled in FlatZinc [20]. Each problem was split by their implementation and each worker was a thread executing an instance of the solver, which explores a sub-search space at a time.

These authors tested their implementation on a machine with 40 cores, achieving a geometric mean speedup of 21.3 with or-tools and 13.8 with Gecode, when comparing with a sequential run. When using the 40 cores, their implementation with Gecode achieved a geometric mean speedup 1.8 times greater than Gecode alone, which uses work stealing for load-balancing.

Campeotto *et al.* [5] developed a complete CSP solver with the Nvidia Compute Unified Device Architecture (CUDA) for Nvidia GPUs. The implementation of constraint propagation follows three main guidelines:

- The propagation and consistency check for each constraint is assigned to a block of threads;
- The domain of each variable is filtered by one thread;
- The constraints related with few variables are propagated in the CPU, while the remaining constraints are filtered by the GPU. This division bound is dynamic to keep the load balanced between host (CPU) and device (GPU).

The data transfer between host and device is reduced to a minimum due to the low bandwidth transfer rate. At each propagation, the domains of the variables that are not labeled yet and the events occurred during the current exploration are copied to the GPU global memory. This data transfer is made asynchronously and only after the CPU has finished its sequential propagation do both the GPU and CPU get synchronized.

The simplest propagators are invoked by a single block of threads, and the more complex ones are invoked by more than one block. For this purpose, the constraints are divided between GPU and CPU, and the GPU part is also divided into the ones that should be split between multiple block of threads and the ones that should not.

Campeotto *et al.* [5] used the MiniZinc/FlatZinc constraint modeling language for generating the solver input and implemented the propagators for FlatZinc constraints and other specific propagators.

These authors obtained speedups of up to 6.61, with problems like the Langford problem and some real-world problems such as the modified Renault problem [16], when comparing a sequential execution on a CPU with the hybrid CPU/GPU version.

III. Solver architecture

PHACT is a complete propagation-based constraint solver, capable of finding a solution for a CSP if one exists. It is meant to be able to use all the (parallel) processing power of the devices available on a system, such as CPUs, GPUs and MICs, in order to speed up the solving process for constraint satisfaction problems.

The solver is composed of a *master process* which collects information about the devices that are available on the system, such as the number of cores and the type of device (CPU, GPU or MIC), and calculates the number of sub-search spaces that will be created to distribute among those devices. For each device, there will be a thread (communicator) responsible for handling all communication with that device. Inside each device there will be a range of threads (search engines) that will perform labeling, constraint propagation and backtracking, on one sub-search space at a time. The number of search engines that will be created inside each device depends on the number of cores and type of that device, and may vary from 8 on a hyperthreaded quad-core CPU to over 50,000 on a GPU.

PHACT may be used to count all solutions to a given CSP, to find just one solution or to find the best one, according to some criterion (for optimization problems).

A. Framework

PHACT provides its own programming interface for implementing CSPs, through a set of methods to create variables and constraints, and the ability to define the search goal, that is, whether the objective is optimization, counting all the solutions or finding the first solution. The solver is also capable

of loading MiniZinc and FlatZinc models, thereby providing direct compatibility with several community developed CSP models, which rely on this constraint modeling language. MiniZinc is a high level constraint modeling language, that gets compiled into FlatZinc, which is readable by many constraint solvers [19].

The FlatZinc interpreter uses Flex and Bison [14] and is still undergoing development, but is already capable of recognizing a useful subset of the FlatZinc specification [3]. In order to load MiniZinc models, we resort to the "mzn2fzn" tool [20] that compiles the MiniZinc model to FlatZinc, which is then loaded to PHACT through its own FlatZinc interpreter.

The FlatZinc interpreter and PHACT are implemented in the C programming language, but the search engines that run on the actual computing devices, as well as the communication and control of those devices are implemented in OpenCL [18]. OpenCL enables PHACT to execute on several types of device, from different vendors and may even be executed on different operating systems, including Linux and Microsoft Windows.

B. Search space splitting and work distribution

In order to distribute the work among the devices, PHACT splits the search space into multiple sub-search spaces. Search-space splitting is done by the master process by partitioning the domains over one or more of the variables of the problem, so that the resulting sub-search spaces form a partition of the entire search space. The number and the size of the sub-search spaces thus created depend on the number of work-items which will be used and, in our experimentation, may go up to a few millions.

Example 1 shows the result of splitting the search space of a CSP with three variables, $V1$, $V2$ and $V3$, all with domain $\{1, 2\}$, into 4 sub-search spaces, $SS1$, $SS2$, $SS3$ and $SS4$.

Example 1: Creation of 4 sub-search spaces

$$SS1 = \{V1 = \{1\}, V2 = \{1\}, V3 = \{1, 2\}\}$$
$$SS2 = \{V1 = \{1\}, V2 = \{2\}, V3 = \{1, 2\}\}$$
$$SS3 = \{V1 = \{2\}, V2 = \{1\}, V3 = \{1, 2\}\}$$
$$SS4 = \{V1 = \{2\}, V2 = \{2\}, V3 = \{1, 2\}\}$$

As any given device will have multiple search engines running in parallel, the computed partition is organized into blocks of contiguous sub-search spaces that will be handled by each device, one at a time. The number of sub-search spaces that will make up each block will vary along the solving process and depends on the relative performance of each device in exploring the previous blocks.

The communicator threads running on the host launch the execution of the search engines on the devices, compute the size of the blocks of sub-search spaces to hand over to the respective device, and coordinate the progress of the solving process as each device finishes exploring its assigned block. The coordination of the devices consists in: assessing the state of the search, distributing more blocks to the devices, signaling to all the devices that they should stop (when a solution has been found and only one is wanted), or updating the current bound (in optimization problems). In the end, the master process collects the results from all the communicator threads and outputs it.

Figure 1 shows a diagram exemplifying the main components of PHACT and their interactions when solving a CSP. Note that, in this example, only four blocks of threads get explored, which would mean that those blocks constituted the full search space (when counting all the solutions). In reality, the number of blocks that are dynamically created along the solving process may go up to a hundred, depending on the number of devices that are used and their performance in solving the current CSP.

C. Load balancing

An essential aspect to consider when parallelizing some task is the even distribution of work between the parallel components. Creating sub-search spaces with balanced domains, when possible, is still no guarantee that the amount of work actually involved in exploring each of them is even remotely similar. To compound the issue, we are dealing with devices with differing characteristics and varying speeds, making it even harder to statically determine an optimal, or even just good, work distribution.

Achieving effective load balancing between devices with such different architectures as CPUs and GPUs is a complex task [13]. When trying to implement dynamic load balancing, two important OpenCL (version 1.2) limitations arise: namely when a device is executing a kernel it is not possible for it to communicate with other devices [10], and the execution of a kernel cannot be paused or stopped. Hence, techniques such as work stealing [7], [22], which requires communication between threads, will not work with kernels that run independently on different devices and load balancing between them must be done on the host side.

To better manage the distribution of work, the host could limit the amount of work it sends to the devices each time, by reducing the number of sub-search spaces in each block. This would cause the devices to synchronize more frequently with the host and allow for a finer control over the behavior of the solver. When working with GPUs, though, the number and the size of data transfers between the devices and the host should be as small as possible, because these are very time consuming operations. So, a balance must be struck between the workload of the devices and the amount of communication needed.

PHACT implements a dynamic load balancing technique which tailors the size of the blocks of sub-search spaces to match the performance of each device solving the current problem, in relative terms, when compared to the performance of the other devices. This feature is extensively described in [24].

IV. Results and discussion

In [24] we described PHACT architecture, its load balancing techniques and compared its performance against Gecode (and PaCCS [21]) on solving some CSPs. Those CSPs were loaded to Gecode through FlatZinc models, but at the time, PHACT

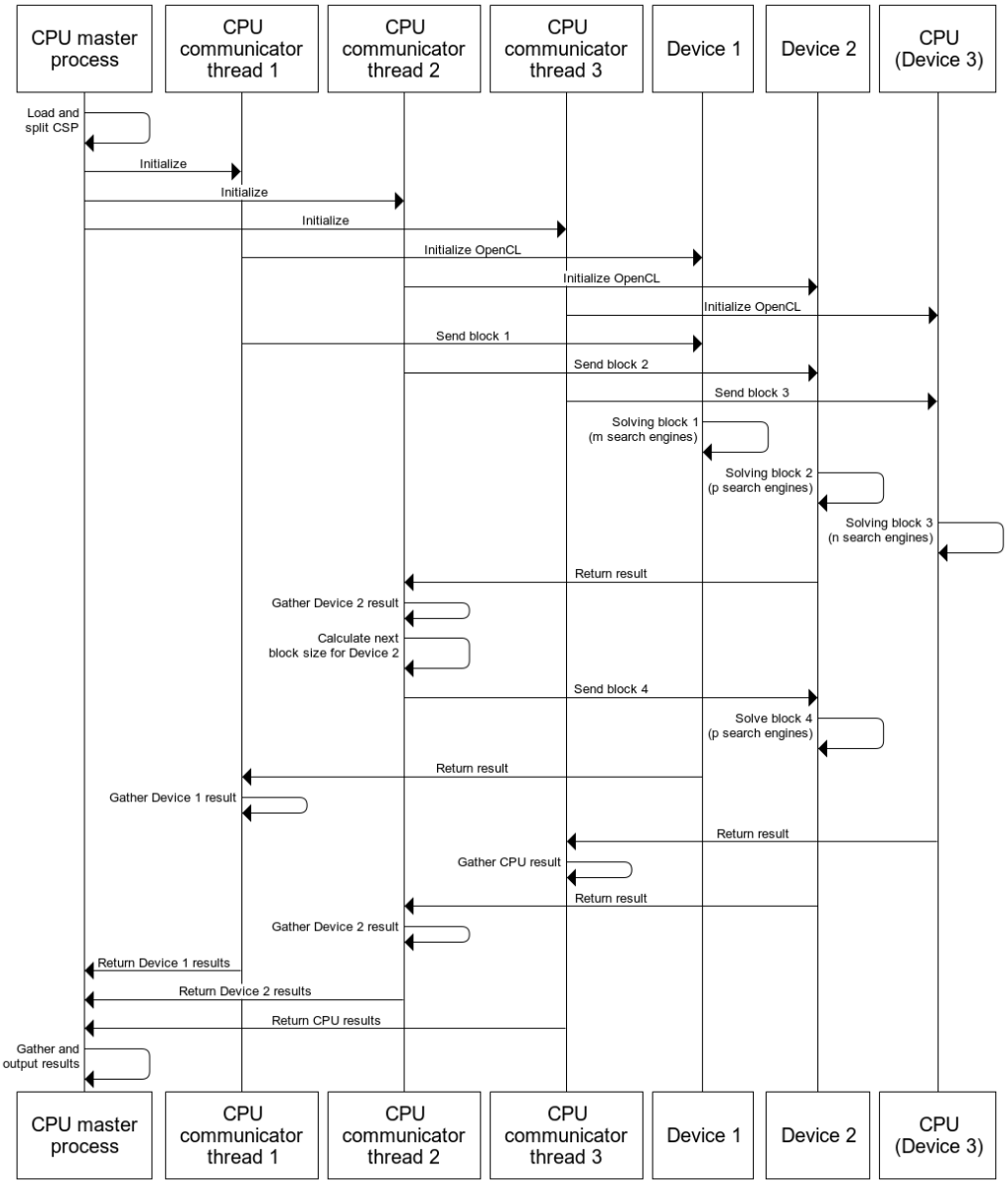

Figure 1. PHACT architecture

did not have a FlatZinc/MiniZinc interpreter, so the same CSPs were implemented for PHACT using its own C interface.

From that comparisons till now, a FlatZinc/MiniZinc interpreter was implemented for PHACT, which allows PHACT to load many existent CSPs modelled in MiniZinc/FlatZinc, and to ensure that Gecode and PHACT are solving the same CSP model.

We evaluated PHACT on a set of benchmarks which entails:

- Finding the first solution for the All Interval problem.
- Optimizing the Golomb Ruler problem.
- Finding all solutions for the following benchmarks: Costas Arrays, Langford Numbers and the Quasigroup problems.

The All Interval and the Langford Numbers problems were retrieved from CSPLib [12] and the remaining CSPs were taken from the MiniZinc Benchmarks suite [17]. All the CSPs were loaded from FlatZinc files.

Those tests were executed on a machine with 128 GB of RAM, hosting 4 AMD Opteron 6376 CPUs with 16 cores each – totaling 64 cores – and two AMD Tahiti GPUs, each one with 32 Streaming Multiprocessors (SMs). Those two GPUs are combined in an AMD Radeon HD 7990, but are managed separately by OpenCL.

PHACT performance was also compared with that of Gecode 5.1.0. We also experimented with Choco version 4.0.4 [23], but the results were non-conclusive, in the time we had

Table I
ELAPSED TIMES AND SPEEDUPS OF PHACT AND GECODE WHEN SOLVING 5 CSPS WITH DIFFERENT NUMBERS OF THREADS

Solver	Threads	All Interval 17 (First)		Costas Array 13 (Count)		Golomb Ruler 11 (Optimization)		Langford Numbers 13 (Count)		Quasigroup 7, 9 (Count)	
		Elapsed time (s)	Speedup vs. 1 thread	Elapsed time (s)	Speedup vs. 1 thread	Elapsed time (s)	Speedup vs. 1 thread	Elapsed time (s)	Speedup vs. 1 thread	Elapsed time (s)	Speedup vs. 1 thread
PHACT	**1**	1320.60		166.37		410.76		448.00		193.65	
	2	868.50	1.52	112.99	1.47	271.65	1.51	278.09	1.61	117.82	1.64
	4	502.18	2.63	60.71	2.74	138.31	2.97	148.99	3.01	62.66	3.09
	8	352.55	3.75	35.08	4.74	80.10	5.13	84.69	5.29	36.34	5.33
	16	334.84	3.94	18.47	9.01	41.06	10.00	43.25	10.36	19.03	10.17
	32	370.52	3.56	10.16	16.38	21.90	18.76	22.51	19.90	10.42	18.58
	64	396.42	3.33	5.97	27.86	12.62	32.56	12.15	36.87	6.10	31.75
Gecode	**1**	3488.64		335.74		635.50		895.65		316.70	
	2	2092.38	1.67	180.33	1.86	336.47	1.89	465.41	1.92	169.86	1.86
	4	1169.26	2.98	87.38	3.84	163.13	3.90	239.74	3.74	86.86	3.65
	8	874.95	3.99	45.74	7.34	83.61	7.60	140.56	6.37	44.76	7.08
	16	2051.83	1.70	26.64	12.60	47.04	13.51	140.42	6.38	27.21	11.64
	32	2485.85	1.40	24.53	13.69	29.63	21.45	184.04	4.87	17.33	18.27
	64	1533.84	2.27	31.46	10.67	23.49	27.05	216.48	4.14	12.50	25.34

available for this experimentation.

Table I displays the elapsed times and speedups when solving one instance of each one of those five CSPs. The elapsed times are the result of the geometric mean of five runs of each instance of the problem, when using from 1 to 64 cores of the CPU. The speedup is calculated when comparing the time taken to solve the same problem sequentially (i.e. with one thread).

In the table, one may observe that PHACT was capable of achieving a speedup when increasing the available number of cores for all the CSPs, except when finding the first solution for the All Interval problem with more than 16 cores. As for Gecode, only when solving the Quasigroup and the Golomb Ruler problems was it capable of always achieving speedups when doubling the previous number of cores, up to 64. When solving the All Interval problem, after increasing the number of cores above 8, it started to get slower. The same situation occurs with the Costas Arrays benchmark, beyond 32 cores and with the Langford Numbers test, it happens right after 16 cores.

The speedups presented in Table I are displayed in Figure 2 with an increasing number of threads used for solving each one of the CSPs with the two solvers. We can observe that when using 64 cores, most PHACT speedups were greater than the

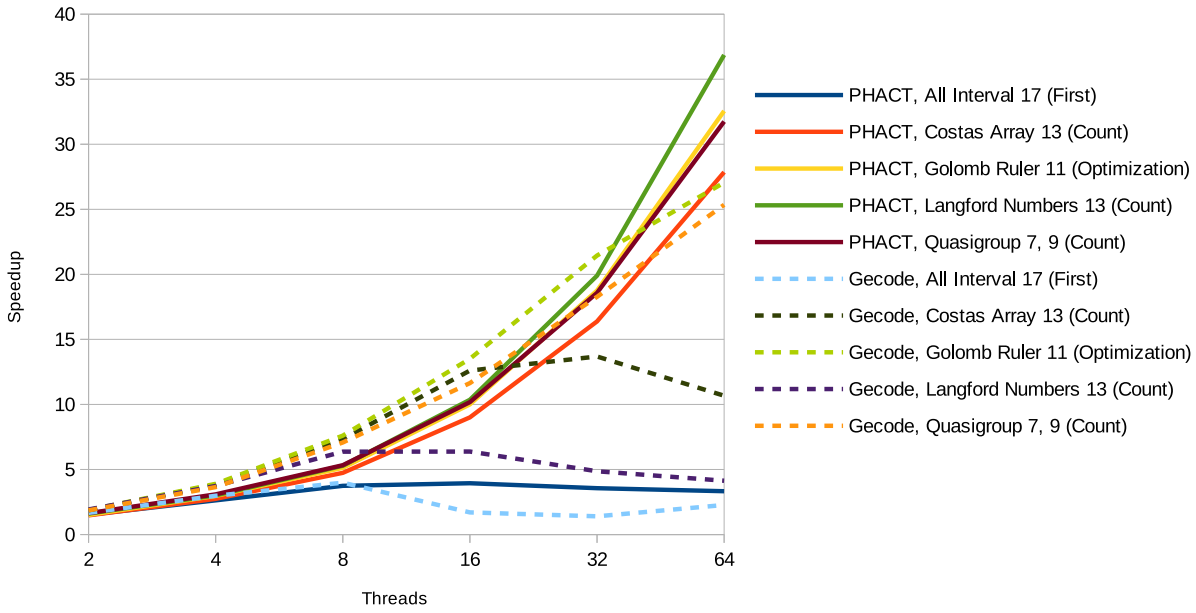

Figure 2. Speedups achieved by PHACT and Gecode with up to 64 threads

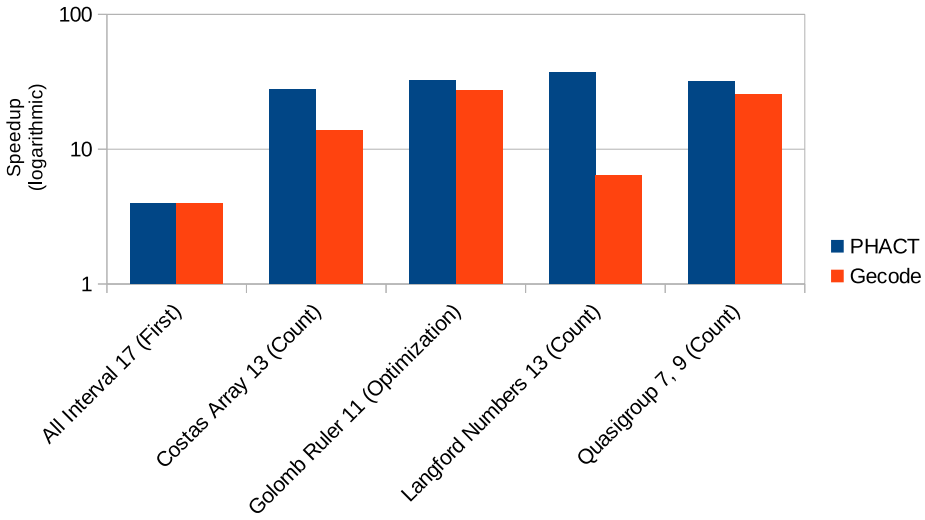

Figure 3. Best speedups achieved by PHACT and Gecode when using more than one thread

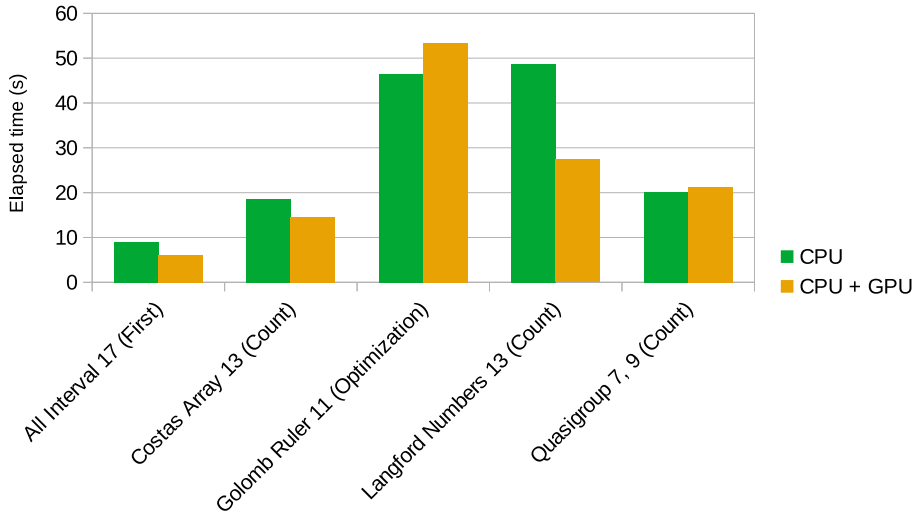

Figure 4. Elapsed times by PHACT when using only an Intel I7-4870HQ CPU, or this CPU and one Nvidia Geforce 980M GTX GPU

other systems.

Figure 3 presents the best speedups achieved by PHACT against Gecode, in which it becomes clear that PHACT achieves better speedups with all the CSPs. Note that, as the speedups shown in this chart are the best achieved by each solver, the number of threads used for that result may vary between solvers and CSPs, according to the speedups presented in Table I.

PHACT was capable of achieving a top speedup of 36.87, which was better than Gecode (27.05), supporting our claim that hybrid systems may be effectively used to solve CSPs. These results shows that, at least for this set of CSPs, the load balancing techniques implemented by PHACT are achieving better results than that of Gecode, which uses work stealing.

As mentioned in section III and described in [24], PHACT is also capable of using GPUs, MICs and any other device compatible with OpenCL in order to speed the solving process up. The chart in Figure 4 shows the speedups achieved when using only an Intel I7-4870HQ CPU (4 cores, 8 threads), or this CPU and one Nvidia Geforce 980M GTX GPU (12 SMs).

We are yet improving the algorithm that does the load balancing between devices, especially for optimization problems, where besides the size of the blocks of sub-search spaces, we must also consider the better solutions that any device may find. This is problematic, as we cannot pause or stop a device that is exploring a block of sub-search spaces to inform it that a better solution was already found, and that it may prune its search with the new best cost.

This may lead to some uninformed devices trying to find

solutions that are already worse than the best one already found, which will increase the total time taken to fully explore the problem. That explains the worse result achieved with PHACT when optimizing the Golomb Ruler with the CPU and GPU when compared with the CPU only.

From the 5 CSPs that were solved, the Quasigroup was the problem with the highest memory requirements, which depleted the GPU RAM. This forced PHACT to limit the number of threads that would run on the GPU, and that limitation together with the depleted RAM decreased the GPUs performance. The performance impact was such, that the GPUs exploration work did not compensate the increased workload that the CPU incurred for controlling and communicating with the GPU.

Nevertheless, for the All Interval, the Costas Array and the Langford Numbers problems, the GPU was capable of speeding up the solving process by .45, 1.28 and 1.78 times, respectively. The geometric mean of the speedups for the five CSPs was of 1.22, which shows that the GPUs can effectively be used for speeding up the solving process.

V. Conclusion and future work

In this paper we described PHACT, a constraint solver capable of achieving competitive speedups in multi-threaded CPUs when compared to state-of-the art solvers, such as Gecode. We tested the two solvers on five different benchmark CSPs, running on a machine with 64 regular cores, and PHACT achieved the best speedups of the two when using all cores, a result which holds across all the benchmarks.

Most significantly, PHACT is also capable of using other devices to help in speeding up the solving process. We used GPUs to help the (multiple) CPUs in solving the same CSPs, which brought the hybrid solver to achieve a top speedup of 1.78, relative to the multicore implementation. This clearly shows that GPUs can effectively be used to improve performance in solving CSPs.

We plan to further explore the performance impact of using a hybrid platform, as enabled by PHACT, by diversifying the auxiliary computational devices: we plan to use Intel Xeon Phi, AMD and nVidia GPUs, separately and in combination. Another aspect which needs further work is the implementation of more base and global constraints, together with their FlatZinc interface.

Acknowledgments

This work was partly funded by Fundação para a Ciência e Tecnologia (FCT) under grant UID/CEC/4668/2016 (LISP). Some of the experimentation was carried out on the `khromeleque` cluster of the University of Évora, which was partly funded by grants ALENT-07-0262-FEDER-001872 and ALENT-07-0262-FEDER-001876.

References

[1] Arbelaez, A., Codognet, P.: A GPU implementation of parallel constraint-based local search. In: 22nd Euromicro International Conference on PDP 14. pp. 648–655. IEEE, Torino, Italy (February 2014)

[2] Barták, R., Salido, M.A.: Constraint satisfaction for planning and scheduling problems. Constraints 16(3), 223–227 (July 2011)

[3] Becket, R.: Specification of flatzinc version 1.6. White paper

[4] Brailsford, S., Potts, C., Smith, B.: Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research 119, 557–581 (1999)

[5] Campeotto, F., Palù, A.D., Dovier, A., Fioretto, F., Pontelli, E.: Exploring the use of GPUs in constraint solving. In: Flatt, M., Guo, H.F. (eds.) PADL 2014. LNCS, vol. 8324, pp. 152–167. San Diego, CA, USA (January 2014)

[6] Caniou, Y., Codognet, P., Diaz, D., Abreu, S.: Experiments in parallel constraint-based local search. In: The 11th European Conference on Evolutionary Computation and Metaheuristics in Combinatorial Optimization (EvoCOP 2011). LNCS, vol. 6622, pp. 96–107. Springer (2011)

[7] Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Lisbon, Portugal (September 2009)

[8] Diaz, D., Richoux, F., Codognet, P., Caniou, Y., Abreu, S.: Constraint-based local search for the costas array problem. In: Hamadi, Y., Schoenauer, M. (eds.) Learning and Intelligent Optimization: 6th International Conference. LNCS, vol. 7219, pp. 378–383. Springer (2012)

[9] Filho, C., Rocha, D., Costa, M., Albuquerque, P.: Using constraint satisfaction problem approach to solve human resource allocation problems in cooperative health services. Expert Syst. Appl. 39(1), 385–394 (2012)

[10] Gaster, B., Howes, L., Kaeli, D., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Morgan Kaufmann Publishers Inc., San Francisco, USA (2011)

[11] Google Inc.: OR-Tools - google optimization tools. https://developers.google.com/optimization/, [Online; accessed 5-February-2018]

[12] Jefferson, C., Miguel, I., Hnich, B., Walsh, T., Gent, I.P.: CSPLib: A problem library for constraints. http://www.csplib.org (1999)

[13] Jenkins, J., Arkatkar, I., Owens, J., Choudhary, A., Samatova, N.: Lessons learned from exploring the backtracking paradigm on the GPU. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011 Parallel Processing, LNCS, vol. 6853, pp. 425–437. Springer Berlin Heidelberg (2011)

[14] Levine, J., John, L.: Flex & Bison. O'Reilly Media, Inc., 1st edn. (2009)

[15] Machado, R., Pedro, V., Abreu, S.: On the scalability of constraint programming on hierarchical multiprocessor systems. In: 2013 42nd International Conference on Parallel Processing. pp. 530–535 (Oct 2013)

[16] Mairy, J.B., Deville, Y., Lecoutre, C.: CPAIOR 2014, pp. 235–250. Springer International Publishing (2014)

[17] MIT: A suite of minizinc benchmarks. https://github.com/MiniZinc/minizinc-benchmarks (2017), [Online; accessed 20-January-2018]

[18] Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide. Addison-Wesley Professional, 1st edn. (2011)

[19] Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: CP 2007. pp. 529–543. Springer-Verlag, Berlin, Heidelberg (2007)

[20] Optimisation Research Group NICTA: MiniZinc and FlatZinc. http://www.minizinc.org/, [Online; accessed 9-January-2018]

[21] Pedro, V.: Constraint Programming on Hierarchical Multiprocessor Systems. Ph.D. thesis, Universidade de Évora (2012)

[22] Pedro, V., Abreu, S.: Distributed work stealing for constraint solving. In: Vidal, G., Zhou, N.F. (eds.) CICLOPS-WLPE 2010. Edinburgh, Scotland, U.K. (July 2010)

[23] Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org

[24] Roque, P., Pedro, V., Abreu, S.: Constraint solving on hybrid systems. In: Dietmar Seipel, Michael Hanus, S.A. (ed.) Declare 2017 – Conference on Declarative Programming. Würzburg, Germany (September 2017)

[25] Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Schulte, C. (ed.) CP 2013. NCS, vol. 8124, pp. 596–610. Springer Berlin Heidelberg (2013)

[26] Schulte, C.: Parallel search made simple. In: Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T., Perron, L., Schulte, C. (eds.) Proceedings of TRICS: CP 2000. Singapore (September 2000)

[27] Schulte, C., Duchier, D., Konvicka, F., Szokoli, G., Tack, G.: Generic constraint development environment. http://www.gecode.org/, [Online; accessed 6-January-2018]