Universidade de Évora - Instituto de Investigação e Formação Avançada

Programa de Doutoramento em Informática

Tese de Doutoramento

Constraint Solving on Massively Parallel Systems

Pedro Miguel da Silva Roque

Orientador(es) | Salvador Luís de Bethencourt Pinto de Abreu

Vasco Fernando de Figueiredo Tavares Pedro

Évora 2020

Universidade de Évora - Instituto de Investigação e Formação Avançada

Programa de Doutoramento em Informática

Tese de Doutoramento

Constraint Solving on Massively Parallel Systems

Pedro Miguel da Silva Roque

Orientador(es) | Salvador Luís de Bethencourt Pinto de Abreu

Vasco Fernando de Figueiredo Tavares Pedro

Évora 2020

A tese de doutoramento foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor do Instituto de Investigação e Formação Avançada:

Presidente	Paulo Miguel Torres Duarte Quaresma (Universidade de Évora)
Vogais	Hervé Miguel Cordeiro Paulino (Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologias) Luís Filipe dos Santos Coelho Paquete (Universidade de Coimbra) Luís Manuel Antunes Veiga (Universidade de Lisboa) Pedro Dinis Loureiro Salgueiro (Universidade de Évora) Salvador Luís de Bethencourt Pinto de Abreu (Universidade de Évora) (Orientador)

Évora 2020

À Ana e ao Tiago

Acknowledgments

Quero agradecer à minha mulher e ao meu filho por me deixaram levar horas seguidas a trabalhar para o Doutoramento.

Um grande agradecimento ao Professor Vasco Pedro por me motivar e manter no rumo indicado para conseguir terminar o Programa de Doutoramento. Agradeço também ao Professor Salvador Abreu por me aconselhar fortemente a submeter artigos a conferências no estrangeiro, o que me obrigou a sair da zona de conforto e a adquirir mais experiência em publicações e conferências internacionais.

Agradeço o apoio institucional providenciado pela Fundação para a Ciência e Tecnologia (FCT) sob concessão UID/CEC/4668/2016 (LISP), e pelo ALENT-07-0262-FEDER-001872 e ALENT-07-0262-FEDER-001876 que financiaram parte do cluster `khromeleque` da Universidade de Évora, onde alguma da experimentação foi realizada.

Agradeço também ao Professor Pedro Salgueiro por ajudar a manter o cluster `khromeleque` e a máquina `niia` a funcionar de modo a que eu os pudesse utilizar.

Por último, mas igualmente importante, agradeço também aos restantes elementos da minha família e também aos meus amigos por me apoiarem quando as coisas corriam menos bem e pelas minhas ausências.

A todos, muito obrigado.

Contents

Contents x

List of figures xii

List of tables xiii

Acronyms xv

Abstract xvii

Sumário xix

1 Introduction 1

1.1 Motivation . . . 3

1.2 Objectives and contributions . . . 3

2 Massively parallel devices 5

2.1 Influence of GPUs memory types . . . 10

2.2 GPUs level of parallelism . . . 13

2.3 Conclusion . . . 16

3 State of the art 19

3.1 Complete search . . . 19

3.2 Incomplete search . . . 26

3.3 SAT . . . 27

3.4 Conclusion . . . 29

4 Solver architecture **31**
4.1 Framework . . . 32
4.1.1 OpenCL . . . 32
4.2 Search space splitting and work distribution . . . 35
4.3 Load balancing between devices . . . 37
4.3.1 Finding all the solutions . . . 39
4.3.2 Optimizing . . . 40
4.3.3 Finding one solution . . . 41
4.4 Load balancing inside a device . . . 42
4.5 Communication . . . 43
4.6 Implementation details . . . 44
4.7 Conclusion . . . 46

5 Experimental results **47**
5.1 Results on GPUs . . . 50
5.1.1 Conclusion . . . 54
5.2 Results on MICs . . . 54
5.2.1 Conclusion . . . 55
5.3 Results on CPUs . . . 56
5.3.1 Conclusion . . . 62
5.4 Results on multiple devices . . . 62
5.4.1 Conclusion . . . 72
5.5 Comparison with the state of the art . . . 73
5.6 Conclusion . . . 79

6 Conclusions and future work **81**
6.1 Future work . . . 82

A Implementing a CSP in PHACT **85**

B Compiling and executing PHACT **91**

C Constraint satisfaction problems used for benchmarking **97**

Bibliography **101**

Index **107**

List of figures

2.1 Architecture of Nvidia Geforce GTX 980 . . . 6
2.2 A Maxwell SM (SMM) from Nvidia Geforce GTX 980 . . . 8
2.3 Architecture of AMD Radeon 7970 HD . . . 9
2.4 A GCN compute unit . . . 10
2.5 Using only global memory, or global and shared memory on three GPUs . . . 11
2.6 Using only global memory, or global and shared memory on Geforce . . . 12
2.7 Parallelism of an Nvidia Geforce 980M GTX . . . 13
2.8 Speedups in GPUs when increasing the number of work-groups . . . 15
2.9 Maximum speedups in GPUs and CPUs when comparing with sequential times . . . 16

3.1 Example of a search-tree division into two disjoint sub-search trees . . . 21

4.1 Expanding the search tree to the same depth level in all branches . . . 36
4.2 PHACT components and execution example . . . 38

5.1 Speedups achieved with PHACT when increasing the number of threads on Geforce . . . 52
5.2 Speedups achieved with PHACT when increasing the number of threads on the Tesla . . . 53
5.3 Speedups achieved with PHACT when increasing the number of threads on a Tahiti . . . 54
5.4 Speedups achieved with PHACT when increasing the number of threads on a Titan . . . 55
5.5 Speedups achieved with PHACT when using an increasing number of threads on a MIC . . . 56
5.6 Speedups achieved with PHACT when using from 1 to 8 threads on I7 . . . 57
5.7 Speedups achieved with PHACT when using from 1 to 32 threads on Xeon 1 . . . 59
5.8 Speedups achieved with PHACT when using from 1 to 32 threads on Xeon 3 . . . 60
5.9 Speedups achieved with PHACT when using from 1 to 40 threads on Xeon 2 . . . 61

5.10 Speedups achieved with PHACT when using from 1 to 64 threads on Opteron 62
5.11 Speedups achieved with PHACT when using 1 thread on the I7 and the Geforce 63
5.12 Speedups achieved with PHACT when using 1 thread on the Xeon 2 and the Tesla 65
5.13 Speedups achieved with PHACT when using 1 thread on the Opteron and the Tahitis . . . 66
5.14 Speedups achieved with PHACT when using 1 thread on Xeon 3 and the Titans 67
5.15 Speedups achieved with PHACT when using 1 thread on the Xeon 1 and the MICs 68
5.16 Speedups achieved with PHACT when using the I7 and the Geforce 69
5.17 Speedups achieved with PHACT when using the Xeon 2 and the Tesla 70
5.18 Speedups achieved with PHACT when using the Opteron and the Tahitis 70
5.19 Speedups achieved with PHACT when using the Xeon 3 and the Titans 71
5.20 Speedups achieved with PHACT when using the Xeon 1 and the MICs 72
5.21 PHACT speedups against Gecode, Choco and OR-Tools when adding threads on the I7 . . 74
5.22 PHACT speedups against Gecode and Choco when adding threads on the Xeon 1 74
5.23 PHACT speedups against Gecode, Choco and OR-Tools when adding threads on Xeon 3 . 75
5.24 PHACT speedups against Gecode, Choco and OR-Tools when adding threads on Xeon 2 . 75
5.25 PHACT speedups against Gecode and Choco when adding threads on the Opteron 76
5.26 Speedups of PHACT against Gecode, Choco and OR-Tools on the Xeon 2 78
5.27 Speedups of PHACT against Gecode, Choco and OR-Tools on different devices 79

List of tables

4.1 Example of the calculation of blocks size when using three devices 40

5.1 Information about the CSPs used in experimental results 48
5.2 Seconds needed for PHACT to solve problems when adding threads on the Geforce 51
5.3 Seconds needed for PHACT to solve problems when adding threads on a MIC 56
5.4 Seconds needed for PHACT to solve problems when adding threads on the I7 57
5.5 Number of propagations when using the I7 CPU 58
5.6 Seconds needed for PHACT to solve problems when adding threads on the Opteron 61
5.7 Seconds needed for PHACT to solve problems when using 1 I7 thread and the Geforce . . 63
5.8 Seconds needed for PHACT to solve problems when using 1 Opteron thread and the Tahitis 65
5.9 Seconds needed for PHACT to solve problems when using 1 Xeon 3 thread and the Titans 66
5.10 Seconds needed for PHACT to solve problems when using 1 Xeon 1 thread and the MICs . 67
5.11 Seconds needed for PHACT to solve problems when using the I7 and the Geforce 69
5.12 Seconds needed for different solvers to solve problems when adding thread on the Xeon 2 . 73
5.13 Seconds needed for different solvers to solve problems on the Xeon 2 77

C.1 Source of the CSPs models and data files 98

Acronyms

ACC Accelerator

AES Advanced Encryption Standard

AMD Advanced Micro Devices, Inc.

API Application Programming Interface

BACP Balanced Academic Curriculum Problem

CELL BE Cell Broadband Engine Architecture

CPU Central Process Unit

CSP Constraint Satisfaction Problem

CUDA Compute Unified Device Architecture

DPLL Davis-Putnam-Logemann-Loveland

DRAM Dynamic Random Access Memory

EPS Embarrassingly Parallel Search

GCN Graphics Cores Next

GPC Graphics Processing Clusters

GPQ Global Priority Queue

GPU Graphics Processing Unit

IIFA Instituto de Investigação e Formação Avançada

LD/ST Load/Store

MIC Many Integrated Cores

MPI Message Passing Interface

OpenCL Open Computing Language

PaCCS Parallel Complete Constraint Solver

PCIe Peripheral Component Interconnect Express

PHACT Parallel Heterogeneous Architecture Constraint Toolkit

POSIX Portable Operating System Interface

RAM Random Access Memory

SAT Boolean Satisfiability Problem

SFU Special Function Unit

SIMD Single-Instruction Multiple-Data

SIMT Single-Instruction Multiple-Threads

SM Streaming Multiprocessor

SMM Maxwell Streaming Multiprocessor

SS Search Space

UNSAT Unsatisfiable Boolean Satisfiability Problem

Abstract

Applying parallelism to constraint solving seems a promising approach and it has been done with varying degrees of success. Early attempts to parallelize constraint propagation, which constitutes the core of traditional interleaved propagation and search constraint solving, were hindered by its essentially sequential nature. Recently, parallelization efforts have focussed mainly on the search part of constraint solving. A particular source of parallelism has become pervasive, in the guise of GPUs, able to run thousands of parallel threads, and they have naturally drawn the attention of researchers in parallel constraint solving.

This thesis addresses the challenges faced when using multiple devices for constraint solving, especially GPUs, such as deciding on the appropriate level of parallelism to employ, load balancing and inter-device communication. To overcome these challenges new techniques were implemented in a new constraint solver, named Parallel Heterogeneous Architecture Constraint Toolkit (PHACT), which allows to use one or more CPUs, GPUs, Intel Many Integrated Cores (MIC) and any other device compatible with OpenCL to solve a constraint problem.

Several tests were made to measure the capabilities of some GPUs to solve constraint problems, and the conclusions of these tests are described in this thesis. PHACT's architecture is presented and its performance was measured in each one of five machines, comprising eleven CPUs, six GPUs and two MICs. The tests were made using 10 constraint satisfaction problems, consisting in counting all the solutions, finding one solution or optimizing. Each of the problems has been instantiated with up to three different dimensions. PHACT's performance was also compared with the ones of Gecode, Choco and OR-Tools.

In the end, these tests allowed to detect which techniques implemented in PHACT were already achieving the expected results, and to point changes that may improve PHACT's performance.

Keywords: Constraint solving, Parallelism, GPU, Intel MIC, Heterogeneous systems

Sumário

Resolução de Restrições em Sistemas Massivamente Paralelos

A paralelização na resolução de restrições parece ser uma abordagem promissora e tem sido feita com vários graus de sucesso. As primeiras tentativas de paralelizar a resolução de restrições, que tradicionalmente é constituída pela propagação intercalada com pesquisa, foram prejudicadas pela sua natureza essencialmente sequencial. Recentemente, os esforços de paralelização concentraram-se principalmente na parte de pesquisa da resolução de restrições e surgiu uma fonte particular de paralelismo — as GPUs, capazes de executar milhares de threads em paralelo, o que naturalmente chamou a atenção dos investigadores na área da resolução paralela de restrições.

Esta tese aborda os desafios enfrentados ao usar vários dispositivos para a resolução de restrições, especialmente GPUs, como decidir o nível apropriado de paralelismo a ser utilizado, o balanceamento de carga e a comunicação entre dispositivos. Para superar esses desafios, novas técnicas foram implementadas num novo solucionador de restrições, chamado PHACT (Parallel Heterogeneous Architecture Constraint Toolkit), que permite usar um ou mais CPUs, GPUs, Intel Many Integrated Cores (MIC) e qualquer outro dispositivo compatível com OpenCL para resolver um problema de restrições.

Vários testes foram feitos para medir as capacidades de algumas GPUs na resolução de problemas de restrições, e as conclusões desses testes estão descritas nesta tese. A arquitetura do PHACT é apresentada e o seu desempenho foi medido em cada uma de cinco máquinas, incluindo onze CPUs, seis GPUs e dois MICs. Os testes foram feitos utilizando 10 problemas de restrições diferentes, consistindo na contagem de todas as soluções, encontrar uma solução ou otimização. Cada um dos problemas foi instanciado com um máximo de até três dimensões distintas. O desempenho do PHACT foi também comparado com o do Gecode, do Choco e do OR-Tools.

No final, estes testes permitiram detetar quais as técnicas implementadas no PHACT já estão a atingir os resultados esperados e apontar mudanças que poderão melhorar o desempenho do mesmo.

Keywords: Resolução de restrições, Paralelismo, GPU, Intel MIC, Sistemas heterogéneos

xx

1

Introduction

Constraint Satisfaction Problems (CSPs) allow modeling problems like the N-queens problem [53] and some real life problems like planning and scheduling [6], resource allocation [23] and route definition [10].

A CSP can be briefly described as a set of variables with finite domains, and a set of constraints between the values of those variables. The solution of a CSP is the assignment of one value from the respective domain to each one of the variables, ensuring that all constraints are met [10].

Definition 1. *Formally a CSP is defined as a triple* $P = \langle X, D, C \rangle$*, where:*

- $X = \langle x_1, x_2, ..., x_n \rangle$ *is an n-tuple of variables;*
- $D = \langle D_1, D_2, ..., D_n \rangle$ *is an n-tuple of finite domains, where* D_i *is the domain of the variable* x_i*;*
- $C = \{C_1, C_2, ..., C_m\}$ *is a set of relations between variables in* X*, designated as* constraints*;*
- *A CSP* solution *is an n-tuple* $A = \langle a_1, a_2, ..., a_n \rangle$ *where* $a_i \in D_i$ *is the value assigned to variable* x_i *and all the constraints* C_j *are met.*

The N-queens problem consists in placing n queens in a $n \times n$ chessboard, such that no queen attacks another one. It can be modeled as a CSP with n variables, corresponding to the queens and each one is already mapped to a different matrix column. The domain of these n variables is composed by the integers that correspond to the matrix rows where each queen may be placed [53]. The constraints are a set of rules to ensure that no queen will be placed on the same row or diagonal as another one.

The methods for solving CSPs can be categorized as *incomplete* or *complete*. Incomplete solvers do not guarantee that an existing solution will be found, being mostly used for optimization problems and for large problems that would take too much time to fully explore. On the contrary, complete methods guarantee that if a solution exists, it will be found.

CSPs can then be solved by machines for multiple purposes, like finding a single solution, the best solution, or counting all the solutions for that problem. The search for solutions for these problems evolved from being executed sequentially on a single CPU to being executed through distributed solvers using multiple single-threaded CPUs on networked environments [64]. Currently, several solvers already exist, capable of using multi-threaded CPUs [16, 18], some of them even in distributed environments [53]. However, only a few are capable of using multiple devices on the same machine, like a CPU and massively parallel devices like GPUs to achieve greater performances [4, 15].

GPUs are very fast for mathematical calculations due to their specific hardware components that deal with this kind of operations [22]. When these calculations are to be made over vectors, the GPUs parallel capabilities allow to perform hundreds or even thousands of them simultaneously. These GPUs capabilities make them appealing to be used for other purposes than graphics processing, leading to the creation of the General Purpose Graphics Processing Units (GPGPUs).

GPGPUs are GPUs compatible with frameworks like the Open Computing Language (OpenCL)[1] and the Nvidia Compute Unified Device Architecture (CUDA) [22], which allow them to be used for other purposes than graphics processing, like numeric calculations, artificial intelligence, computer vision and constraint solving. Nowadays most of the GPUs are actually GPGPUs and for simplicity they are normally referred to as GPUs.

The Parallel Heterogeneous Architecture Constraint Toolkit (PHACT) constraint solver was developed to take advantage of any CPU, GPU or Intel Many Integrated Cores (Intel MICs) available on a machine to speed up the solving process of a CSP. The development of this solver, its features, performance and architecture are the main topic of this thesis.

PHACT provides its own interface for modeling CSPs, or can load them from a MiniZinc or FlatZinc [43] model. Then, it can use from a single thread on a CPU, GPU or MIC, to thousands of threads spread among several of those devices to speed up the solving process. For that purpose, the work distribution is made at two levels. One between devices, and another between the threads inside each device.

To distribute the work, the search space is split into multiple disjoint sub-search spaces that are grouped in blocks which are sent to the devices. The number of sub-search spaces that compose each block is dynamically calculated during the solving process, considering the speed of each device when solving the previous blocks. In the devices, each thread will solve a sub-search space at a time until a solution, the best solution or all the solutions are found, or the block is fully explored.

The work described in this thesis was partially published in the articles [58, 59, 60].

In this chapter the motivation and objectives of this thesis are described. One of the main focus of this work is the usage of massively parallel devices for constraint solving, so the influence of the architecture of such

[1] The OpenCL programming language is addressed in Section 4.1.

devices in constraint solving is examined in Chapter 2, where multiple charts are presented to exemplify the parallel capabilities of these devices.

Several works related with constraint solving that use some of the most well known solving techniques for complete search, local search and Boolean Satisfaction Problems (SAT) on CPUs and GPUs are presented in Chapter 3.

Chapter 4 describes the architecture and features of PHACT. The results achieved with PHACT when using from a single CPU thread to multiple threads on CPUs, GPUs and MICs to solve a set of CSPs are shown and discussed in Chapter 5. In that chapter the performance of PHACT in solving those CSPs is also compared with the one of Gecode, Choco and OR-Tools. The conclusions and directions for future work are presented in Chapter 6.

1.1 Motivation

The parallelism of CPUs is already being used with success to speed up the solving processes of harder CSPs [16, 18, 53, 63, 37]. However, very few constraint solvers contemplate the use of more than one device on the same machine, or massively parallel devices like GPUs and MICs. In fact, Jenkins *et al.* recently concluded that the execution model and the architecture of GPUs are not well suited to computations displaying irregular data access and code execution patterns such as backtracking search [31].

Mostly due to the increasing performance of the new GPUs and to the evolution of the programming languages compatible with most known CPUs, GPUs and MICs, this work intended to develop new techniques capable of using the parallel processing power of CPUs, GPUs and MICs to speed up the solving process of constraint problems.

1.2 Objectives and contributions

This thesis main focus is on the development of a constraint solver named PHACT that is already capable of achieving state-of-the-art performances on multi-core CPUs, and can also speed up the solving process by adding GPUs and processors like MICs to solve the problems.

To our knowledge, PHACT is the only constraint solver capable of using simultaneously CPUs, GPUs, MICs and any other device compatible with OpenCL to solve CSPs in a faster manner.

Due to the evolution of programming languages like OpenCL, it is now possible to implement algorithms for GPUs and other massively parallel devices with almost the same difficulty as for CPUs. However, to make those algorithms capable of utilizing the processing power of these devices is much harder.

When we are dealing with complex hardware architectures as the ones of the GPUs, this complexity must be considered during the implementation of the algorithms. PHACT starts by analysing the CSP to solve, the objective of the solving process and the hardware of the devices that will be used, to decide on the value of several parameters that will control the execution. These parameters will define, for example, the number of threads to use on each device and the type of memory where each data structure will be stored at the device. Other more specific control is also used, as for example, which propagators must be compiled and if the portion of the code responsible for optimization must also be compiled or not, as the code that will be executed on the devices is always compiled at runtime.

PHACT implements two stages of work distribution, one for partitioning the work among multiple devices and another one to distribute the work among the threads on each device. New techniques were developed

to try to achieve a good load-balancing between devices with much different architectures and performances. Those techniques allow the solver to dynamically adjust the amount of work to send to each device during the solving process according to its performance up to the moment.

2

Massively parallel devices

Computer processing power began to be augmented by increasing the frequency of the single core processor. In 2001, the Cell Broadband Engine Architecture (CELL BE) [71] and the POWER4 [69] were presented, as innovative multicore microprocessors. Nowadays, the hardware industry states that the computers processing power will keep growing exponentially, but instead of doing so by increasing the processor frequency, it will do it by increasing the number of cores and processors [9].

In recent years, a particular source of parallelism has become pervasive, in the guise of GPUs, able to run thousands of parallel threads, and they have naturally drawn the attention of researchers in parallel constraint solving.

Currently, most computers have a multicore CPU and some of them, specially the personal computers also include a GPU that contains hundreds or even thousands of cores. For more processing power, accelerators may be used, like the Intel Xeon Phi family of MICs, which are coprocessors that combine around 60 Intel processor cores onto a single chip with dedicated RAM, connected to the system through PCI-express [21].

When massive processing power is required, supercomputers are built, like the fastest supercomputer in the world, by June 2019, named Summit, which contains 27,648 Nvidia Volta^{TM} Tensor Core GPUs and 9,216 IBM Power 9 CPUs, achieving a total of 2,414,592 cores [38, 36].

Nevertheless, although programming languages like the Nvidia CUDA and the OpenCL [22] have simplified the programming effort of building software capable of running on GPUs, the peculiar architecture of these devices must be taken into account to achieve good performances.

For example, the Nvidia Geforce GTX 980 is a GPU based on the Nvidia Maxwell architecture [47], which contains 2,048 CUDA cores split by 16 multi-threaded Streaming Multiprocessors (SMs), as represented in Figure 2.1. A CUDA core is similar to the CPUs cores, but with a smaller instruction set.

Figure 2.1: Architecture of Nvidia Geforce GTX 980 [47]

This GPU main components are:

- The PCIe 3.0, which connects the GPU to the CPU;
- The GigaThread engine, which transfers data between the CPU and the GPU and distributes the threads among the SMs;

- 16 SMs;
- All the SMs share the same level 2 cache;
- 4 Graphics Processing Clusters (GPC). Each one is a block of hardware components similar to a self contained GPU [44];
- One raster engine for each GPC, responsible for functions like rasterization;
- 4 GB of GDDR5 global memory (RAM), accessible to all the SMs.

An SM of this architecture is detailed in Figure 2.2. Each SM manages its 128 CUDA cores, Load/Store (LD/ST) units and Special Function Units (SFU), scheduling a top of 4 groups of 32 threads, called warps, to them at a time.

Each SM includes:

- 128 CUDA cores;
- 32 LD/ST units that allow loading, storing and atomic instructions on memory access;
- 32 SFU responsible for fast calculation of 32 bits floating point instructions like square root, cosine and sine;
- 4 Warp Schedulers and 8 Dispatch units that select a warp (set of 32 threads) and dispatch it to the CUDA cores, the LD/ST units or the SFUs. Each Warp Scheduler is capable of dispatching two instructions per warp every clock cycle;
- 96 KB of shared memory, which is available for all the CUDA cores;
- One polymorph engine responsible for specific functions like tessellation [44], and other components for optimizing graphical calculations.

Each warp scheduler can issue two instructions at the same time, as for example, a mathematical operation to a CUDA core and a load operation to a LD/ST unit. Each SM is capable of supporting up to 48 warps at the same time, although it can only execute 4 warps at the same time, 1 per warp scheduler. This means that the Nvidia Geforce GTX 980 is capable of running 2,048 threads simultaneously, which is a greater level of parallelism when compared to CPUs.

However, the Nvidia GPUs have several other factors that influence the amount of threads that can be executed simultaneously, as for example the amount of memory that each thread will require. The Nvidia Geforce GTX 980 possesses 4 GB of global memory and only 96 KB of shared memory, which is much faster and can be used explicitly for storage. These 96 KB are shared among the 32 threads that compose a warp, which means that if the warp needs more than 96 KB not all of the 32 threads will be executed in parallel.

GPUs work in a Single-Instruction Multiple-Threads (SIMT) parallelism model, which means that the 32 threads that compose a warp will only be executed simultaneously if they are executing the same instruction at the same time [14]. If a thread is to execute an instruction different from the instructions of all the other threads, it will be executed alone. As such, the number of threads that may be executed simultaneously, is very dependent on the amount of possible divergent paths implemented in the code with conditional instructions, like “if-else”, therefore, these must be avoided as much as possible.

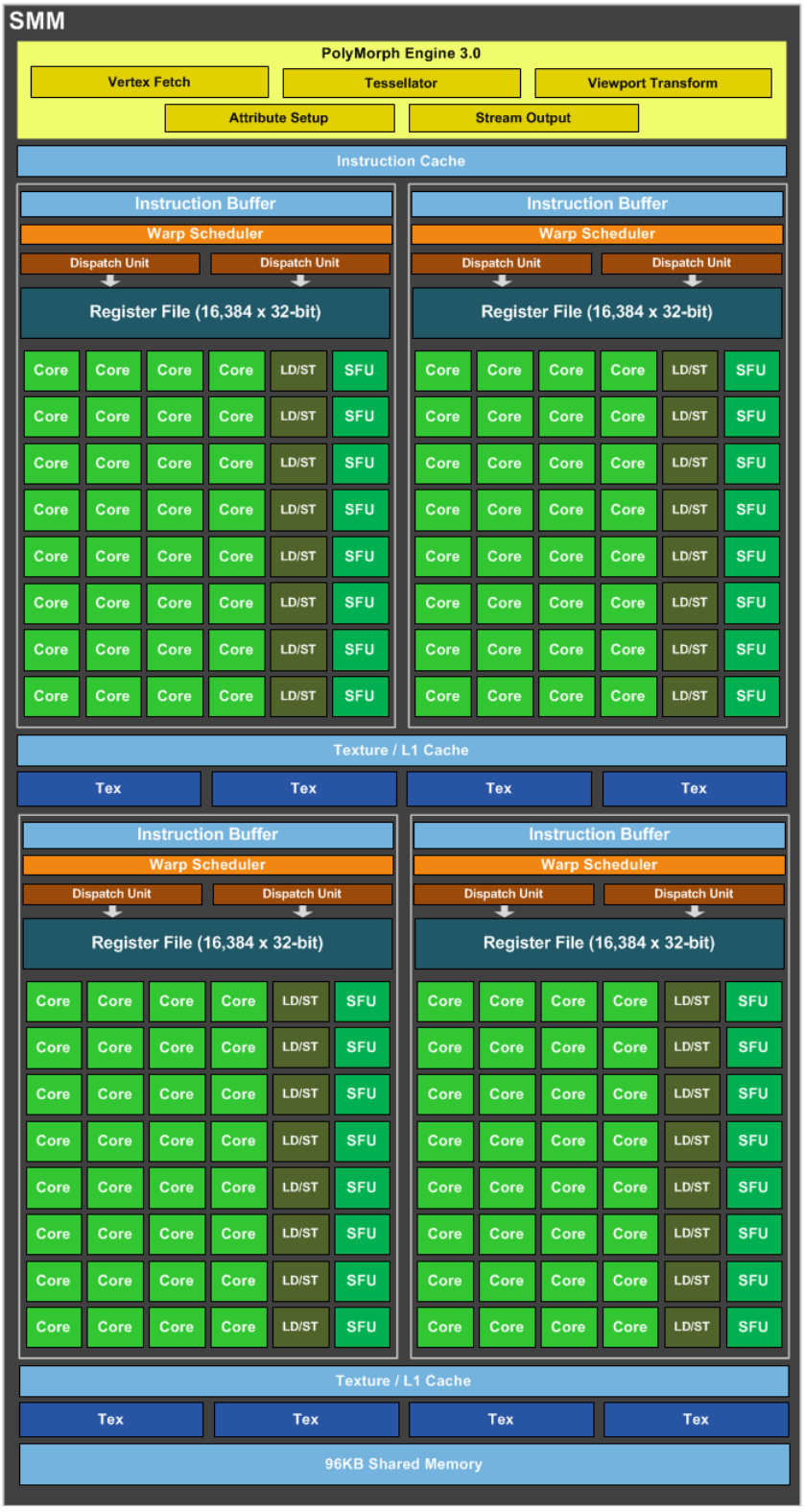

Figure 2.2: A Maxwell SM (SMM) from Nvidia Geforce GTX 980 [47]

The architecture of the AMD GPUs, like the Radeon 7970 HD [2], which is represented in Figure 2.3, is much different when compared with the Nvidia GPUs, like the Geforce GTX 980.

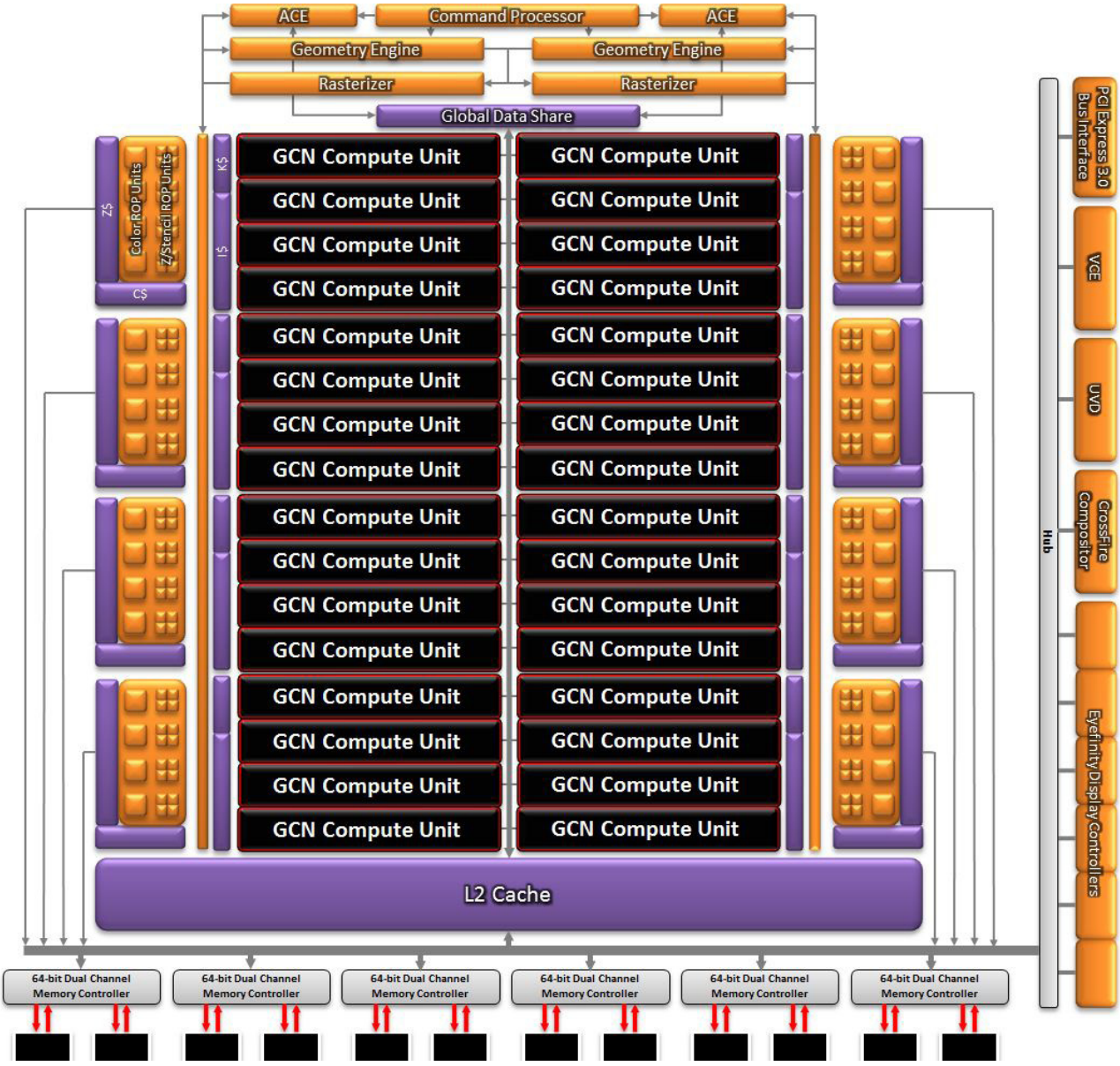

Figure 2.3: Architecture of AMD Radeon 7970 HD [2]

The AMD Radeon 7970 HD includes 32 Graphics Cores Next (GCN) compute units whose roles are similar to the Nvidia SMs, and both GPUs connect to the CPU through PCIe 3.0 and all the GCNs/SMs share the same level 2 cache. Figure 2.4 represents a GCN. Each GCN includes 4 Single-Instruction Multiple-Data (SIMD) vector units, which can be compared to the Nvidia CUDA cores, but unlike CUDA cores, each SIMD vector unit has its own registers and more resources available, which makes a SIMD vector unit much more efficient than a CUDA core.

Each Nvidia GTX 980 warp consists in 32 threads, but the AMD Radeon 7970 HD wavefronts, which correspond to the Nvidia warps, consist of 64 threads, which are executed by the SIMD vector units. By comparing the Nvidia GTX 980 and the AMD Radeon 7970 HD, we may be led to think that the Nvidia GPU is much faster than the AMD GPU, due to its 2048 CUDA cores, when compared with the 128 SIMD (32×4) of the AMD GPU. However, the AMD SIMD vector units are more efficient than the CUDA cores, as each one of them has more private resources than the NVIDIA CUDA cores.

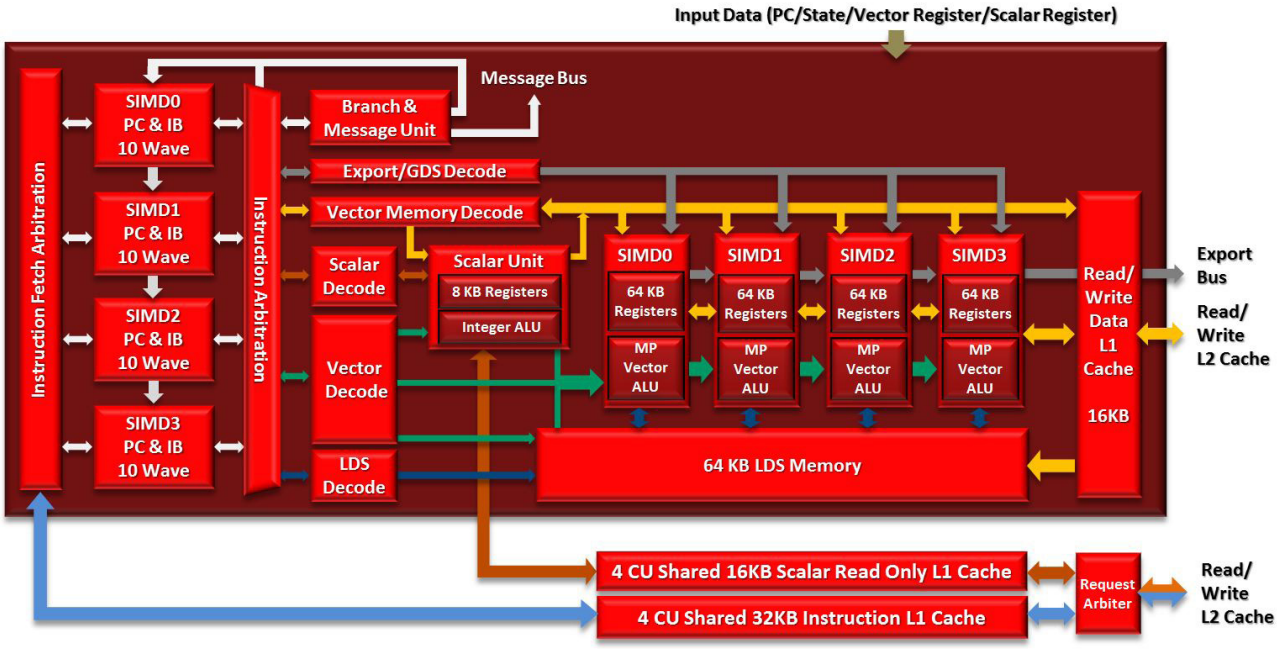

Figure 2.4: A GCN compute unit [2]

This section described the main hardware components of the Nvidia Geforce GTX 980 and of the AMD Radeon 7970 HD, as Nvidia and AMD are the two main GPU vendors and two similar GPUs were used for benchmarking purposes in this chapter and in Chapter 5.

The next section presents some results achieved with PHACT that exemplify the relations between GPUs architecture and different CSPs, when trying to find the best implementation to speed up the process of solving CSPs on GPUs. Section 2.2 demonstrates the level of parallelism provided by some GPUs and compares it to the one made available by some CPUs.

2.1 Influence of GPUs memory types

When implementing software to run on GPUs, the types of memory that are used must be considered, because the usage of shared memory may improve or worsen the software performance. For example, using a single thread on an Nvidia Geforce GTX 980M to count all the solutions for the 12-queens problem, PHACT took 36.4 s when using only global memory and 23.2 s when using also some of the shared memory. However, the shared memory in the GPUs is usually very small (usually 32 KB or 64 KB) which means that for problems that require much more memory, it may not be enough, even for the requirements of a single thread.

PHACT memory requirements are greater than 64 KB, so it is not possible for it to solve CSPs using only shared memory. Some tests were made loading some of the data to shared memory and the remaining to global memory. Figure 2.5 presents the elapsed times of PHACT when solving the Costas Array 12 and the 14-queens problems on three GPUs, using only global memory (identified as "G"), or global and shared memory (identified as "GS"). The Costas Array problem consists in placing n dots on a $n \times n$ matrix such that each row and column contain only one dot and all vectors between dots are distinct.

The limited size of the shared memory implies that its usage is only advantageous when it does not limit the number of threads that can be executed simultaneously on an SM. When the amount of shared memory

required limits the number of threads, the performance of the GPU may improve if only global memory is used, because it allows to use more threads per SM, as presented in Figure 2.5.

The three GPUs used were an Nvidia Geforce 980M GTX, an Nvidia Tesla K20c and an AMD Tahiti GPU, identified as Geforce, Tesla and Tahiti in Figure 2.5, respectively. On the three GPUs, solving the Costas Array 12 problem with global and shared memory was faster than using only global memory up to a few threads per SM. However, when the shared memory was not enough to run simultaneously all the threads, the time began to increase.

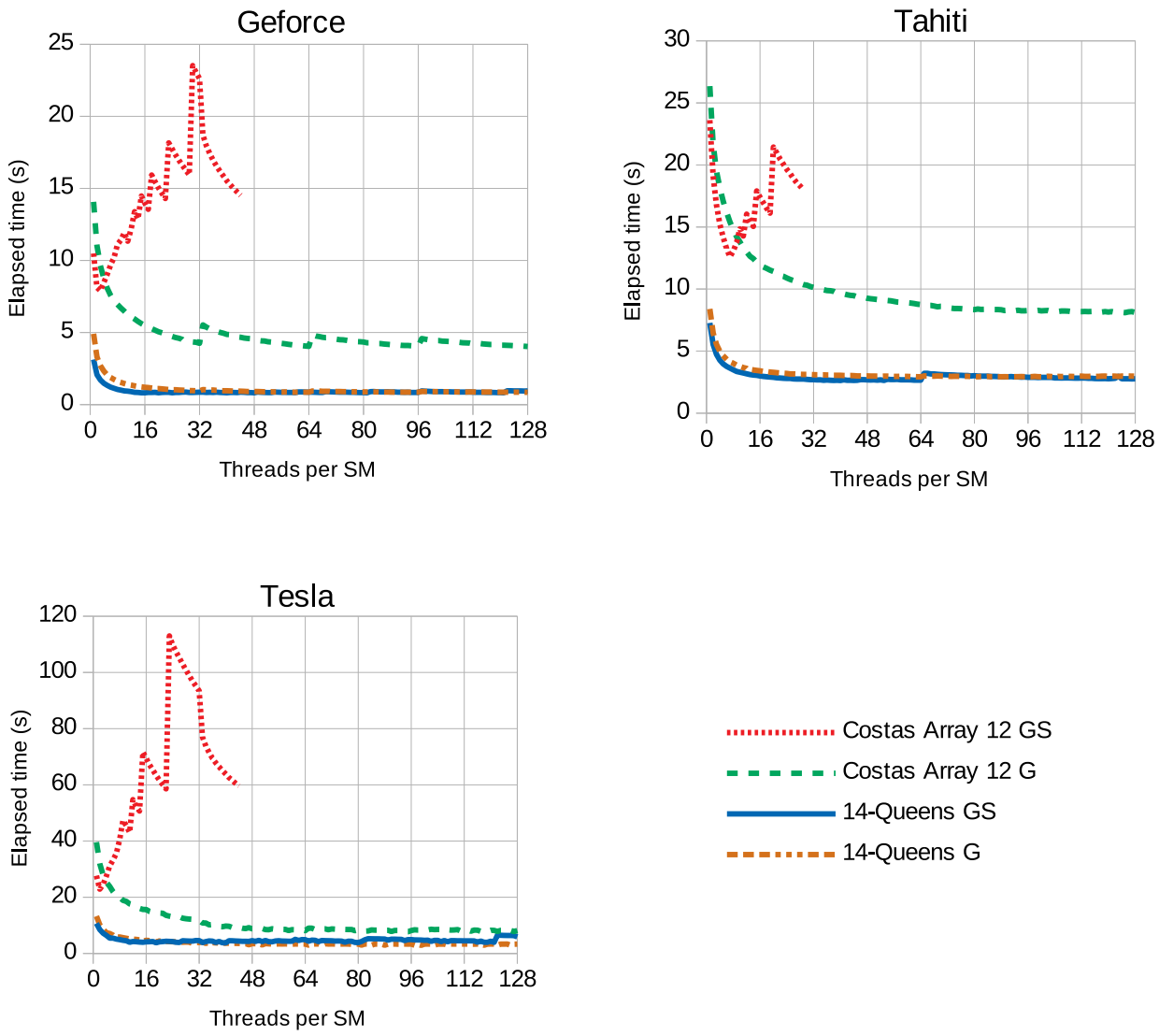

Figure 2.5: Using only global memory (G), or global and shared memory (GS) on three GPUs

On the contrary, using only global memory allowed PHACT to decrease the elapsed time by increasing the number of threads per SM, up to 128 threads, with some gaps on the Geforce. These gaps occur immediately after increasing the number of warps (groups of 32 threads), which leads to running a warp with 32 threads and another warp with the remaining ones, resulting in a performance decrease when the second warp is running. With more than 16 threads per SM, PHACT was faster for solving the Costas Array 12 problem when using only global memory.

The 14-queens problem was represented as a CSP with 14 variables and 273 constraints and the Costas Array 12 problem with 79 variables and 464 constraints, which makes the Costas Array a more complex CSP. This complexity explains the fact that no results were obtained for the Costas Array problem when using more than a certain number of threads per SM, as the shared memory size was not enough for the requested number of threads when using global and shared memory. That limit was reached sooner on Tahiti as its shared memory size is smaller than the one of Geforce and Tesla (32 KB against 49 KB).

As the 14-queens problem requires less shared memory, it allows PHACT to use more threads per SM when using that type of memory. However, less memory accesses allows to decrease the difference between elapsed times when using shared and global memory or only global memory, because although accesses to the global memory are slower than the ones to the shared memory, their small number is not enough to result in a greater variation of the elapsed times. That difference is only barely visible when using one or two threads per SM. When increasing the number of threads, the elapsed times when using only global or global and shared memory are almost the same. Only in Tesla and when using more than 122 threads, an elapsed time increase is noticed when using shared memory, which may be due to the older architecture of Tesla, Kepler [46], when comparing with the Geforce GTX 980 architecture, Maxwell [47].

However, even with Geforce, which was the fastest GPU of those three for solving the Costas Array 12 and the 14-queens problems, if the dimension of the N-queens problem is increased, the gap between elapsed times when using global and shared memory or only global memory becomes visible, as shown in Figure 2.6.

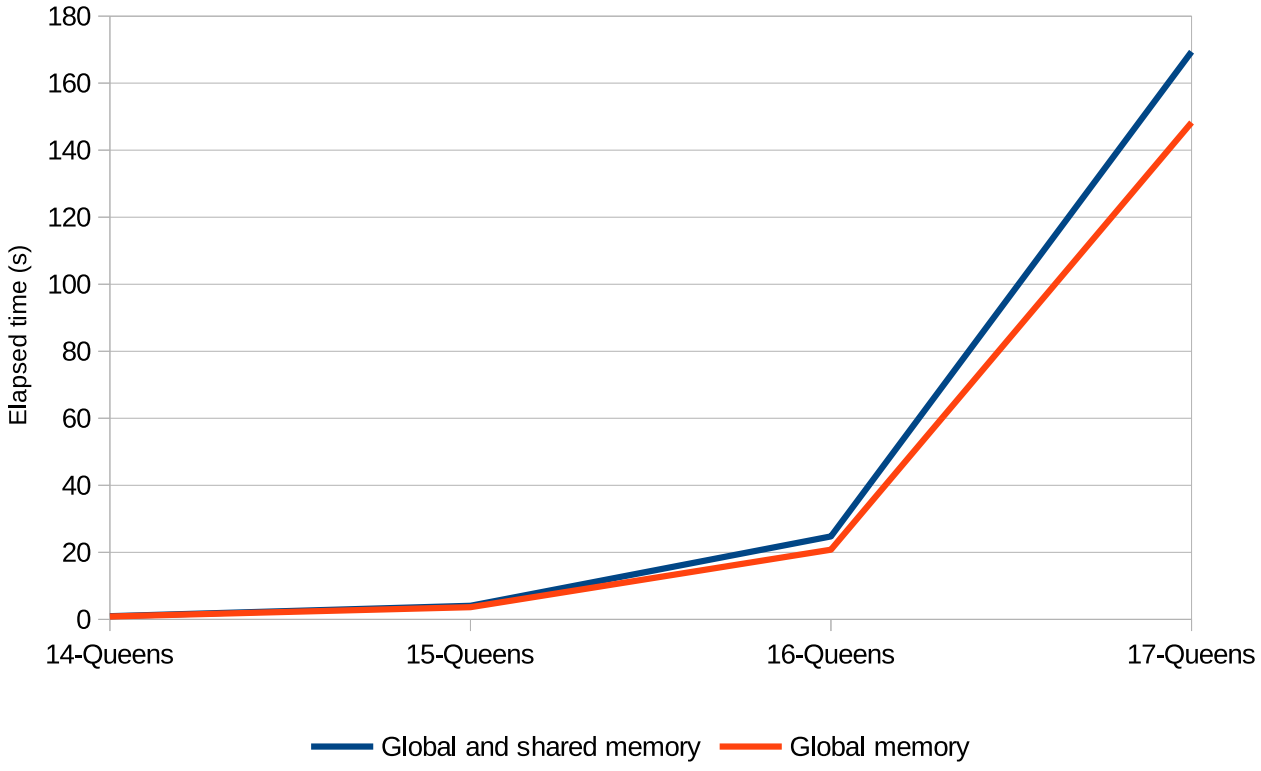

Figure 2.6: Using only global memory, or global and shared memory on Geforce for solving the N-queens problem

The results presented in Figures 2.5 and 2.6 show that, at least for the three GPUs that were used for solving these CSPs, using only global memory will allow to achieve greater performances. The reason for

this behaviour may be due to the amount of shared memory being used leading to bank conflicts when multiple threads need to access the shared memory. According to Nvidia [45], bank conflicts are very expensive and may worsen the GPUs performance when using shared memory instead of global memory.

After several tests of this type, it was decided to not use shared memory in GPUs. However, the CPUs and accelerators used for testing[1] achieved better results when using shared memory, so by default, they will use also shared memory. This may be due to the lower number of threads that these CPUs and accelerators can execute simultaneously, which allow all of them to use the shared memory (32 KB) at the same time.

2.2 GPUs level of parallelism

Several tests were made to try to find the number of threads to use on a GPU that would lead to the best performance of PHACT. Figure 2.7 presents the time needed for PHACT to find all the solutions for several instances of the 10-queens problem. When using a GPU, two levels of parallelism must be defined, the number of threads that will be executed on each SM, which is the same for all the SMs, and the total number of threads that will be executed on the GPU, which must be a multiple of the number of threads that will be executed on each SM.

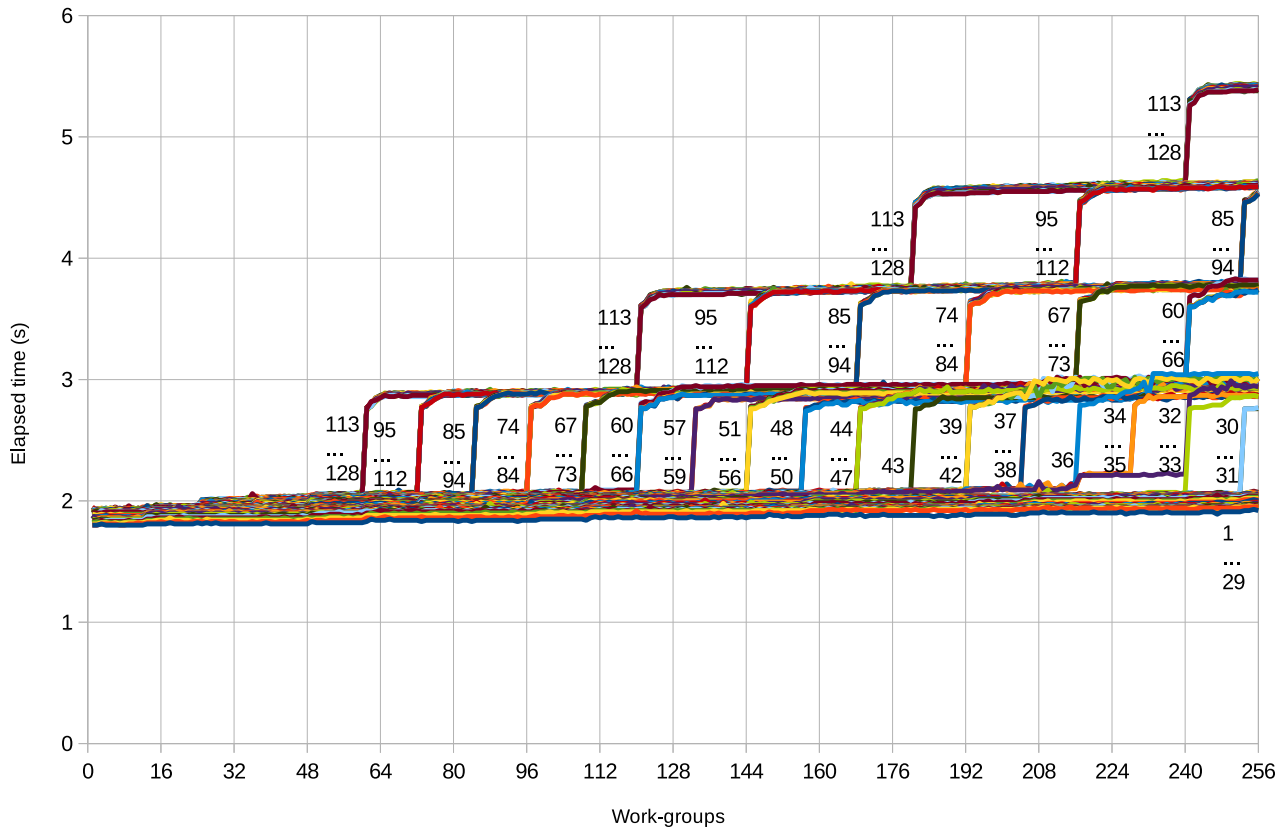

Figure 2.7: Parallelism of an Nvidia Geforce 980M GTX

In Figure 2.7, the numbers near the lines indicate the range of threads per SM that yield that set of lines, as each "step" is composed by as many lines as the range of those numbers. For example, the fine increase

[1] Intel Core i7-4870HQ, AMD Opteron 6376, Intel Xeon CPU E5-2640 v2, Intel Xeon E5-2690 v2 and Intel Many Integrated Core 7120P.

from 2 to 3 s happens when the number of work-groups reaches 61 and between 113 and 128 threads are executed on each SM. The work-groups represent the total number of threads divided by the number of threads executed on each SM, which indicates the number of groups of threads that will be executed on the SMs.

Each thread was counting all the solutions for a copy of a 10-queens problem, allowing all the threads to execute the same code instructions at the same time, which is best suited to take advantage of the GPUs parallelism. The tests ranged from a single thread solving one 10-queens problem, to 32,768 (256×128) threads, each one solving a 10-queens problem.

This chart shows, for example, that this GPU is capable of solving 7,424 (256×29) instances of the 10-queens problem taking almost the same time (about 2 s) as one thread solving a single instance of the problem. The Nvidia GTX 980M possesses 12 SMs which means that it can execute a maximum of 1,536 (12×128) threads simultaneously. However, each SM can support up to 48 warps, which means that up to this number the data of the warps may be stored in the SM resources, making their intercalation much faster. Nevertheless, those values will change according to the amount and type of resources (memory, CUDA core, LD/ST unit, SFU, registers) required for each thread, which may explain the well defined "steps" in Figure 2.7.

The values presented in this chart are only valid for the Nvidia Geforce 980M GTX and for solving copies of a 10-queens problem. If a different GPU or CSP is used, the results will change. For example, an AMD Tahiti GPU is capable of solving 11,264 instances of an 11-queens problem taking almost the same time (10 s) as for solving one instance of the problem. The results are different due to the distinct hardware specifications of the GPUs and to the memory requirements and complexity of the CSPs.

Normally, the more variables and constraints a CSP contains, the more divergent paths must be explored to solve it, making it a more complex CSP. When solving a CSP with multiple threads, each thread will explore one of those paths at a time, leading to most of the threads trying to execute different code instructions at the same time, which greatly decreases the performance of the GPUs, going against their SIMT parallelism model. Nevertheless, more recent GPUs are better prepared to minimize this problem and are improving the usage of their parallelism to speed up this kind of problems.

For example, to count all the solutions for the 12-queens problem using a single thread, the Nvidia Geforce 980M GTX takes 38.7 s, but when using 32 threads on a single SM, it takes 11.6 s. An Nvidia Tesla K20c takes 76.7 s to solve the same CSP with one thread and 28 s to solve it with 32 threads on the same SM. This demonstrates that Geforce achieved an higher speedup than Tesla, 3.3 against 2.7, supporting the statement that newer GPUs like the Geforce are more prepared to deal with threads executing different code instructions.

Figure 2.8 presents the speedups obtained on an Nvidia Geforce 980M GTX, an Nvidia Tesla K20c and on an AMD Tahiti GPU when increasing the number of work-groups with just one thread each, to find all solutions for the Costas Array 12 and the 14-queens problems. The number placed in the chart, after each GPU name, corresponds to the number of SMs each one contains.

In this chart we can observe that the number of work-groups that lead to the best performance depends on the device and on the problem to solve. The speedups increase up to a certain number of work-groups and then they stagnate, meaning that, even if we try to execute more threads on that device, it will not change the time needed to solve the problem. Several tests were made to try to explain this behavior and it was noted that the GPUs limit the number of work-groups that they can handle at the same time. For example, from 4096 work-groups, the Geforce only executes the first 384 to solve the problem and only after these have finished, does it execute the remaining 3,712. In the case of PHACT, this means that all the work is done by the first 384 work-groups and the others will only check that all the work has already

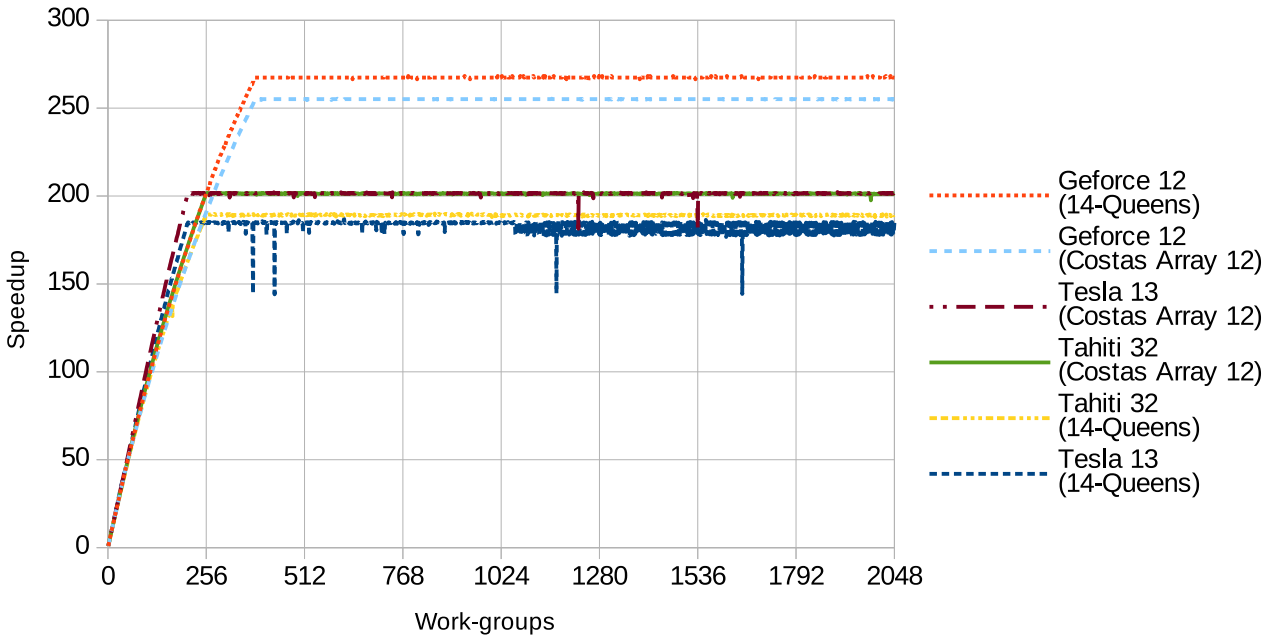

Figure 2.8: Speedups in GPUs when increasing the number of work-groups

been done, and exit.

The 384 value corresponds to the number of SMs (12) of this GPU multiplied by 32. In Tesla, 208 work-groups, which are the work-groups that actually do some work, is equal to the 13 SMs multiplied by 16. However, the Tahiti shares all the work between 384 work-groups, but there is no speedup when increasing their number above 256 work-groups (32 SMs multiplied by 8), which may be explained by the different hardware arrangement when comparing Nvidia and AMD GPUs.

GPUs can choose to pause the execution of a work-group and start executing another when the previous one is waiting for resources [12]. This may mean that for CSPs that require much time to solve, if the number of work-groups created is much higher than the number of SMs it may improve the probability of there existing a work-group that has all the needed resources available. For that reason, by default, PHACT will create 512 work-groups on any GPU to solve a problem.

GPUs allow two levels of parallelism, one related with the number of SMs as presented in Figure 2.8, and another with the number of cores inside each SM as represented in Figure 2.5. The number of threads that a GPU will run simultaneously on an SM is always equal to or less than the work-groups size (number of threads in each work-group). This number is limited by each device architecture, but it will always consist only of threads that will execute the same code instructions next [70]. As so, the number of threads that will be executed simultaneously on an SM is dependent on the complexity of the code. More specifically, on the different paths that a thread may follow in the code.

When using the number of work-groups and threads per work-group that yield the best elapsed times in the previous tests and comparing those to the sequential times (one thread in one work-group), we can observe that the GPUs achieve a considerable speedup, as represented in Figure 2.9.

When comparing those speedups with the ones achieved by CPUs and a MIC, the GPUs greater level of parallelism is much evident. The tests on the CPUs were done with as much threads as their number of

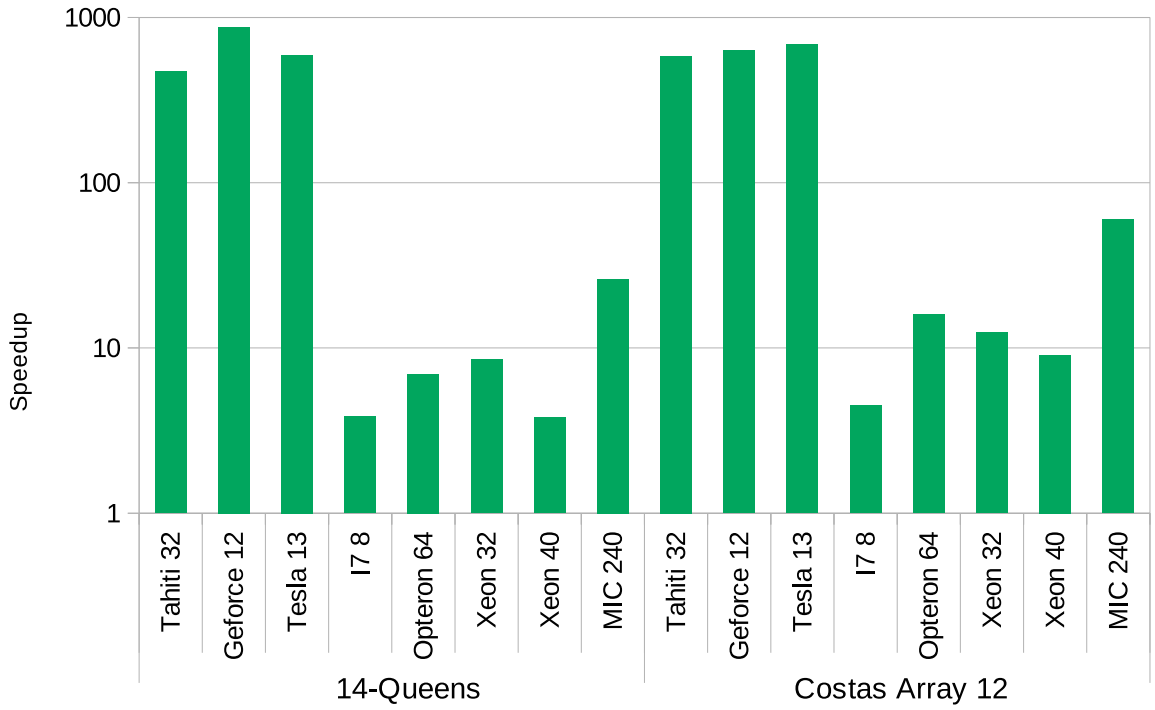

Figure 2.9: Maximum speedups in GPUs and CPUs when comparing with sequential times

cores, and in case of the MICs, 736 threads were used. The numbers next to the name of the device in the chart represent the number of compute units detected by the OpenCL programming language, which in case of GPUs, are the number of SMs, and in CPUs and MICs are equal to or greater than the number of cores. The GPUs used for this comparison were an AMD Tahiti GPU, an Nvidia Geforce 980M GTX and an Nvidia Tesla K20c. The CPUs were an Intel Core i7-4870HQ, an AMD Opteron 6376, an Intel Xeon CPU E5-2640 v2 and an Intel Xeon E5-2690 v2. An Intel Many Integrated Core 7120P was also used.

2.3 Conclusion

The GPUs shared memory may speed up the execution of the code, but its small size does not allow it to be used to solve more complex problems, or it can even worsen the performance of the GPU due to bank conflicts. For that reason, by default, PHACT does not used shared memory when running on GPUs.

GPUs can execute thousands of threads simultaneously, but for that to be possible, the code to execute must possess as little divergent paths, like "if-else" as possible. However, most of the constraint propagation and backtracking algorithms possess many divergent paths, which make constraint solving a hard task for GPUs, specially when the problems are composed by many constraints.

Nevertheless, it was found that when executing code with many divergent paths, if the number of threads is increased above the number of threads that the GPU is capable of running simultaneously, the performance of the GPU will improve. This happens because increasing the number of threads will also increase the chance of existing more threads that will execute the same code instruction at the same time, meaning that they can be executed simultaneously.

By analysing this results and the ones of several other tests that were made to try to select the best default values for the number of work-groups and work-items to use for each device, the following default values were introduced in PHACT:

- CPU or accelerator - 1 work-group with 1 work-item per compute unit;
- GPU - 512 work-groups with 128 work-items each.

Nevertheless, those values are only used if the user does not want to use his own, by introducing them in the execution command of PHACT.

3

State of the art

In the past decades, several constraint solvers and techniques to improve their performance were developed. At the beginning, due to the absence of multithreading CPUs, only sequential solvers were developed, being followed by distributed solvers implemented over networked machines. Later, the multithreading CPUs allowed to speed up the solvers by distributing tasks among the parallel threads. Nowadays, even though the most current GPUs allow unprecedented levels of parallelism on a single device, their capabilities are yet to be tamed for constraint solving.

This chapter introduces some relevant works made in the past, relating to complete search, local or incomplete search, and SAT on CPUs, GPUs and on multiple devices/machines simultaneously. As PHACT is a complete solver, the first section presents this approach, describing most of the techniques used by other authors. Sections 3.2 and 3.3 introduce some techniques used for incomplete search and SAT, respectively.

3.1 Complete search

In complete methods, finding solutions to a CSP is usually done by iterating two stages:

- Labeling - Selecting and assigning a value to a variable. This value must belong to the variable domain;
- Constraint propagation and consistency check - Propagates the assignment of a value to one variable to the domains of other variables. This process aims to reduce the number of elements in those domains and to check if any constraint is no longer satisfiable, which would mean that the set of assigned values is not part of a CSP solution.

The order in which the selection of the variables to label and of the values to assign to them is done, may influence greatly the time taken to solve a CSP. Thereby, several heuristics may be used to make those selections. The most used for selecting the variable for labeling, are:

- Select the first unassigned variable that was introduced in the CSP model. Normally called "Left-most";
- Select the unassigned variable that has the fewest values left in its domain. Normally called "First-fail";
- Select the unassigned variable that is more constrained. Normally called "Fail-first".

Unfortunately, the same heuristic may behave greatly on a CSP, but very poorly on a different CSP, so the selection of the heuristic must be very well thought out, when possible. For example, to count all the solutions for the 15-queens problem with a single thread on an I7 CPU, using the Leftmost (input order) heuristic, PHACT needs 72 s, but when using First-fail it takes only 60 s. On the contrary, for the Costas Array 12 problem, using First-fail, it needs 28 s, but when using Leftmost it needs only 16 s.

When labeling a variable, a value from its domain must also be selected. Several heuristics exist for making this selection, being the most common the one that selects the minimum value. Other possible heuristics are selecting the maximum value or the mid value. This selection also influences the performance of the solver. For the Costas Array 12 problem, PHACT needs 15 s when selecting the maximum value, against 16 s when selecting the minimum value.

When the search is made in parallel, the influence of the heuristics may be attenuated. For example, for counting all the solutions of the 15-queens problem with 8 threads, PHACT took 14 s against 12 s when using, respectively the Leftmost and the First-fail heuristics. However, these results may vary depending on the techniques used for distributing the work to all the threads, and each thread can even use different heuristics.

The process of searching for a CSP solution may be represented in the form of a search tree and the solution is usually searched through backtracking. Backtracking is used when, during the search, an inconsistent state is reached, that is, when an assignment of a variable and/or subsequent propagations cause a constraint to no longer being respected. In this case, all the changes made to the domains of the variables after the last labeling are reverted, the last value assigned to the variable being labeled is removed from its domain, and a new value is selected.

The search tree can be divided into multiple sub-trees which can be traversed in parallel, thereby taking advantage of possible parallel architectures. The division is made by splitting the domains of variables into disjoint domains. Figure 3.1 represents the division of a CSP with three variables ($V1$, $V2$ and $V3$), each one with the values 1 and 2 on their domain, into two sub-search trees.

The division of the search tree representing the problem can be done through different techniques, as for example the one implemented in Parallel Complete Constraint Solver (PaCCS) which is a complete

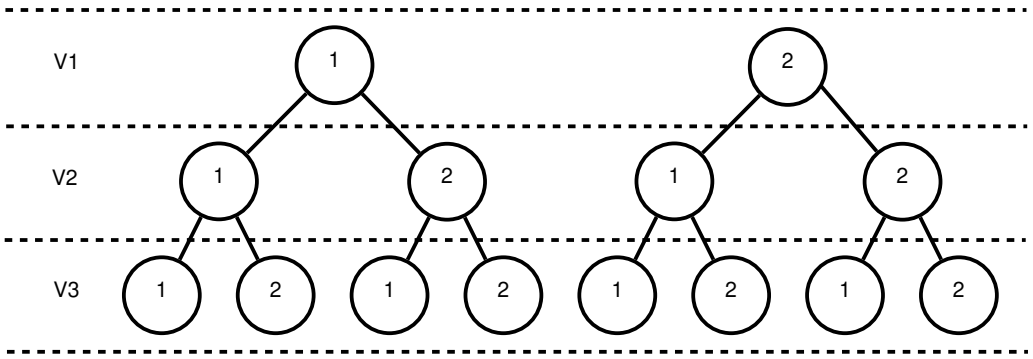

Figure 3.1: Example of a search-tree division into two disjoint sub-search trees

constraint solver developed by Pedro [53], that uses work stealing to distribute the work among multicore CPUs on a distributed environment.

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

According to Gent *et al.* [27], multi-agent search is a promising approach to parallelism usage in CSP solving. This is done by using multiple agents to solve the same problem. Any of the agents is capable of solving the problem independently, or a part of it. Also, the agents may be different and may communicate between them.

The process of work stealing is transparent to the co-workers from which the work is stolen. According to Pedro [53], work stealing is an highly parallelizable load-balancing technique that enables the full use of the power of multiprocessor computers.

PaCCS was implemented for Unix, in the C programming language, with the objective of providing a backend to an higher level language allowing constraint modeling constructs and the transparent usage of multithreading CPUs, possibly in distributed environments. PaCCS uses Portable Operating System Interface (POSIX) threads for easier memory sharing and the Message Passing Interface (MPI) standard to distribute the search space through the workers and for transmitting other required messages.

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

The internal representation of the domains of the CSP variables is called the domain store and was implemented as an array of domains in a contiguous region of memory. Each variable domain was implemented with a fixed-size bitmap, allowing the use of two more fields containing the maximum and minimum values of the domain. Next to each store is included information about the variable whose domain was divided to yield this store. The CSP variables and constraints are stored in shared memory.

According to Pedro [53], the domain store contains all the dynamic information needed to define a search space. As such, when a worker steals work from another worker (teammate or not), one store is the stolen unit. Each worker maintains a pool of stores arranged as an array of stores indexed according to its age and may split its current search space into multiple stores that may be stolen by other workers from the same team or from a different one.

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

both. To achieve this level of distribution, PaCCS is based on a two level architecture:

- Lower level - It represents the teams. Each team is constituted by a group of tightly coupled workers that share resources;
- Higher level - It consists in the coordination of the teams for solving the CSP.

Pedro [53] tested his implementation by solving the N-queens, Langford Numbers, Golomb Ruler and Quadratic Assignment problems. The Langford Numbers problem consists in arranging m sets of integers from 1 to n, such that between two consecutive occurrences of integer i there exist exactly i other numbers. The Quadratic Assignment Problem consists in assigning a set of n facilities to a set of n locations, while minimizing the sum of the distances between locations multiplied by the flow or weight of the facilities.

The results obtained by Pedro [53] showed that PaCCS is a very scalable parallel constraint solver which achieved an almost linear performance for all the tested problems.

Chu *et al.* [18] implemented an adaptive work stealing algorithm that automatically executes different work stealing techniques, according to the estimated solution density (estimated probability of containing a solution) of each sub-tree.

According to Chu *et al.* [18], the efficiency of the branching heuristic that is used is directly related with the efficiency of the achieved load balancing. These authors created an algorithm for estimating the sub-tree solution density that defines the branching confidence of each node. The confidence of a node is the estimated ratio of the solution density in the sub-trees derived from that node.

For estimating the solution density, Chu *et al.* attended to the following properties:

- The solution density between nearby sub-trees is highly correlated because many of the values are shared between them;
- As the number of nodes from a sub-tree that are not the solution increases, the solution density of that sub-tree and of the nearby sub-trees decreases.

Using these properties, Chu *et al.* [18] manage to assign a confidence value to each node while searching for a solution. At the beginning of a search there are no confidence levels for any node. The initial confidence value, ideally, could be developed by the problem modeler, as an expert on the problem to solve. If that is not possible, those values could just be equal in every node, being updated as the search continues.

With a confidence value assigned to every node, Chu *et al.* [18] managed to develop a confidence-based search algorithm with the following features:

- After a sub-tree is fully explored, the confidence value of all the nodes above it is updated;
- The number of threads exploring each branch is updated as the search advances;
- When a worker finishes its sub-tree and needs a new one, it "steals" it according to the following rules:
  - Always start searching for an unexplored node on the root of the search tree;
  - Search for an unexplored node on the left branch if it has fewer working threads than the right branch and taking into account their confidence value, or on the right branch otherwise;

  - If after going down the tree for a certain number of nodes (depth), no unexplored node is found, steal the first unexplored node above that depth (even if it has a smaller confidence value);
  - The maximum allowed depth value is dynamically changed to maintain a minimum sub-tree size, so that work stealing does not occur more often then a predefined threshold to limit communication costs.

- A master coordinates all the workers;
- A worker explores a sub-tree for a predefined restart time and after that time passes, it will return the results of its exploration to the master, and steals a new sub-tree.

Chu *et al.* [18] implemented their confidence-based work stealing algorithm using Gecode [66]. Gecode is a free software library developed to simplify the implementation of constraint-based systems and applications. It is implemented in C++ and has bindings for Python, Prolog, Ruby, Java and other programming languages. Gecode also uses work-stealing when doing parallel search.

Chu *et al.* tested their system with the Traveling Salesman, the Golomb Ruler, the Queen-Armies, the N-queens, the Knights and the Perfect Square placement problems. The Traveling Salesman problem consists in finding the shortest possible route that visits a group of cities. Each city can only be visited once, except for the origin one, that must also be the one to return to at the end [17]. The Golomb Ruler problem consists in defining a set of n marks on an imaginary ruler, such that the distance between each pair of marks is unique between all pairs. The first and the last mark define the size of the ruler, and the ruler should have the minimum size possible [53]. In the Queen-Armies problem two equal-sized armies of queens must be placed on a chessboard, such that no queen from one army attacks a queen from the other army, and the maximum number of queens must be placed [67]. The Knights problem consists in moving a Knight on a chessboard such that he visits every square only once [52].

Those tests were run using eight threads on a Dell PowerEdge 6850 with four 3.0 GHz Dual Core Pro 7120 CPUs and 32 GB of memory. The used restart time was 5 s and the minimum allowed time between work stealing for each thread was 0.5 s.

According to Chu *et al.* [18], even using biased initial confidence values was sufficient to obtain an almost linear speedup, but if the initial confidence values were specifically assigned to try achieving the best results, the outcome was even better. These authors also state that the developed system was capable of achieving a speedup of about 7 using the 8 threads for all the tested problems, and that the communication costs were almost imperceptible.

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.* [63], split the search-space only at the beginning of the solving process.

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

These authors implemented a master, 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. Each worker takes a sub-search space for exploration. When a worker depletes that sub-search space, it will take another one from the queue, until no work remains.

Régin *et al.* [63] tested their technique with the Gecode [66] and the OR-Tools [29] solvers, using 20 problems modeled in FlatZinc [49]. Each problem was split by their implementation and each worker was a thread executing an instance of the solver, which explored 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.

Schulte [64] implemented a distributed solver, using the Mozart OZ programming language [62], which simplifies the usage of networked computers with search engines.

This author based his solver on one master and multiple workers, where the master is responsible for initializing workers, collecting solutions, assist in finding work for workers and detecting termination. Each worker explores the sub-trees, sends solutions to the master and shares work with the other workers. When a worker finishes its sub-tree, it informs the master, which in turn will look for another worker with enough work to share. This worker will then deliver a branch of its sub-tree to the master, and the master will give it to the idle worker.

For benchmarking, the author used the Alpha, the 10-S-Queens, the Photo and the MT 10 problems. The Alpha is an alphabetic puzzle that consists in assigning 26 variables $(a, b, ..., z)$ with distinct values, respecting 25 equations [65]. The 10-S-Queens is a problem identical to the 10-queens problem, but modeled in a distinct way, allowing for a faster exploration [65]. In the Photo problem, a group of persons wants to take a photo while keeping their preferences on the persons next to whom they want to be placed in the photo [65]. The MT 10 is a 10×10 job-shop scheduling problem presented by Muth and Thompson [42].

Schulte [64] tested his implementation on a distributed environment composed by two Intel Pentium II and four Intel Celeron, and achieved speedups that ranged from 1.74 when using 2 workers (two Intel Pentium II) to 5.21 when using 6 workers (all the machines).

The latest versions of the OR-Tools [29] solver used by Schulte [64] are already capable of using parallelism to find one solution or near optimal solutions to a CSP. For that purpose it uses a portfolio of workers with different strategies that try to solve the problem.

Using a similar strategy, namely a portfolio of workers, Prud'homme, Fages and Lorca [55] implemented Choco which is also capable of finding one solution and near optimal solutions when solving optimization problems.

Both OR-Tool and Choco were initially implemented without parallel capabilities, and are capable of enumerating all the solutions of a problem if only one thread is used. Nevertheless, both solvers are under improvement and in the future they may be able to do so also in parallel.

Xie *et al.* [72] developed a constraint based solver for massive parallel systems, such as the IBM BlueGene/L and BlueGene/P, with 65,536 and 1,048,576 processors, respectively. Their parallel solver is based on the C++ constraint programming based Watson Scheduling Library, developed at IBM.

Xie *et al.* implemented a load balancing technique based on dynamically partitioning the search space among the available processors. This was done by dividing processors into masters and workers. The master processors have a global view of the search tree and coordinate the workers by dividing the search tree between them and keeping track of the branches that have been explored or are yet to be explored. Each worker has only one master, but each master is able to coordinate multiple workers.

The workers request a sub-tree from their master and explore it. Each ramification on the search tree corresponds to a constraint added to the tree. A sub-tree (or sub-problem) is passed to the workers as a set of constraints, using the MPI standard.

Each master keeps a representation, called the Job Tree, of the parts of the search tree that have been explored, are being explored or are unexplored by the workers. The unexplored sub-trees are the ones that the masters send to the workers that claim a new sub-problem.

When the solving process starts the search tree is divided between the masters in a static way. For load balancing purposes, a number of possible branchings is evaluated before the search tree is divided. After that division each master creates a Job Tree by exploring some part of his sub-tree. Then, if no solution was found, it starts dispatching branches of its sub-tree to the workers that request them. When a worker, while exploring its sub-tree, notes that it has become too large (more nodes then a predefined threshold), it sends a branch of its sub-tree to its master, and keeps working on the part it did not send. Its master expands its Job Tree with the new unexplored nodes.

After testing their implementation, Xie *et al.* [72] stated that they achieved almost linear scaling with one master. But with multiple masters the results were far from linear scaling. Xie *et al.* state that for achieving that kind of performance with multiple masters, a technique for dynamically allocating sub-problems between masters must be developed.

While solving CSPs through parallelization has been a subject of research for decades, the usage of GPUs for that purpose is a very recent area, and as such there are not many published reports of related works.

Campeotto *et al.* [15] developed NVIDIOSO, which is a complete CSP solver with 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. The division bound is dynamic to keep the load balanced between the host (CPU) and the device (GPU).

For faster access, each CSP variable domain is represented as a bit-mask in a 32-bit unsigned variable, and other three variables are also used for storing the domain bounds and the last event associated with that domain. With this representation, negative numbers can be stored using offset values.

The events allow to classify the last action applied to that domain. For example, they may indicate that an element was removed and this allows to apply the appropriate propagator. Each search tree node is represented in a vector containing all the variables domains and the other three variables mentioned above, allowing to take advantage of the GPU cache.

Data transfers between the host and the device are reduced to a minimum due to the low bandwidth transfer rate. At each propagation, the domains of the variables that have not yet been assigned and the events that 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 his sequential propagation both GPU and CPU will be synchronized.

The simplest propagators are invoked by a single work-group, and the more complex ones are invoked by more than one work-group. 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 work-groups and the ones that should not.

Campeotto *et al.* [15] 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 for some synthetic problems, when comparing a sequential execution on a CPU with the hybrid CPU/GPU version.

3.2 Incomplete search

Local search is an incomplete method for finding a solution for a CSP, and as such it does not guarantee that a solution will be found, if one exists, but it is essential for solving hard combinatorial problems, where a complete method would take too long.

Typically, local search starts with a candidate solution, that is, all the variables assigned with random values, or selected by means of an heuristic. Then, applying specific heuristics it tries to improve that possible solution by changing the assignments. The quality of the candidate solution in normally measured through an evaluation function, which computes the degree of the violation of the constraints, or through a cost set on a global variable.

To improve the candidate solution, the search moves to a local neighborhood by changing the value assigned to one or more variables, repeatedly, until a termination condition is reached. The variables to reassign between iterations are selected by specific heuristics, which are the core of the local search. For example, the *Iterative improvement* or *Hill-climbing* method checks for variables that improve the quality of the candidate solution and selects one of them to be reassigned. It can select, for example, the first one that improves the candidate solution, or the one that most improves it, strategies known as *First-Improvement* and *Best-Improvement*, respectively [61].

The main problem with *Iterative Improvement* is that it tends to stagnate in local minima of the evaluation function. To minimize this problem, several approaches exist, as for example, to randomly select the variables to reassign in some steps of the iterative process, which is called *Random Iterative Improvement* [4]. Another approach to avoid getting stuck in a local minimum is the usage of *Tabu Search*, which consists in memorizing some of the previous neighborhoods that were visited, and avoiding visiting them during the next predefined number of steps [16].

One possible way to improve the performance of local search is to adapt it to the incremental knowledge about the search space obtained while the search is running. Caniou *et al.* [16] developed a parallel Adaptive Search algorithm for solving CSPs, whose main steps are:

1. An heuristic function (also called "error function") is defined for each constraint, which indicates how much that constraint is violated;
2. To each variable is associated an error value that corresponds to the sum of the errors of the constraints (how much the constraint is violated) in which it appears;
3. The variable with the highest error (called the "culprit") is assigned the value from its domain that will result in the littlest error with the next configuration (all variables assigned one value from the respective domain).

Caniou *et al.* [16] used an Adaptive Search method, available for download at [20] as a freeware C-based framework, combined with OpenMPI (Open Message Passing Interface) for parallelization. OpenMPI allows for simple message passing in networked environments, providing the next main features [48]:

- Support for network heterogeneity, allowing the use of multiple types of devices, operating systems and/or protocols;

- Thread safety and concurrency control for safety and consistent resource sharing between threads;
- Network and process fault tolerance to avoid most of the system failures;
- Dynamic process spawning allowing the addition of processes to a running job.

To deal with local minima, these authors used Tabu Search to avoid returning to neighborhoods already visited on a predefined number of previous steps.

For the parallelization, every available core receives a sequential fork of the Adaptive Search method, and runs it for a predefined number of iterations. After that number of iterations a test is made to check if there is any message indicating that a process has found a solution. If a solution has been found, the execution is terminated. If during that amount of iterations, more that one process has found a solution, the fastest one is selected.

Caniou *et al.* [16] used three benchmarks, namely the All Interval Series problem, the Perfect Square Placement problem and the Magic Square problem. The All Interval Series problem can be explained as defining a vector $s = (s_1, ..., s_m)$, such that s is a permutation of $\mathbb{Z}_n = \{0, 1, ..., m-1\}$ and the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, ..., |s_m - s_{m-1}|)$ is a permutation of $\mathbb{Z}_n - \{0\} = \{1, 2, ..., m-1\}$ [3]. The Perfect Square placement problem consists in placing n squares of different sizes inside a master square [39]. The Magic Square problem consists in filling a square grid with distinct integers, such that the sum of each horizontal, vertical and diagonal line of integers is equal [32].

These authors achieved speedups of about 30 to 50, when using 64 and 256 cores respectively, when compared with sequential resolutions, and remark that the speedups were more relevant on bigger benchmarks.

Arbelaez and Codognet [4] developed a constraint-based local search solver that runs an Adaptive Search algorithm similar to the one created by Caniou *et al.* [16], on an Nvidia GPU. These authors implemented parallelism in two levels to take advantage of the GPUs parallel capabilities:

- Multiple instances of the adaptive search algorithm are executed, one per block of threads;
- Inside each block of threads, parallel neighbourhood evaluation is implemented by selecting two variables for evaluation. These variables may be randomly selected or the ones that minimize the global error.

These authors evaluated their implementation with the Magic Square, the Number Partition and the Costas Array Problems. The Number Partition problem consists in dividing a set of n numbers in two groups, such that each group has the same cardinality and their sum and squared sum are also equal, respectively.

When comparing their results with a sequential CPU implementation, these authors achieved speedups of up to 17 in the Magic Square and Number Partition problems, and of up to 3 in the Costas Array problem.

3.3 SAT

The SAT consists in deciding if a solution exists for a propositional formula. SAT is similar to a CSP where the domain of all the variables contains only *true* and *false*, however, as in SAT all the variables are known to have only these two possible values, the propagation of values through (boolean) constraint propagation is normally much simpler, leading to greater performances. In SAT, two concepts are fundamental, the literals that constitute the variables, and the clauses of the formula, each one including one or more literals.

These problems may be solved through incomplete methods like local search, through complete methods like the ones based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [19], and some hybrid solutions also exist [35]. The DPLL is a backtracking algorithm improved for SAT with the addition of two rules in each step:

- Unit Propagation - *Unit Clauses*, that is, literals appearing in clauses containing only one unassigned literal, are set with the value needed for that clause to be true;
- Pure literal elimination - Remove all the clauses that contain a literal that has always the same value.

The application of these two rules can generate new states where the same rules can be reapplied, leading to a faster propagation.

Palù *et al.* [51] developed a SAT solver in the C and CUDA programming languages, capable of using simultaneously a CPU and a GPU. It splits the search space into multiple sub-search spaces that are explored in parallel within CPU and GPU threads that execute the DPLL algorithm [19].

Initially, the CPU loads the problem and executes the DPLL algorithm for a short period. If, after that execution, the number of uninstantiated variables does not surpass a predefined threshold (50 to 80, according to the authors), the GPU will also be used. To reduce the amount of data transferred to the GPU, all the literals set to false and all the clauses satisfied by the current partial assignment are ignored, and only the remaining filtered version of the formula and the unassigned variables are passed to the GPU.

On the GPU, each thread will generate a sub-search space resulting from the assignment of a variable, according to the enumeration of that thread on the GPU. Then, each thread will iteratively run the DPLL algorithm, do propagation, labeling (splitting rule) and backtracking on that sub-search space. If the formula is satisfied, the whole assignment is returned, if not, a flag meaning failure is passed to the CPU.

Palù *et al.* [51] tested their implementation on two machines. One with an Intel Xeon e5645 (12 cores) and an Nvidia Tesla C2075 (448 cores), and another with an Intel I7 and an Nvidia GeForce GTX 560 (336 cores). All the SAT problems that were used, were retrieved from classical examples and related competitions.

The authors stated that their implementation was several orders of magnitude slower than the cutting-edge solvers, but it allowed to demonstrate that the GPUs can be used to achieve great speedups. These authors achieved top speedups of 40 on the first machine and of 15 on the second machine when using the CPU aided by the GPU, comparing with the CPU executions. These authors also pointed out that the tests produced comparable results on the two machines, but using different parameters on the GPUs.

Menouer and Baarir [37] developed an hybrid partition method to try to improve the load balancing between the parallel execution of multiple sequential instances of the Glucose SAT solver [5]. These authors start by splitting the problem in 2^n sub-problems, where n corresponds to the number of most frequent unassigned variables that will be expanded. To calculate n, these authors use the formula $n = \lfloor \log_2 p \rfloor$, where p is the number of cores that will be used.

The sub-problems are then inserted into a Global Priority Queue (GPQ) from where each core will retrieve one for exploration. To improve the load-balancing between the cores, Menouer and Baarir [37] implemented a dynamic partition technique that allows a core that depletes his sub-problem to signal that it is waiting for more work. Then, the core that has the most unassigned variables on its sub-problem at the moment, which is indicative of being the core with most work remaining, will split the sub-problem in two, placing one half in the GPQ and proceeding with the exploration of the other half. After this, the waiting core will pick the new sub-problem from the GPQ.

Menouer and Baarir [37] executed their implementation with 27 SAT and 17 Unsatisfiable SAT (UNSAT) problems retrieved from the SAT Competition 2015 [40] on an Intel Xeon X5650, capable of executing 12 threads simultaneously.

When using the hybrid partition technique with one sequential execution of Glucose per each of the 12 threads available, these authors achieved average speedups of 2.8 with the 17 UNSAT problems and of 16.7 with the 27 SAT problems when compared with the static partition method.

3.4 Conclusion

Several constraint solvers are already capable of using from 1 thread to multiple threads, or even many multi-threaded CPUs on different machines to solve a CSP. These solvers use techniques like work-stealing and work-sharing to distribute the work among all the threads. Some of them can even achieve almost linear speedups when using more threads to solve a CSP. However, at the present date, there does not seem to exist a constraint solver capable of using CPUs, GPUs and MICs to solve a constraint problem, which is the gap PHACT tries to fill.

4

Solver architecture

PHACT is a complete 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, to speed up solving constraint problems. It splits the search space of a problem in multiple disjoint sub-search spaces by partitioning the domains of the variables. Then, it distributes groups of sub-search spaces to each one of the devices, where each thread will explore one sub-search space at a time.

The solver is composed of a master process which loads the CSP, collects information about the devices that are available on the machine, such as the number of cores and the type of the device (CPU, GPU or MIC), and calculates the number of sub-search spaces that will be created to distribute among those devices.

After loading the CSP and collecting information about the devices available on the machine, PHACT uses one thread of the CPU to try to simplify the CSP. This step of filtering tries to remove values from the CSP variables that can not be part of any solution, and may also remove constraints that are already respected with all the combinations of values from the variables that they constrain.

Then, for each device there will be one thread (communicator) responsible for communicating with that device, and 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 will depend on the number of cores and type of that device, and may vary from 8 on a hyper-threaded Quad-core CPU to more than 50,000 on a GPU.

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

4.1 Framework

PHACT provides its own interface for implementing CSPs, through a set of methods to create variables and constraints, and the ability to define the search goal, that is, if the objective is optimization, counting all the solutions or finding one solution. The solver is also capable of loading some of the MiniZinc and FlatZinc models to allow it to process/solve CSPs already modeled in these constraint modeling languages. MiniZinc is an high level constraint modeling language, that may be converted into FlatZinc which is readable by many constraint solvers [43].

PHACT includes a FlatZinc interpreter which is based on the flex/bison interpreter distributed with Gecode [66]. For loading MiniZinc models, it uses the "mzn2fzn" tool [49] that converts the MiniZinc model to FlatZinc, which is then loaded by PHACT through its own FlatZinc interpreter.

The FlatZinc interpreter for PHACT is yet under development, but is already capable of recognizing most of the FlatZinc specification predicates [7]. However, currently, PHACT only works with CSPs whose variables domains are composed only by positive integers, and will not work with floats or negative values, nor with sets. In some cases, the negative integers can be replaced with positive integers, by applying an offset to the values. Boolean variables, at is, CSPs variables whose domains only contain 0 and 1, are treated as integer variables.

The FlatZinc interpreter and PHACT are implemented in the C programming language, but the search engines that run on the devices and the communication and control of these devices is implemented through OpenCL C [41]. Currently, the most used programming languages for implementing constraint solvers capable of using GPUs are CUDA from Nvidia and OpenCL from the Khronos Group [15, 14, 4, 51]. Although CUDA is only compatible with the Nvidia GPUs, OpenCL is compatible with most of the current CPUs, GPUs and MICs from different vendors, including Nvidia [33].

OpenCL allows PHACT execution on multiple types of devices from different vendors and the capability of being executed on Linux and on Microsoft Windows.

4.1.1 OpenCL

The OpenCL framework comprises a programming language based on the C99 standard and a set of Application Programming Interfaces (APIs) to define and control compatible devices. Each device vendor is responsible for building their own OpenCL compiler and drivers to allow their devices to execute software implemented in the OpenCL programming language.

The OpenCL programming language extends the C99 standard with some features intended to facilitate the implementation on heterogeneous parallel systems, and removes other features, as for example the possibility of dynamic memory allocation.

Some OpenCL concepts must be introduced, in order to better understand the architecture of PHACT:

- **Compute unit** One or more processing elements and their local memory. In Nvidia GPUs each SM is a compute unit. AMD GPUs have their own components called Compute Units that match this definition. For CPUs and MICs, the number of available compute units is normally equal to or higher than the number of threads that the device can execute simultaneously [41];
- **Host** CPU where the application responsible for controlling and communicating with the devices is run;
- **Device** A device compatible with the OpenCL framework. The OpenCL distinguishes the devices in three types, CPUs, GPUs and accelerators like the Intel MIC;
- **Kernel** The code written in OpenCL that all the work-items inside a device will execute;
- **Work-item** An instance of the kernel (thread);
- **Work-group** Composed of one or more work-items that will be executed on the same compute unit, in parallel. All work-groups for one kernel on one device have the same number of work-items.

OpenCL defines four types of memory (address spaces) that, if used properly according to the devices, may improve the software performance. The four types of memory are:

- **Global** The device RAM. It is used for transferring data between the host and device and is accessible to all the work-items of the device. It may be used for read and write operations;
- **Local** Depending on the devices, it may consist on cached Dynamic Random Access Memory (DRAM) on CPUs, and on local memory in GPUs. It is faster than Global memory, but its size is normally around 32 Kb to 48 Kb. It may be used for read and write operations, and is shared among all the work-items of the same work-group;
- **Constant** A portion of RAM (about 64 Kb to 128 Kb) marked as read-only, that may be cached to improve its performance. Is accessible to all the work-items of the respective device;
- **Private** Consists of the device architectural registers for the instruction set and is accessible to a single work-item for read and write operations.

Local memory is very fast and may improve the software performance when used instead of global memory. However, if its small size is not enough for the requirements of all the work-items of the same work-group, it may worsen the performance instead of improving it, as demonstrated in Section 2.1.

These many types of memory add a new complexity level for programmers, and some choices must be made. For example, local memory can be used for speeding up the kernel, but due to its small size, data will normally be split among local and global memory. However, a data structure in one type of memory cannot contain pointers to any data stored in a different type of memory. In constraint solving, if we decide to store all the information about the variables in local memory, and the local memory size will not allow to also store the constraints, we will need to store the constraints in a different type of memory. This will influence all the code, because variables will not be able to have pointers to the constraints that constrain them, and constraints will not be able to have pointers to the variables that they constrain.

PHACT is capable of using simultaneously all the four memory types. For that purpose, pointers between data in different memory types were replaced by indexes of vectors as exemplified in Example 1.

Example 1 *Replacing pointers between different memory types in OpenCL.*

- *Variable $V5$ is stored in position 5 of a vector of variables stored in local memory;*
- *Constraint $C9$ is stored in position 9 of a vector of constrains stored in constant memory;*
- *The constraint $C9$ constrains the variable $V5$;*
- *$C9$ will have a field with the value 5, indicating that it constrains the variable stored in the fifth position of the vector of variables;*
- *$V5$ will have a field with the value 9, indicating that it is constrained by the constraint stored in the ninth position of the vector of constraints.*

Using indexes of vectors instead of pointers requires more code instructions than if pointers were used. However, the speedup caused by using different types of memory is greater than the overhead caused by the added code.

The usage of different types of memory also adds a new complexity when coding functions, as the same function will not be able to receive the same arguments from different memory types. This will obligate the programmer to implement the same function with a different name as many times as the combinations of memory types of the arguments that will be used. Example 2 shows the implementation needed for using the "same" function with an argument from local or global memory:

Example 2 *Creation of the "same" function for arguments with different memory types.*

```
int function_l(__local int variable) {...}
int function_g(__global int variable) {...}
```

The absence of dynamic memory allocation in OpenCL forces the amount of memory that will be required for running a kernel inside a device to be defined prior to running the kernel. Also, the amount of memory can not be extended during the execution of the kernel, which means that it must be enough to fit the maximum requirements, even if for the most of the CSPs, the full amount will never be used. This means that PHACT will always be running in the worst case scenario in matters of memory requirements for the devices.

When looking for one solution, or for the best one, the output of PHACT will have a solution, if one exists. When counting all the solutions, the output will be the number of solutions that were found. However these solutions will not be printed due to some OpenCL limitations. Namely, the OpenCL does not possess concurrency control between devices which would lead to the output of the "printf" of the devices to be mixed, making them unreadable. Also, some devices have a buffer where the contents to be printed are stored during the execution of the kernel, and that content will be printed only when the kernel ends its execution. However, for some devices that buffer has only 1 MB of size, which is not enough to store all the solutions for some problems. Nevertheless, PHACT may try to print all the solutions if "-PRINT-SOLUTIONS" is enabled as described in Appendix B, which is only available when using one thread per device.

Although OpenCL 2.2 is already available, most of the existent devices, specially GPUs, are only compatible with OpenCL up to version 1.2, which lead to the choice of using the 1.2 specification in PHACT. This allows PHACT to be compatible with older and newer devices, but it does not allow it to take advantage of the new features added with version 2.0. For example, OpenCL 2.0 introduces the generic address space which removes the need of implementing the same function multiple times, as demonstrated in Example 2.

OpenCL 2.0 also introduces the possibility of sharing pointers between the host and the device, allowing them to share data while kernels are running and without explicit data transferring [34].

In the implementation of PHACT described here, the master process and the threads responsible for communicating with the devices (one per device), run on the OpenCL host and the search engines run on the devices. The OpenCL host may also be treated as a device, in which case it will be simultaneously controlling and communicating with the devices and running search engines. Each search engine corresponds to a work-item, and all work-items execute the same kernel code, which implements the search engine.

4.2 Search space splitting and work distribution

For distributing the work between 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 of one or more of the variables of the problem, so that the resulting sub-search spaces partition the full search space.

Example 3 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 3 *Creation of 4 sub-search spaces with balanced domains*

$$
\begin{aligned}
SS1 &= \{V1 = \{1\}, V2 = \{1\}, V3 = \{1, 2\}\} \\
SS2 &= \{V1 = \{1\}, V2 = \{2\}, V3 = \{1, 2\}\} \\
SS3 &= \{V1 = \{2\}, V2 = \{1\}, V3 = \{1, 2\}\} \\
SS4 &= \{V1 = \{2\}, V2 = \{2\}, V3 = \{1, 2\}\}
\end{aligned}
$$

A more complex example is presented in Example 4 where the domain of $V1$ is $\{1, 2, 3\}$ and the remaining variables have the same domain as in Example 3.

Example 4 *Creation of 4 sub-search spaces with unbalanced domains*

$$
\begin{aligned}
SS1 &= \{V1 = \{1, 2\}, V2 = \{1\}, V3 = \{1, 2\}\} \\
SS2 &= \{V1 = \{1, 2\}, V2 = \{2\}, V3 = \{1, 2\}\} \\
SS3 &= \{V1 = \{3\}, V2 = \{1\}, V3 = \{1, 2\}\} \\
SS4 &= \{V1 = \{3\}, V2 = \{2\}, V3 = \{1, 2\}\}
\end{aligned}
$$

The number of sub-search spaces that will be created is limited to a maximum of 1,000,000 and depends on the type and number of devices that will be used:

- 1 CPU - 5,000 sub-search spaces per compute unit;
- 1 GPU - 500,000 sub-search spaces;
- 1 accelerator - 250,000 sub-search spaces;
- 1 CPU and other devices - 5,000 sub-search spaces per each CPU compute unit;
- Multiple devices but no CPU - 500,000 sub-search spaces per device.

When using one or more devices that are not CPUs, with a number of threads different from the default one, the number of sub-search spaces to create is proportional to the amount of threads that will be used. A GPU by default will use 512 work-groups and 128 work-items, which correspond to the creation of 500,000 sub-search spaces. For example, if the number of work-groups and work-items is specified to 256 and 32, respectively, 62,500 sub-search spaces will be created ($256 \times 32 \times 500,000/(512 \times 128)$).

When expanding the search space to create sub-search spaces, only the variables marked for labeling are expanded. Thus, the total number of sub-search spaces that will be created depends on the number of values on their domains, and if they are not enough, the number of sub-search spaces will be less than desired. If they are enough, the number of sub-search spaces that will be created will always be equal to or greater than the required amount, because the expansion of the sub-search tree is always done down to the same depth level in all branches, as represented in Figure 4.1, to allow their faster generation in the devices.

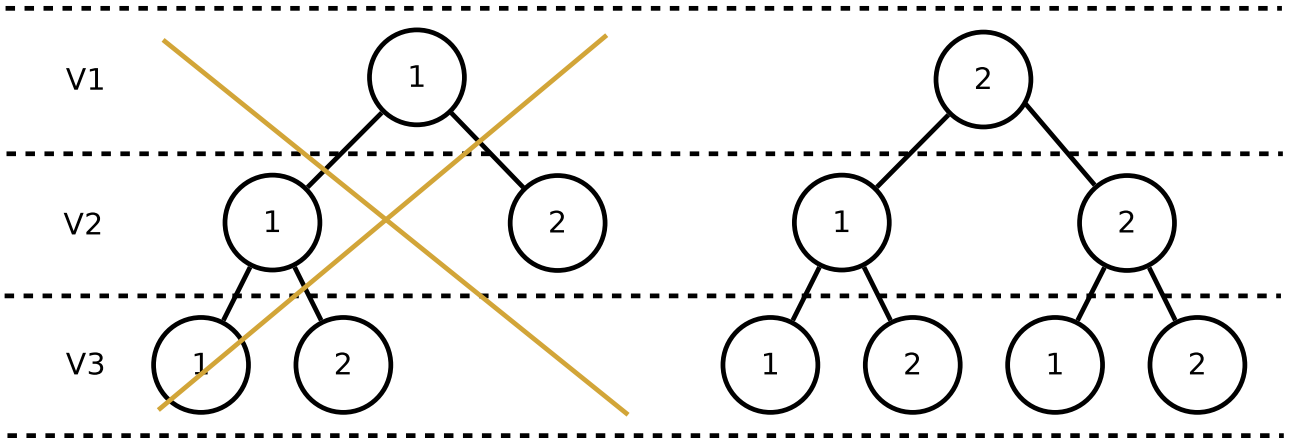

Figure 4.1: Expanding the search tree to the same depth level in all branches (right figure)

When expanding the search space to create the sub-search spaces, no constraint propagation is done, which means that in some cases those sub-search spaces may be already inconsistent. For example, if the variables $V1$ and $V2$ of Example 3, were constrained by a "not equal" constraint ($V1 \neq V2$), the sub-search spaces $SS1$ and $SS2$ are already inconsistent. That verification is ignored in the host, but done in the devices, to greatly reduce the size of data transfer between the host and the devices, as it will be described in Section 4.5.

Since each 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 compose each block will vary during the solving process and depends on the performance of each device on exploring the previous blocks.

The communicator threads running on the host launch the execution of the search engines on the devices, calculate the size of the blocks of sub-search spaces to hand 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.

Depending on the device that is being controlled by the communicator threads and on the version of the OpenCL drivers, on some occasions, the communicator thread will be using 100% of the CPU thread when waiting for the device to finish solving the CSP. On these occasions, the CPU will be using one more thread

than desired and that thread will not be available to solve the CSP when on of the devices to use for solving it is also the host.

Figure 4.2 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.

4.3 Load balancing between devices

An essential aspect to consider when parallelizing some task is the balancing of the work between the parallel components. Creating sub-search spaces with balanced domains, when possible, is no guarantee that the amount of work involved in exploring each of them is even 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 good, work distribution.

Achieving effective load balancing between devices with such different architectures as CPUs and GPUs is a complex task [31]. 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 [26], and the execution of a kernel can not be paused or stopped. Hence, techniques like work stealing [18, 54], 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 reduce the amount of work it sends to the devices each time, by reducing the number of sub-search spaces in each block. This would make the devices synchronize more frequently on 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.

If only one device is to be used, that device will get a single block composed by all the sub-search spaces. When using more than one device, PHACT implements a dynamic load balancing technique which adjusts the size of the blocks of sub-search spaces to the performance of each device solving the current problem, when compared to the performance of the other devices. It also uses different techniques if finding all the solutions, only one solution or optimizing.

If more than one device d is to be used, the number of sub-search spaces that will compose the first block that will be sent to each device is calculated according to the number of threads that each device is capable of running simultaneously, its clock speed and the type of device. All this data can be retrieved with specific calls to the OpenCL API which groups all the devices in three types - CPU, GPUs and accelerators (ACC) like Intel MICs.

After retrieving the number of compute units, the clock speed and the type of device, PHACT assigns each device a value ranging from 0 to 1, where the closer to 1, the faster the device is. The sum of the calculated values is always 1, making it a relative speed between all the devices that will be used. Their relative speed, $s(d)$ is estimated through Equation (4.1):

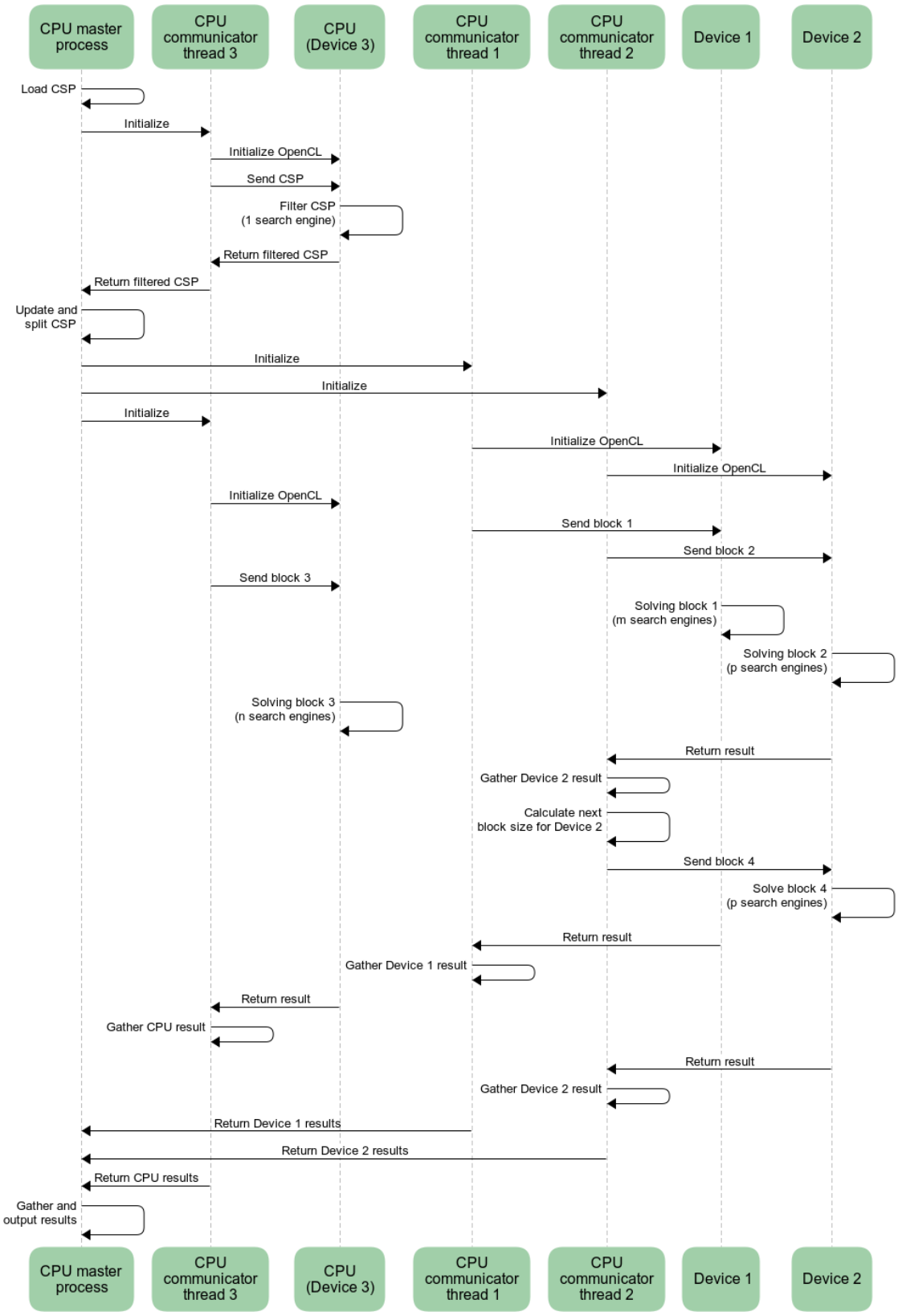

Figure 4.2: PHACT components and execution example

$$s(d) = \frac{\frac{f(d) \times c(d)}{n(d)}}{\sum_{i=1}^{m} \frac{f(i) \times c(i)}{n(i)}} \tag{4.1}$$

In Equation (4.1), d is the device for whom s is being calculated, f is the maximum clock frequency of the device cores, c is the number of compute units of the device, n is the number of times that a core of device d is estimated to be slower than a CPU core and m is the total number of devices. The values of n were obtained through empirical methods, but those values may be inaccurate for some combinations of devices. The default values of n are 8 for GPUs and 4 for ACCs.

4.3.1 Finding all the solutions

When finding all the solutions, s is only used to calculate the size of the first three (small) blocks of sub-search spaces that each device will explore at the beginning of the solving process. For that purpose, s is multiplied by 25% of the number of sub-search spaces that were created.

By using s to calculate the size of those blocks, PHACT avoids sending a block that would take too long to solve on a very slow device, which would lead to the other devices finishing to explore the remaining blocks much earlier than it would take the slow device to explore those three first blocks.

For some CSPs, the first block of sub-search spaces created through the method exemplified in Examples 3 and 4 are trivially inconsistent and as so, they are explored very fast, resulting in the device that explored them being incorrectly catalogued as a very fast device. That would lead to an oversized block being assigned to that device in the next iteration, which could result in that device finishing to explore that block much after the other devices deplete all the remaining sub-search spaces. To diminish the probability of this problem occurring, instead of one small block, three small blocks are explored.

The second and third blocks will have the same number of sub-search spaces as the previous one, except if the device took more than 1 s to explore it, in which case the next block will have half the size of the previous one.

The first device to finish exploring those three blocks is probably the fastest device, so the next block will be composed with twice the number of sub-search spaces of the previous one. But if that number is greater than 20% of the remaining sub-search spaces, the next block will have the same size as the previous one.

When all the devices finish exploring those first three blocks, their *rank*, $rank(d)$ is calculated according to Equation (4.2), where m is the total number of devices. The value of the *average time*, $avg(d)$ of a device is the result of dividing the total time that the device was exploring sub-search spaces by the total number of constraint propagators executions.

$$rank(d) = \frac{\frac{1}{avg(d)}}{\sum_{i=1}^{m} \frac{1}{avg(i)}}, \quad avg(i) > 0 \tag{4.2}$$

Similar to s, the rank of a device consists in a value between 0 and 1, corresponding to the relative speed of the device against all the devices that were used for solving a block of sub-search spaces. Faster devices will get an higher rank than slower devices, and the sum of the ranks of all the devices will be 1. The rank is then used to calculate the size of the next block of sub-search spaces to send to the device, by

multiplying its value by a percentage of the number of sub-search spaces that are yet to be explored. That percentage is different according to the type of device. It will be 0.8 for CPUs, 0.6 for accelerators and 0.3 for GPUs.

Those percentages differ between types of devices, because it was noted that the time needed for a GPU to explore a block of sub-search spaces is much more affected by the work needed to explore it than CPUs or accelerators. Thus, those percentages are intended to prevent slow devices from dominating the solving process.

As search progresses, every time a device finishes exploring another block, its average time and rank are updated. Table 4.1 exemplifies the usage of rank for calculating the number of sub-search spaces that will compose the block which will be sent for each device as soon as each of them finishes its previous block. This is repeated until a device is estimated to take less than 2 s to solve all the remaining sub-search spaces, in which case that device will receive a last block with all the remaining sub-search spaces for exploration.

Table 4.1: Example of the calculation of blocks size when using three devices

Device	Average time per propagation (ms)	Rank	Remaining sub-search spaces to explore	Size of the next block of sub-search spaces
CPU	0.00125	0.793	1,233,482	782,555
GPU	0.01236	0.080	450,927	10,850
ACC	0.00782	0.127	440,077	33,472

Note that the values of the second and the third rows of the column "Remaining sub-search spaces to explore" are the result of subtracting the value of the "Size of the next block of sub-search spaces" from the previous row from the value of the "Remaining sub-search spaces to explore" also from the previous row, as it would occur if these devices finished their previous blocks of sub-search spaces in that chronological order.

4.3.2 Optimizing

When solving optimization problems, the best solution is required. To identify the best one, a value, normally named *cost*, classifies each one of the solutions. When working with more than one device, each time a better solution is found, the cost must be updated between the devices. However, with OpenCL, when a device finds a better solution, it cannot update the cost that the other devices are currently using. Also, with OpenCL we cannot pause or stop a device that is exploring a block of sub-search spaces to make it update the cost. This means that even if a device finds a better solution and it updates its own cost, the other devices will keep looking for solutions based on the cost they know, that may be worse than the ones already found by the other devices.

Only when a device finishes exploring its block, it can communicate to the host the new better cost it has found or update its own cost with the new one found by another device. This may lead to unnecessary exploration efforts being done by the devices, until they finish their block and resynchronize with the host. To reduce the unnecessary exploration efforts being done by the devices looking for solutions already worse than the one found by another device, the size of each block of sub-search spaces is reduced to make the devices synchronize more frequently with the host. However, by reducing the number of sub-search spaces that compose each block, more blocks will have to be communicated from the host to the devices, which leads to more time spent in communications between the host and the devices and initializing data

structures on the devices.

Similar to when searching for all the solutions, the number of sub-search spaces that will compose the first block that will be explored by each device is obtained by multiplying s by a percentage of the total number of sub-search spaces that were created. But, to reduce the size of the first block, instead of 25%, only 1% is used.

The number of sub-search spaces that will compose the next blocks will remain the same until each one of the three previous blocks takes less than 2 s to explore, in which case the next block will have 20% more sub-search spaces. This is done considering that when a better solution is found, the time needed to explore the next sub-search spaces is normally reduced due to the propagation of the new cost. This allows PHACT to dynamically adapt to the new work-load of each sub-search space.

Considering the three previous blocks instead of one or two, decreases the chances of increasing the size of the next block due to an abnormal block that was too easily explored. However, if a block takes more than 2 s to explore, the next block will have half the number of sub-search spaces, as similar sub-search spaces that take longer to explore may be found in the next blocks, leading to an increased time needed to explore them. This strategy is repeated until all the sub-search spaces have been explored.

4.3.3 Finding one solution

Similar to when solving optimization problems, the number of sub-search spaces that will compose the first block that will be explored by each device is obtained by multiplying s by 1% of the total number of sub-search spaces that were created.

Sending smaller blocks of sub-search spaces when searching for a single solution allows the devices to synchronize more frequently with the host to check if any device has already found a solution. However, that may increase the number of blocks that will be communicated to the devices, which for GPUs are very time consuming operations.

Considering those two factors, PHACT tries to deliver to each device, blocks of sub-search spaces that take about 2 s to solve, each. For that purpose, the host keeps a record of the average time each device needed to solve one sub-search space, considering the time spent exploring blocks of sub-search spaces and the total time taken to explore them.

The first blocks will have the same number of sub-search spaces until all the devices have explored at least three blocks. This is done to better estimate the average time each device need to solve one sub-search space, prior to using this value to calculate the number of sub-search spaces needed to occupy the respective device for about 2 s.

After that, when a device finishes exploring a block without finding a solution and none of the other devices has already found a solution, it will receive a new block with the number of sub-search spaces resultant from dividing 2 s by the average time that the respective device needed to solve one sub-search space. If the previous block of sub-search spaces took more than 4 s to solve, the next block will have half the number of sub-search spaces.

This process is repeated until a device finds a solution and all the devices finish their current block of sub-search spaces, or if the CSP in unsolvable, until all the sub-search spaces have been explored.

4.4 Load balancing inside a device

Another challenge GPUs pose is that they achieve the best performance when running hundreds or even thousands of threads simultaneously. But to use that level of parallelism, they must have enough resources and work to keep that many threads busy. Otherwise, when a GPU receives a block with less sub-search spaces than the number of threads that would allow it to achieve its best performance, the average time needed to explore one sub-search space increases sharply.

For example the Nvidia GeForce GTX 980M takes about 1.1 s to find all the solutions for the 13-queens problem when splitting the problem in 742,586 sub-search spaces, and 2.4 s when split in only 338 sub-search spaces. This challenge is also valid for CPUs, but not so problematic due to their lesser degree of parallelism when compared with the GPUs.

To overcome that challenge, sub-search spaces may be further divided inside a device, by applying a multiplier factor m to the size of a block and turning a block of sub-search spaces into a block with m times the original number of sub-search spaces. For example, applying a multiplier factor of 2 to $SS1$ of Example 3 would result in the 2 sub-search spaces presented in Example 5.

Example 5 *Result of applying a multiplier factor of 2 to the first sub-search spaces from Example 3.*

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

Without this technique, to achieve a good occupancy rate when using more than one device to solve a CSP, more sub-search spaces would need to be created. Increasing the number of sub-search spaces could lead to a greater number of blocks of sub-search spaces and thus, to a decrease in performance due to the increased communications between the host and the devices. More sub-search spaces may also lead to more propagations needed to solve the same CSP, because the first variables whose domains are expanded to create the sub-search spaces may be assigned with the same values in more than one sub-search space. This means that the same propagations will be done in different sub-search spaces, what may not happen if less sub-search spaces were created.

For example, propagating the values of $V1$ and $V2$ in the sub-search spaces $SS1$ and $SS2$ of Example 5 will have the same result as propagating $V1$ and $V2$ of only $SS1$ from Example 3, but twice the number of propagations are needed for the sub-search spaces from Example 5.

Applying a multiplier factor inside the devices, when needed, allows the number of sub-search spaces that is created in the host to remain the same as if only the CPU was to be used, despite the number of devices that will be used. This also allows to decrease the overhead added to the CPU (host) when more devices are used as the multiplier factor is only applied inside the devices that receive blocks of sub-search spaces to explore that are very small. Blocks with a few sub-search spaces are only sent to devices whose rank is very small, or at the end of the solving process, when the sub-search spaces yet to be explored are almost depleted.

The value of the multiplier factor will be the one needed to match the number of sub-search spaces that the respective device would receive if it was the only device being used. Those values are defined in Section 4.2. When a device receives a block of sub-search spaces, it will also receive the value of the multiplier factor, and each search engine will use that information to generate the next unexplored sub-search space from that block, as it will be described in the next section, and explore it. All the search engines, that is, the threads exploring sub-search spaces, will execute the same source code, independently of the device on which they will be executed. It will do labeling, constraint propagation and backtracking on that sub-search

space, until fully exploring it or finding a solution if only one is wanted. After depleting that sub-search space, the search engine will generate a new one. This process is repeated by each search engine until all the sub-search spaces of the block have been explored, or a solution is found if only one is wanted.

Unbalanced sub-search spaces may cause some search engines to finish their exploration work much before the others. This is problematic when the sub-search spaces are depleted, which may cause a few search engines to remain working on their last sub-search space while all the others have already finished. To try to overcome this problem, PHACT implements a work-sharing technique that allows some search engines to share some work with other search engines when the sub-search spaces are depleted. For that purpose, a pool of sub-search spaces is created inside each device. It has the capacity needed for as many sub-search spaces as the number of compute units of the CPU or accelerator, or the number of compute unites multiplied by the number of work-items that compose each work-group for GPUs.

Each time a search engine labels a variable, and the pool of sub-search spaces for work-sharing is not full yet, it does two labelings, the second of which will be copied to the pool of sub-search spaces for work-sharing with the current domains of the remaining variables. This will be done by each search engine, until the pool is full. When a search engine tries to collect a sub-search space from the block of sub-search spaces and notes that the block is already depleted, it will try to get one from the pool of sub-search spaces for work-sharing. Due to the heavy costs of using concurrency control to allow this technique, the pool of sub-search spaces is only filled once. Although this technique allowed to achieve a finer load balancing when terminating the block of sub-search spaces, the costs of using concurrency control decreased the performance of PHACT, and as so it is disabled by default.

Work stealing techniques were also considered for load balancing inside the devices. However, due to OpenCL limitations on concurrency control, this was an infeasible task. Namely, OpenCL (version 1.2) lacks a feature to block threads from accessing a sub-search space while another thread is changing some of the respective contents. In OpenCL, concurrency control can only be achieved through atomic operations over one C variable at a time.

In some CPU devices, concurrency control over a sub-search space could be achieved by implementing a semaphore with OpenCL atomic operations. However, after some testing it was noted that this algorithm was not working in GPUs, as it was always leading to an infinite loop. During the tests in GPUs, it was found that when an OpenCL thread blocked the access to a sub-search space data, that same thread was never executed again, which was leading to the infinite loop. To achieve a better occupancy, GPUs try to run at the same time the most threads that will execute the same code instruction in the next clock cycle. Applying this algorithm to the semaphore that was implemented, led to the GPU executing only the threads that were waiting to access the sub-search space data inside a loop, while the only thread that was capable of unlocking the semaphore would never be executed again.

4.5 Communication

To reduce the amount of data that is transferred to each device, all of them will receive the full CSP, that is, all the constraints, variables and their domains, at the beginning of the solving process. Afterwards, when a device must be instructed to solve a new block of sub-search spaces, instead of sending all the sub-search spaces to the device, only the information needed to create those sub-search spaces is sent.

If a device is to solve sub-search spaces $SS2$ and $SS3$ from Example 3, it will receive the information that the tree must be expanded down to depth 2, that the values of the first variable are repeated 2 times and the values of the second variable are repeated 1 time only (not repeated). With this information the device will know that the values of the first variable are repeated 2 times, so the third sub-search space ($SS3$)

will get the second value of that variable, and so on to the expansion depth. The values of the variables that were not expanded are simply copied from the original CSP that was passed to the devices at the beginning of the solving process.

This technique allows to greatly decrease the amount of data that is transferred from the host to the devices. For example, when solving the 17-queens problem a 32-bit bitmap is used for the domain of each of the 17 variables. If 200,000 sub-search spaces were to be created, that would lead to a search tree expansion depth of 5. As such, at least these five levels would be created for each sub-search space resulting in 1,000,000 bitmaps ($200,000 \times 5$) of 32-bits each (32,000,000 bits) that would have to be stored in the host memory and transferred to the device memory.

Each time a work-item needs a new sub-search space to explore, it increases by one the number of the next sub-search space that is yet to be explored on that device and creates the sub-search space corresponding to that number before being increased. Then it will do labeling, propagation and backtracking until either all the sub-search spaces of that block have been explored, when all the solutions must be found or optimizing, or only one solution is wanted and one of the work-items on that device finds a solution.

4.6 Implementation details

OpenCL allows to compile the kernels at runtime at the cost of some delay, which in the tests that were made, ranged from 0.2 s to 2 s, depending on the device for which it was compiled. However, this allows to compile a kernel more adapted to the CSP that will be solved. For example, it allows to compile only the propagators whose constraints are used in the CSP, which may lead to a great boost in the performance of PHACT, as it was noted in the N-queens problem. When using the Nvidia Geforce GTX 980M to solve the 12-queens problems while compiling only the propagators that are needed, PHACT took 4 s, but when compiling all the propagators it took 11 s, which shows the potential of the runtime compilation of OpenCL.

When this thesis was written, PHACT had 35 constraints implemented and supported reification. In PHACT, reification is implemented by adding a new boolean variable associated with the reified constraint, and by implementing a reification tester and an opposite propagator. The reification tester verifies if the constraint is already respected with all the values of the respective variables, which will set the boolean variable to value true, or if, on the contrary, the constraint will never be respected with all the values of the respective variables, in which case the boolean variable will be set to false.

If the boolean variable of reification is set to true, the normal propagator of the constraint is executed. On the contrary, if the boolean variable is set to false, the opposite propagator will be executed. For example, for the "not equal" constraint, the normal propagator will guarantee that the respective two variables are never assigned with the same value. On the contrary, the opposite propagator will guarantee that the respective two variables will be assigned with the same value.

The runtime compilation of PHACT allows it to enable and disable features inside the kernels without the need of recompiling the entire code. PHACT uses this OpenCL feature to enable/disable *Revision*. Revision is a technique implemented in PHACT that is applied after backtracking, that is, when a variable assignment by labeling causes an inconsistency, if it has yet values to be assigned, then it will be propagated with all these remaining values. In some cases, this allows to anticipate the removal of values from the domains of variables, or even to discard that branch of the search tree due to its inconsistency. In these cases, some propagations are only done once, which if revision was not used, might be done once per remaining value in the domain of the variable selected for labeling. However, after some tests it was found that most of the times this technique was greatly increasing the number of propagations done to solve a CSP, and in the end that was worsening the performance of PHACT. For that reason, this technique is

disabled by default.

This OpenCL feature is also used to compile only the heuristic that will be used for selecting the variable for labeling and the heuristic to select the value to assign to that variable. Currently PHACT has implemented three heuristics for selecting the variable to label:

- First fail - Select the variable with less values in its domain;
- Input order - Select the first variable introduced by the CSP model;
- Occurrence - Select the variable restricted by more constraints.

For selecting the value to assign to the variable for labeling, three heuristics are implemented:

- Maximum value - Select the greatest value from the variable domain;
- Minimum value - Select the smallest value from the variable domain;
- Split values - Splits the domain of the variable about the mean between the minimum and the maximum value and selects the lower half.

PHACT represents the variable domains as 32-bit bitmaps, multiples of 64-bit bitmaps, or as (compact) intervals. When using intervals, PHACT is slower than when using bitmaps, but intervals are meant to be used instead of larger bitmaps on systems where the size of the RAM is an issue. The constraint propagation when using intervals is much less efficient, as it can not remove values from the middle of the domains because they are represented only by their minimum and maximum values.

The host will always use bitmaps, but when intervals are to be used, the bitmaps will be converted to intervals before being sent to the devices, and converted back to bitmaps when received from the devices. The kernels will be compiled, at runtime, to use bitmaps (of the needed size), or intervals. For that purpose, all the functions for manipulating domains are implemented twice in the kernel, once for bitmaps and another for intervals. New domain representations can be added by implementing those functions accordingly and possibly an interface for translating the domains between the host and the devices.

With intervals, the domain of each CSP variable is represent as a 32-bit variable containing two 16-bit fields (OpenCL cl_ushort2), the minimum value and the maximum value of the variable domain. By default, in the host, the variables domains are represented as a 1024-bit bitmap, but this value may be changed before compiling PHACT. Thanks to the OpenCL runtime compilation, each device will always use the minimum size possible of bitmaps to store the variables domain, and it can even use intervals instead of bitmaps, without the need of recompiling PHACT.

However, PHACT uses the same size of bitmaps for all the CSP variables, which for some CSPs may force the allocation of more memory than the amount needed. For example, if most of a CSP variables are boolean, and only a few others have 100 values on their domain, PHACT will allocate 128 bits per variable for storing the values of its domain. Nevertheless, this allows for simpler bitmap operations, as all of them will have the same size.

The main reason for storing all the domains of the CSP variables in bitmaps of the same size is that, as stated in Section 4.1.1, OpenCL does not allow dynamic memory allocation inside a kernel. This forces PHACT to allocate the maximum memory it may need for solving a CSP inside a device. As the kernels are compiled at runtime, that amount is always calculated according to the current CSP requirements.

However, as each work-item will be exploring one sub-search space at a time, each one will need enough memory to store its own backtracking history.

In case of GPUs, which can execute thousands of work-items simultaneously, this may lead to large memory requirements, and for CSPs with more variables and/or bigger variables domains, this can force PHACT to reduce the number of work-items to fit the memory available. To reduce the memory requirements of backtracking histories, PHACT considers that the backtracking history will never go deeper than the number of variables marked for labeling in the CSP model. However, some MiniZinc/FlatZinc [7] models do not define which variables will be subject to labeling, in which case PHACT will consider the ones marked for output, or if none, it will consider all of them.

4.7 Conclusion

The techniques described in this chapter allow PHACT to use all the devices compatible with OpenCL to solve a CSP. It splits the search space in multiple search spaces that are distributed among the devices in blocks to reduce the number of communications between the host and the devices. The size of each block is calculated according to the speed of the respective device when solving the previous blocks to try to achieve a good load balancing between the devices.

The size of the data transfers between the devices and the host is reduced by replacing the blocks of fully created sub-search spaces with a small data set containing the information needed for a device to generate those sub-search spaces. Inside the devices, each search engine (thread) will do labeling, constraint propagation and backtracking on a sub-search space at a time.

Due to the absence of dynamic memory allocation inside the kernels, each search engine will always have allocated the maximum memory it may need to explore a sub-search space, which in case of GPUs, can limit the number of search-engines it can run simultaneously. However, PHACT uses some OpenCL features to improve its performance, as for example the runtime compilation. It allows PHACT to change the kernel according to the CSP it will solve, for example, by compiling only the propagators of the constraints that the CSP uses, and can even change the representation of the domains of the CSP variables between bitmaps and intervals without the need of recompiling PHACT.

5

Experimental results

PHACT was evaluated on finding all the solutions for two CSPs, on optimizing six other CSPs and on finding one solution for two CSPs. Table 5.1 presents some information about these CSPs, namely:

- CSP - the name and instance of the CSP that will be used in this Chapter;
- Number of variables - the number of variables in the FlatZinc model, after converting it from the MiniZinc model;
- Maximum domain value - The maximum value of the CSP variables;
- Number of constraints - the number of constraints described in the FlatZinc model, after converting it from the MiniZinc model;
- Types of constraints - the number of different FlatZinc Predicates [7] used in the FlatZinc model, after converting it from the MiniZinc model;
- Objective - If the objective of the CSP is to count all the solutions (Count), to find only one solution (One) or to find the best solution (Optimization).

Table 5.1: Information about the CSPs used in experimental results

CSP	Number of variables	Maximum domain value	Number of constraints	Types of constraints	Objective
bacp_2	1,970	100	2,028	5	Optimization
bacp_6	1,970	100	2,028		
bacp_7	1,970	100	2,026		
cryptanalysis_128_3_4	4,661	7	8,373	12	Count
cryptanalysis_128_5_11	15,793	7	32,637		
cryptanalysis_128_5_14	15,793	7	32,637		
golomb_9	37	81	554	3	Optimization
golomb_11	56	121	1,300		
golomb_12	67	144	1,881		
java_routing_6_3	752	258	1,093	8	Optimization
java_routing_8_5	1,256	338	1,873		
langford_3_9	783	27	2,178	3	One
langford_3_18	3,024	254	8,730		
m_queens_8	1,642	8	1,651	8	Optimization
m_queens_12	3,774	12	3,783		
m_queens_13	4,346	13	4,355		
market_split_s4_07	30	1	4	1	One
market_split_s5_01	40	1	5		
market_split_s5_04	40	1	5		
open_stacks_10_10	573	10	504	7	Optimization
open_stacks_20_20_a	2,296	20	2,459		
open_stacks_20_20_b	2,209	20	2,369		
project_planning_13_7	2,499	169	3,063	13	Optimization
project_planning_13_8	2,500	169	3,065		
queens_15	15	15	315	1	Count
queens_17	17	17	408		

All the CSPs were retrieved from the MiniZinc Benchmarks [49] and converted from MiniZinc to FlatZinc using the "mzn2fzn" tool [49]. From these 10 CSPs, the bacp, the cryptanalysis, the java routing and the project planning problems are categorized by the MiniZinc authors [50] as real problems and the remaining ones as academic.

A brief description of each one of the 10 CSPs is now presented and a more complete definition of them is contained in Appendix C.

- Bacp - The Balanced Academic Curriculum Problem (BACP) consists in achieving an academic curriculum by assigning periods to courses while maintaining a balanced academic load and respecting a set of administrative and academic rules [30];
- Cryptanalysis - Chosen Key Differential Cryptanalysis consists in the first step of a cryptanalysis attack against block ciphers to test their level of confidentiality, integrity and signature by modeling the Advanced Encryption Standard (AES) rules [28];
- Golomb ruler - Consists in placing a set of marks on an imaginary ruler, such that no two marks are at the same distance as any other two marks, minimizing the total length of the ruler [8];

- Java routing - May be defined as a routing problem in which multiple map locations must be visited, while visiting some locations prior to others, respecting a predefined maximum time to visit each one, and minimizing the total time required to visit all the locations [24];
- Langford - The Langford numbers problem consists in placing k sets of numbers ranging from 1 to n such that the number m is placed m numbers after the previous one [11];
- M-queens - To solve this problem, the minimum number of queens must be placed in a chessboard such that all the possible squares are covered, each one by a single queen [25];
- Market Split - This problem may be described as the allocation of retailers to one of two supplier divisions of a company, such that each division controls a predefined amount of the distribution of each product [1];
- Open Stacks - It consists in scheduling the construction of multiple products required to complete a customer order, such that the usage of the limited space in the production area is optimized to fulfill the most customer's orders in the fastest time [13];
- Project planning - Planning the execution of a set of tasks with predecessors while minimizing the total execution time [68];
- N-queens - The problem consists in placing n queens in a $n \times n$ chessboard, such that no queen attacks another one [53].

These CSPs were generated from the MiniZinc files shown in Table C.1. However, as PHACT does not yet possess all the heuristics for variable and value selection supported by the FlatZinc specification [7], neither can it use different heuristics while solving one CSP, all the MiniZinc files were set to use only the "input_order" and "indomain_min" heuristics.

When looking for CSPs to use for experimental results, the following aspects were considered:

- There should be a MiniZinc/FlatZinc model of the CSP to allow the usage of many CSPs available on-line and their input to other solvers for comparisons with PHACT;
- Models should have variables whose domains are only composed by positive integers and without sets, as PHACT does not work with variables whose domains are composed by negative values, floats or sets;
- The models should only contain variables whose domains lie between 0 and 1023;
- The models should contain only constraints that are already implemented in PHACT and recognized by its FlatZinc interpreter.

From Table 5.1 we can see that the ten CSPs used for benchmarking present many differing characteristics among them, allowing for a more generalised analysis of the performance of PHACT. Namely, they possess from 15 up to 15,793 variables with domains from 2 up to 338 values. These problems contain also from 4 up to 32,637 constraints, which in one problem correspond to 13 different propagators. As stated before, the objective of the CSPs does also vary and can be counting all the solutions, optimizing or finding a single solution. From these CSPs only the three Cryptanalysis instances are unsatisfiable, that is, they have no solutions.

The benchmarks were executed on one, two and three devices and on five different machines running Linux to evaluate the speedups when more devices are added to help the CPU. The five machines have the following characteristics:

M1 Machine with 32 GB of RAM and:

- Intel Core i7-4870HQ (8 compute units);
- Nvidia GeForce GTX 980M (12 compute units and 4 GB of RAM).

M2 Machine with 64 GB of RAM and:

- Two Intel Xeon CPU E5-2640 v2 (referred to as Xeon 1, 32 compute units);
- Two MICs (Xeon Phi coprocessors) 7120P (240 compute units and 16 GB of RAM each).

M3 Machine with 64 GB of RAM and:

- Two Intel Xeon E5-2690 v2 (referred to as Xeon 2, 40 compute units);
- Nvidia Tesla K20c (13 compute units and 5 GB of RAM).

M4 Machine with 128 GB of RAM and:

- Four AMD Opteron 6376 (64 compute units);
- Two AMD Tahitis (32 compute units and 3 GB of RAM each). These two devices are combined in an AMD Radeon HD 7990, but are managed separately by OpenCL.

M5 Machine with 96 GB of RAM and:

- Two Intel Xeon Silver 4110 (referred to as Xeon 3, 32 compute units);
- Two Nvidia Titan V (40 compute units and 12 GB of RAM each).

All the tests were repeated five times, and the elapsed times presented in this chapter are the geometric mean of these five runs and are shown in seconds. As the execution of some tests took longer than expected, the ones that were running for more than twelve hours were stopped and no results are presented for those five executions. For a better identification of the correspondence between the legend elements and each data series (line) of the line charts presented in this section, the elements of the legend are ordered according to the data series height on the rightmost end of the chart.

The results of these tests with GPUs, MICs and CPUs are presented and discussed in Sections 5.1, 5.2 and 5.3, respectively. Section 5.4 shows the results achieved when using more than one device to solve the problems. PHACT performance was compared with those of Gecode 6.2.0 [66], Choco 4.0.4 [55] and OR-Tools 7.2.6977 [29], whose results are presented in Section 5.5. The source code of PHACT is available for download at [57].

The performance of the solver EPS developed by Régin *et al.* [63] was not compared with the one of PHACT, as by following the instructions and running the installation script provided with the source code of EPS [56] it was not possible to install the solver in any of the five machines used for benchmarking. OR-Tools was also not used in M2 and M4 machines as it does not support the older versions of the software libraries installed in them.

5.1 Results on GPUs

In this section the smaller instances of eight of the ten CSPs described above were used for testing the speedups that could be achieved with four different GPUs. Only those eight instances of the problems were used due to the long time that PHACT would take to solve other instances with a single GPU thread.

Note that, for the GPU tests presented in this section, the number of work-items per work-group is always set to 128, except when using a single thread. This number of work-items per work-groups is set by default in PHACT, as explained in Section 2.2. This means that, for example, using 32,768 threads (work-items) corresponds to using 256 work-groups with 128 work-items each.

The speedups obtained by PHACT when using the Geforce GPU when incrementing the number of threads, are presented in Figure 5.1. The speedups are calculated by comparing the time that PHACT took to solve the problems with each number of threads, against the time needed to solve the same problem with 1 thread. The respective times are presented in Table 5.2.

Table 5.2: Seconds needed for PHACT to solve the problems with different numbers of threads on the Geforce

CSP	Number of Threads									
	1	**128**	**256**	**512**	**1,024**	**2,048**	**4,096**	**8,192**	**16,384**	**32,768**
Bacp_2	1,936.37	18.36	18.24	12.45	9.69	11.93	14.61	13.13	12.16	10.29
Cryptanalysis_128_3_4	7.76	5.04	4.75	4.19	4.11	3.88	3.61	3.85	5.39	6.88
Golomb_9	96.45	54.53	31.85	25.64	18.76	16.12	10.80	6.63	4.70	4.69
Langford_3_9	0.84	1.05	0.83	0.78	0.76	0.90	0.75	1.12	1.43	1.98
M_queens_8	9.64	8.72	8.27	7.57	5.93	5.69	5.56	5.68	5.85	5.43
Market_split_s4_07	3,192.48	81.67	37.47	19.56	10.60	6.43	4.30	3.79	4.19	5.42
Open_stacks_10_10	5.18	1.53	1.16	1.05	0.94	1.13	0.92	0.90	1.40	1.47
Queens_15	15,405.14	417.05	212.59	105.82	55.46	30.18	18.07	11.15	9.62	9.32

From all the CSPs used for testing, the N-queens is the simplest one, when attending to the type of constraints used. This problem is modelled only with "not equal" constraints, which state that a variable x must be assigned with a value different from the value assigned to a variable y, which leads to a very simple propagator.

As mentioned in Section 2.2, GPUs are mainly prepared to execute the same code instruction over many cores at the same time. This means that the more divergent paths the source code has the less efficiently the GPU will behave. When using a GPU to solve a CSP whose source code will only contain a few divergent paths, the GPU will be capable of using most of its parallel capabilities. This was the case when solving the Queens_15 problem, which is also visible in Figures 5.2 and 5.4.

The Queens_15 problem allowed the three GPUs to achieve speedups over 600, when comparing to them solving the same problem with a single thread, which shows the extreme parallel capabilities of these devices. To solve the Queens_15 problem with one thread, the Geforce GPU took 15,415.14 s, that were reduced to 9.32 s when using 256 work-groups with 128 work-items each, totalling 32,768 threads, achieving a speedup above 1,600.

Besides being one of the simplest problems used for these benchmarks, the Queens_15 was also the one that took more time to solve (to count all the solutions) with a single thread, which leaves more room for a speedup when increasing the number of threads. That is also consistent with the speedups obtained when using 32,768 threads to solve the Market_split_s4_07 and the Bacp_2 problems, which were the next two problems that took longer to solve sequentially (with a single thread).

On the contrary, the Langford_3_9 is solved very fast sequentially, which leaves no room for improvement when using more threads. From the 0.84 s needed to solve the problem, only 313 ms were used in the solving process, and the remaining time was used to load the model and to initialize the device. In fact, apart from when using 256, 512, 1,024 and 4,096 threads which allowed a very small speedup, with the

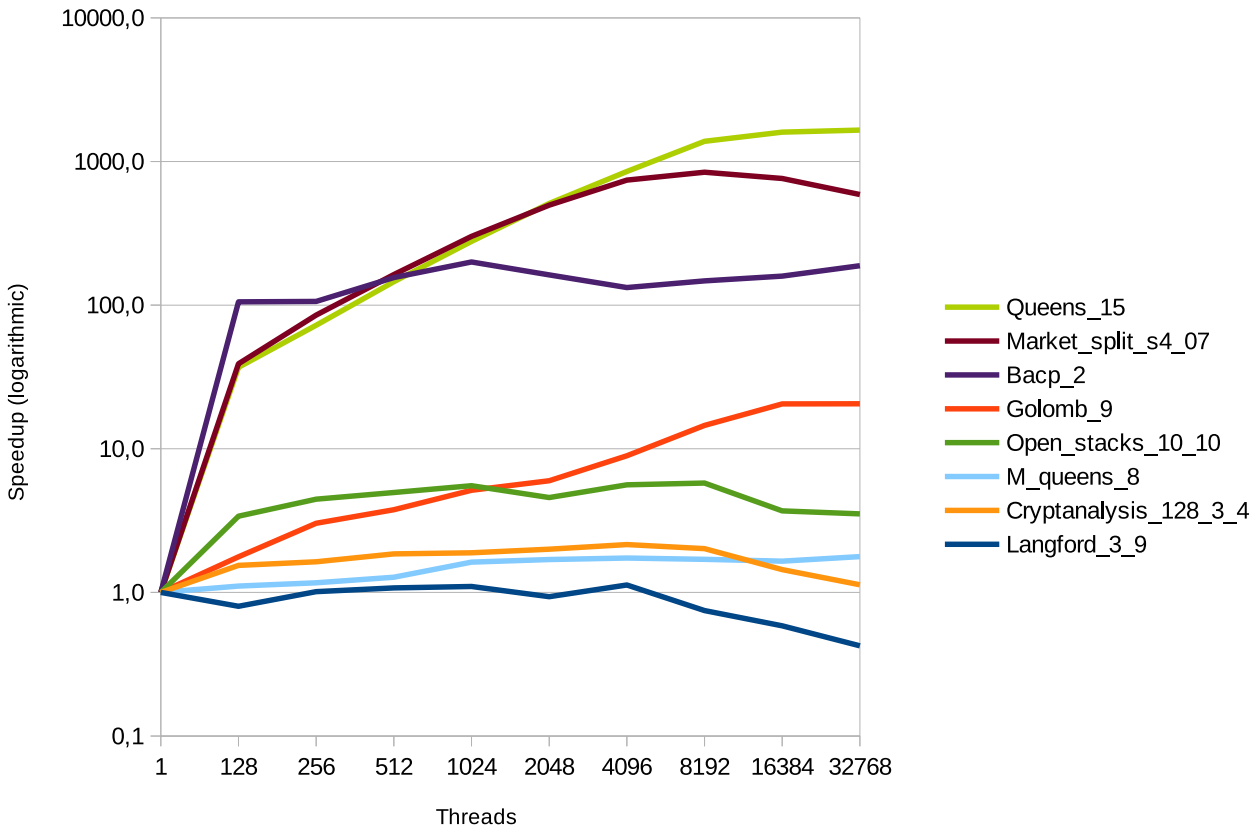

Figure 5.1: Speedups achieved with PHACT when increasing the number of threads on the Geforce GPU

remaining numbers of threads the time took to solve the problem increased. This may be due to the increased time needed to initialize all the threads and the respective resources, along with the increased number of sub-search spaces not compensating the actual work done in parallel by the threads.

Besides the Langford instance, the Cryptanalysis, the M_queens and the Open_stacks that were used in this section are also solved in a few seconds sequentially, which resulted in small speedups when increasing the number of threads to solve them. However, these instances were selected because all the other instances that were tested were either solved even faster or would take too long to solve sequentially.

The most accentuated increases and decreases in the speedups represented in Figures 5.1, 5.2, 5.3 and 5.4 are consistent with changes in the number of sub-search spaces that are created for that particular combination of work-groups and work-items, which may increase or decrease the number of propagations needed to solve the problem, as shown in Table 5.5. These accentuated increases and decreases are most visible for the Bacp_2 and the Langford_3_9 problems in the Geforce, the Tesla and the Tahiti GPUs. In the Titan GPU this result is much more attenuated for the Langford_3_9 problem, which may be due to the more recent architecture of this device making it more capable of balancing the distribution of the work among its hardware components.

One of the biggest obstacles when using backtracking techniques are the memory requirements to store all the previous states of the search space. This is the main bottleneck when running PHACT on a GPU, as they can run thousands of threads simultaneously and all of them need to store the previous states of their own search space.

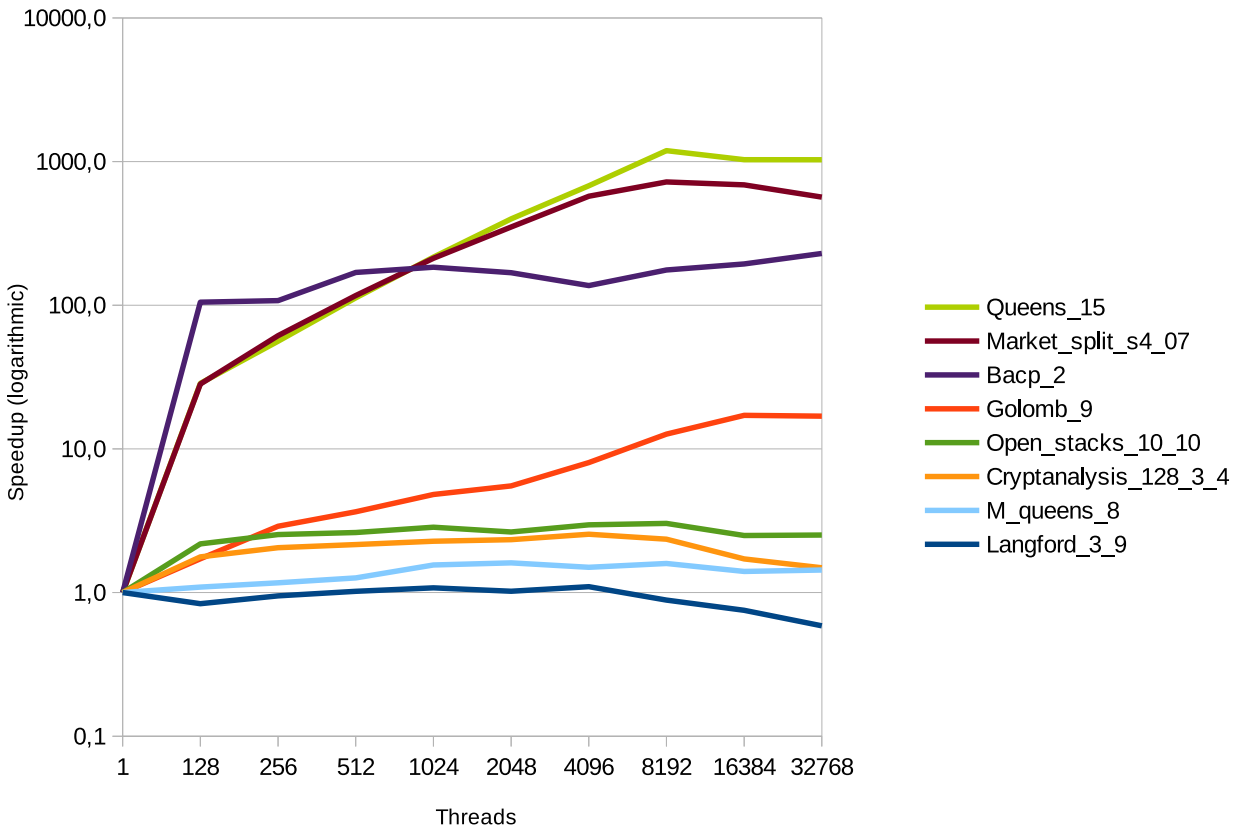

Figure 5.2: Speedups achieved with PHACT when increasing the number of threads on the Tesla GPU

Even with the smaller instances of the CSPs presented in this section, the memory requirements of PHACT to solve some of them did not allow the GPUs to use 32,768 threads. Instead, PHACT used as many threads as the RAM of the GPU allowed. When solving the problems in the Geforce, the Golomb_9, the Market_split_s4_07, the Open_stacks_10_10 and the Queens_15 problems allowed to use the 32,768 threads, but the Bacp_2, the Cryptanalysis_128_3_4, the Langford_3_9 and the M_queens_8 problems did not. They only allowed to use 756, 1,134, 32,640 and 10,368 threads, respectively.

Although for some problems the GPUs did not have enough memory to execute the 32,768 threads, the number of threads that they allowed are arranged in different numbers of work-groups when trying to use more threads. For example, the Bacp_2 problem did not allow to use more than 756 threads, however that number is only surpassed when trying to use 1,024 threads, which corresponds to 8 work-groups with 128 work-items each. After surpassing the number of threads that can be used on a device, PHACT reduces the number of work-groups and/or work-items to fit the memory. But that process can result in a different number of work-groups and consequently in a different number of work-items per work-group, according to the required amount of each.

These different combinations of work-items per work-group for the same number of threads result in a different usage of the GPU resources, like the shared memory which is shared among all the work-items of the same work-group. That resulted in the different performances of the GPU when solving, for example, the Bacp_2 with the maximum allowed number of threads, which can be seen in Figure 5.1, after surpassing the 756 threads.

Even when using a GPU from a different vendor, an AMD instead of a Nvidia, the results where very similar. Figure 5.3 shows the speedups of a Tahiti when solving seven of the eight problems. There are no results for the Queens_15 problem, as solving this problem with one thread on this GPU exceeded the 12 hour limit.

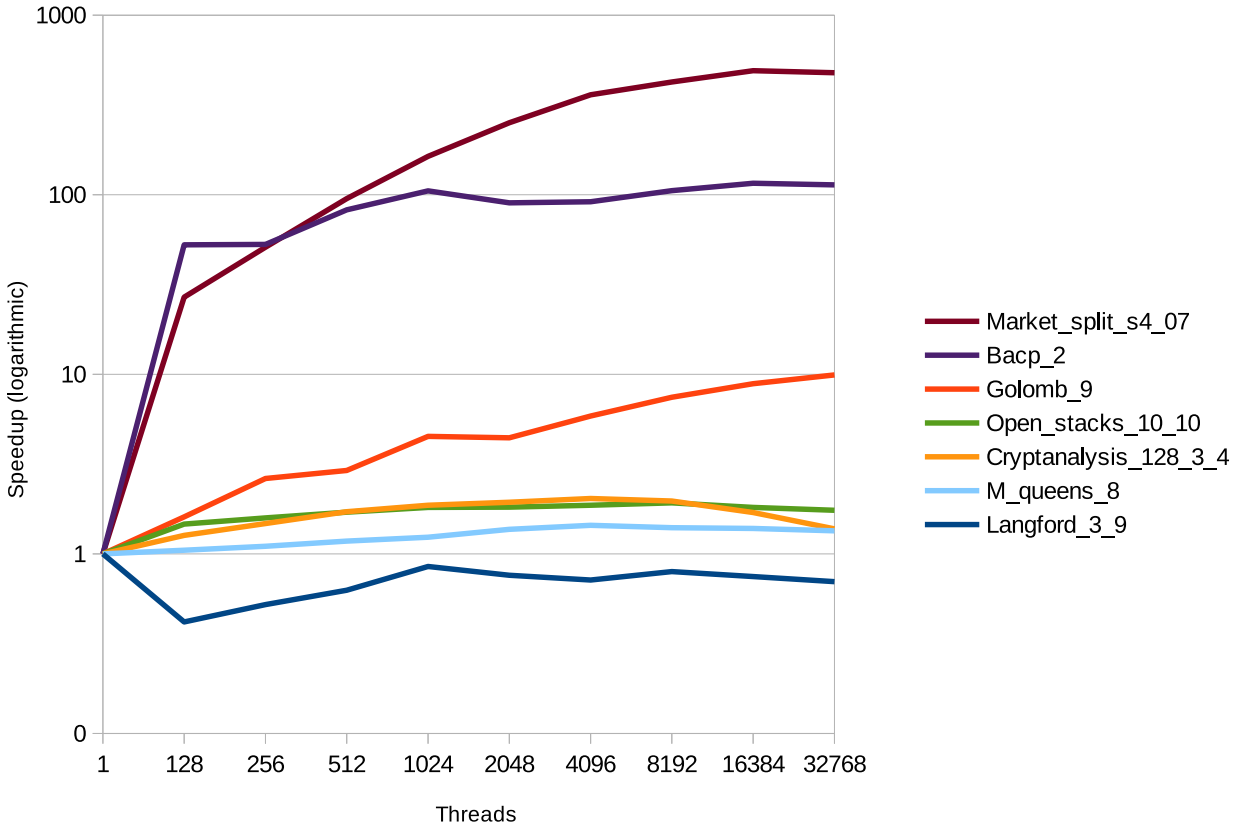

Figure 5.3: Speedups achieved with PHACT when increasing the number of threads on the Tahiti 1 GPU

5.1.1 Conclusion

In this section, four different GPUs were used to test the performance of PHACT while solving 8 CSPs. These GPUs allowed PHACT to achieve speedups greater than 1,600 when comparing to the same problem being solved with a single GPU thread, which shows the adaptability of PHACT to take advantage of massively parallel devices to speed up the solving process of constraint problems.

5.2 Results on MICs

In this section, the performance of PHACT was tested on a MIC while solving the same eight problems that were solved using the three GPUs in the previous section, and the smaller instances of the java_routing and of the project_planning problems.

The Intel MIC allows a greater level of parallelism than the CPUs, but smaller when compared to GPUs. However, its cores are faster than the ones of the GPUs and possess a bigger instruction set, which allows

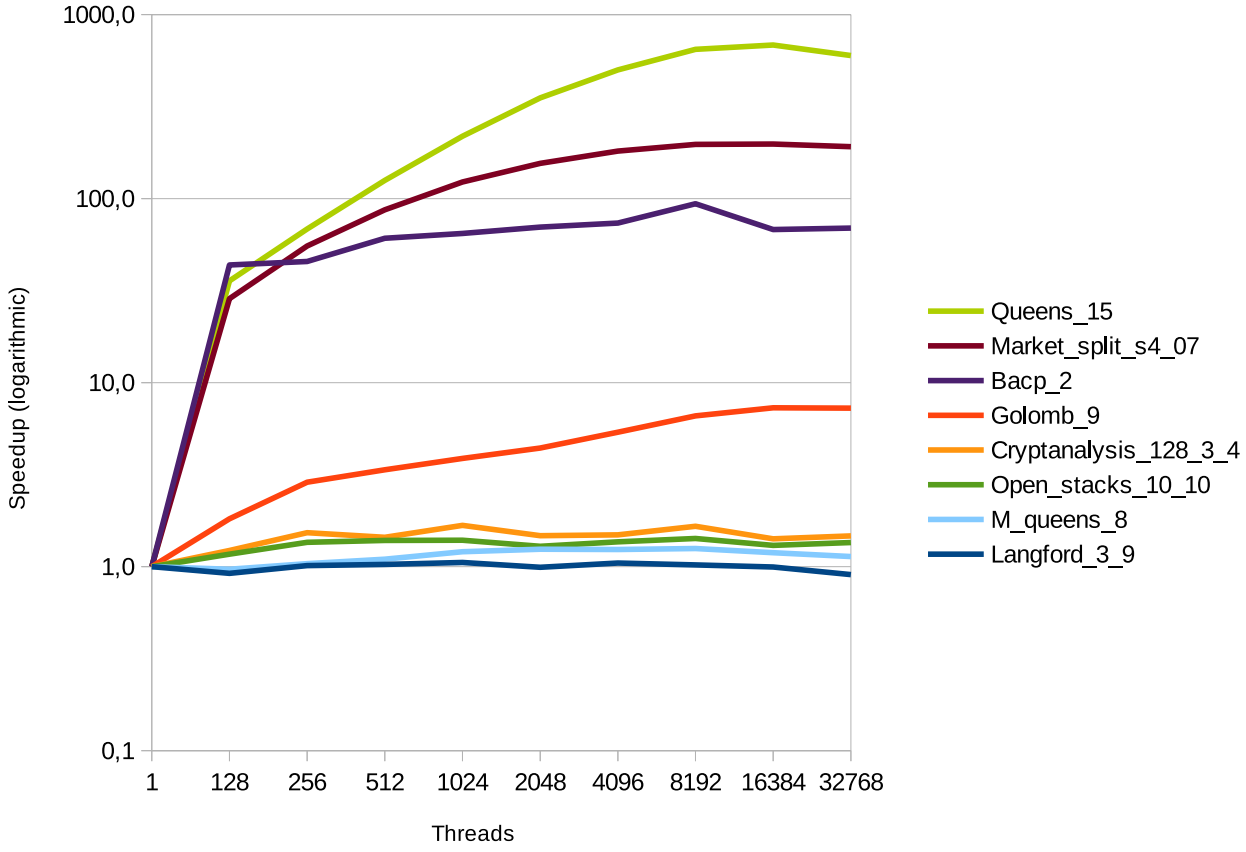

Figure 5.4: Speedups achieved with PHACT when increasing the number of threads on the Titan 1 GPU

them to solve each problem faster than the GPUs when using the same number of threads. This allowed to benchmark PHACT on the MIC with ten CSPs, instead of only the eight used with the GPUs in the previous section, without the problem of them taking too long to solve.

Figure 5.5 shows the speedups against the sequential run, achieved with PHACT when solving the ten problems while increasing the number of threads in a MIC. The elapsed times used for constructing the chart are shown in Table 5.3.

The best speedups were also achieved when solving the Queens_15 problem, and the worst speedups were achieved with the same problems as when using GPUs, namely, the M_queens_8, the Langford_3_9, the Open_stacks_10_10 and the Cryptanalysis_128_3_4 problems, which shows that these four problems are the hardest to solve of this set of ten.

5.2.1 Conclusion

The expected level of parallelism when using MICs is situated between the one of most CPUs and GPUs, as they possess more cores than most of the CPUs, but much less cores than most of the GPUs. As such, the results that were achieved are in accord with what was expected.

Table 5.3: Seconds needed for PHACT to solve the problems with different numbers of threads on a MIC

CSP	Number of Threads								
	1	2	4	8	16	32	64	128	240
Bacp_2	131,12	65,93	38,11	21,23	14,56	10,48	9,42	9,17	9,55
Cryptanalysis_128_3_4	7,66	7,75	7,82	7,51	7,72	7,75	7,76	7,83	8,97
Golomb_9	13,66	13,23	9,60	9,49	8,13	7,70	7,65	7,33	7,50
Java_routing_6_3	547,87	275,49	141,65	77,00	43,04	26,38	18,07	15,32	16,19
Langford_3_9	4,47	4,40	4,18	4,26	4,39	4,27	4,27	4,35	4,33
M_queens_8	5,87	5,91	5,65	5,72	5,69	5,58	5,71	5,73	5,77
Market_split_s4_07	218,16	134,01	74,67	47,64	26,15	16,90	11,35	8,90	7,77
Open_stacks_10_10	5,48	5,65	5,58	5,40	5,47	5,50	5,29	5,61	5,56
Project_planning_13_7	6286,65	3067,63	1528,27	789,66	457,62	380,50	347,33	204,10	207,30
Queens_15	890,89	442,62	224,87	121,44	66,25	36,08	22,06	15,19	13,69

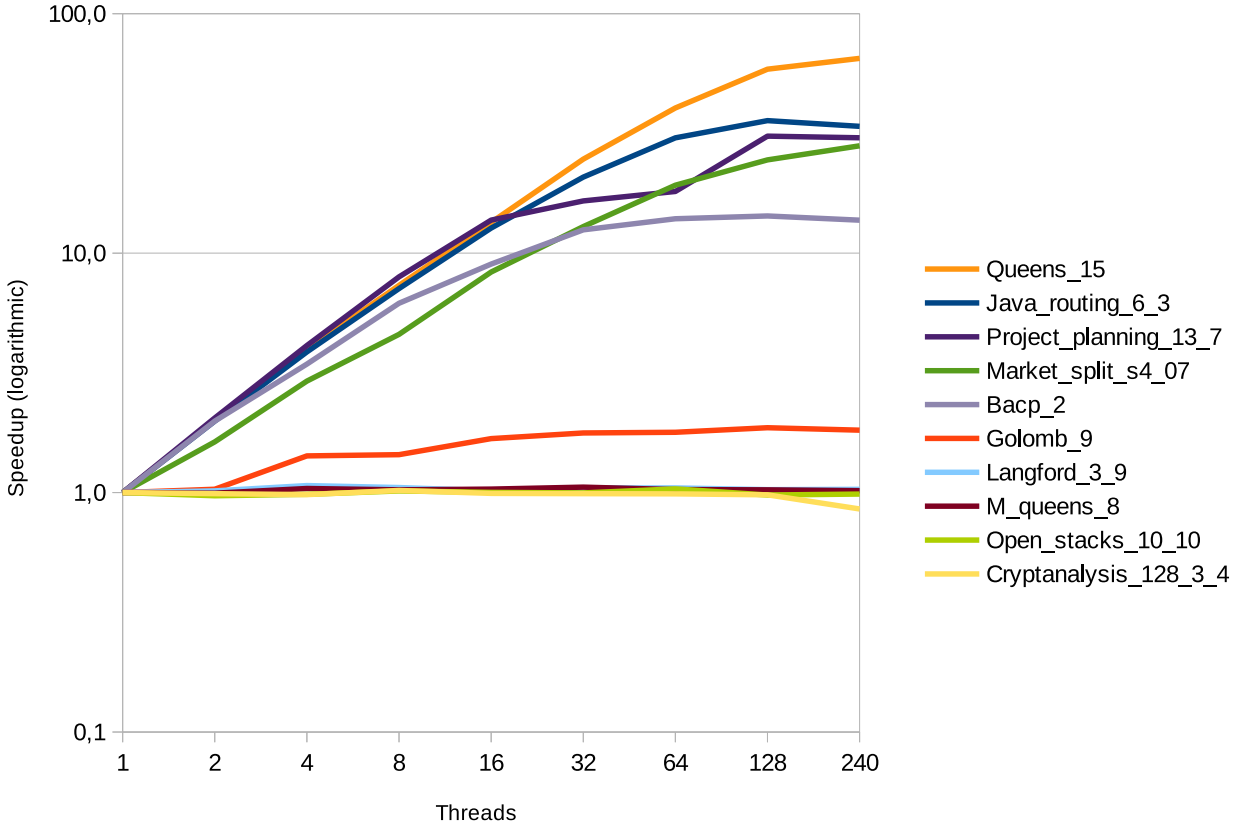

Figure 5.5: Speedups achieved with PHACT when using an increasing number of threads on a MIC

5.3 Results on CPUs

The capabilities of PHACT to use the parallel processing power of multi-threading CPUs were tested with 10 different CSPs on the five CPUs presented in the beginning of this chapter.

Figure 5.6 presents the speedups achieved when solving the CSPs on the I7 CPU, using 1, 2, 4 and 8 threads,

corresponding to the elapsed times from Table 5.4. The speedups were calculated when comparing the run times against the one obtained when using only one thread.

Table 5.4: Seconds that PHACT took to solve each CSP when using from 1 to 8 threads on I7

CSP	Number of threads			
	1	2	4	8
Bacp_7	133.13	66.32	47.38	35.87
Cryptanalysis_128_5_11	95.73	54.36	38.58	30.98
Golomb_11	115.69	63.76	40.45	36.59
Java_routing_6_3	43.42	23.34	14.51	11.51
Langford_3_18	79.24	64.98	38.02	56.15
M_queens_12	102.48	58.55	63.77	61.81
Market_split_s5_04	65.28	39.04	25.77	18.09
Open_stacks_20_20_a	92.90	47.43	29.21	23.77
Project_planning_13_7	286.72	141.85	86.03	62.64
Queens_15	68.87	43.60	23.35	15.66

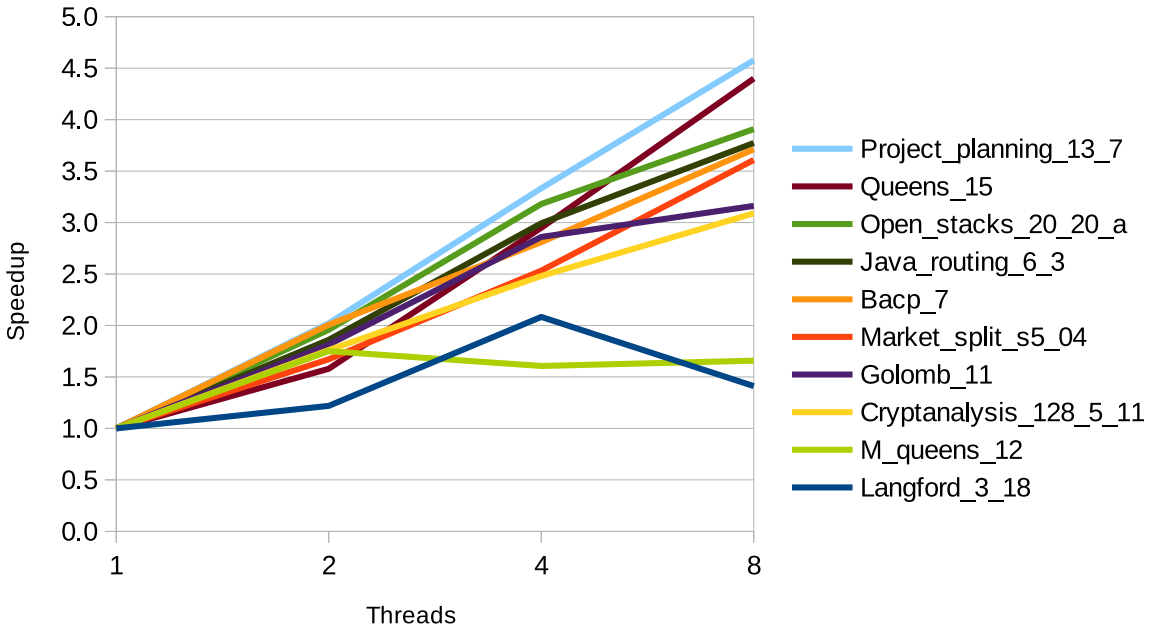

Figure 5.6: Speedups achieved with PHACT when using from 1 to 8 threads on I7

When using only one thread to solve a CSP, PHACT does not split the CSP in sub-search spaces, but when using more than one thread, by default, it creates at least 5,000 sub-search spaces per CPU thread (Section 4.2). For some CSPs, some of the sub-search spaces that are created are already inconsistent. As each sub-search space is independently checked for inconsistency, that verification may imply a lot of repeated constraint propagation between threads when the inconsistency is located higher in the search tree, or each variable is very constrained.

Table 5.5 presents the number of sub-search spaces that were created for each CSP, when using the I7

CPU with 8 threads. It also compares the number of propagations needed to solve each CSP with these sub-search spaces and when using a single thread and consequently one search space. The number of propagations is the number of times any constraint propagator was executed after a change of the domain of any variable constrained by some constraint.

Table 5.5: Number of propagations when using the I7 CPU

CSP	Propagations to explore the CSP (1 search space) with 1 CPU thread	8 CPU threads		
		Number of sub-search spaces created	Propagations to explore all sub-search spaces	Percentage of propagations against 1 CPU thread
Bacp_7.fzn	597,009,177	65,536	642,077,128	107.55%
Cryptanalysis_128_5_11.fzn	1,186,394,856	46,875	1,245,526,916	104.98%
Golomb_11.fzn	4,181,239,768	49,284	4,288,844,713	102.57%
java_routing_6_3.fzn	641,242,225	41,472	661,756,797	103.20%
Langford_3_18.fzn	1,811,275,857	40,000	5,978,654,537	330.08%
M_queens_12.fzn	1,348,496,724	65,536	1,348,043,046	99.97%
Market_split_s5_04.fzn	268,813,632	65,536	268,280,306	99.80%
Open_stacks_20_20_a.fzn	2,155,561,602	40,000	2,182,324,092	101.24%
Project_planning_13_7.fzn	2,733,271,361	40,960	2,702,642,780	98.88%
Queens_15.fzn	5,629,019,972	40,500	5,630,564,494	100.03%

Note that the values shown in the column "percentage of propagations against 1 CPU thread" were retrieved from a single run and may be different when using another number of threads and/or sub-search spaces.

Apart from when counting all the solutions, which was the case for the Cryptanalysis_128_5_11 and for the Queens_15 problems, these values may even change when repeating the same executions with 8 threads, as the ones used for gathering data for this table. They may change because when using multiple threads for searching for one solution for a CSP or optimizing it, different results may be achieved as different resources and execution times may be applied by the Operating System to each one of the threads. That may lead to some sub-search spaces being explored prior to others, which may even result in different solutions to be found for a problem, when looking for one solution or optimizing it.

As shown in Table 5.5, the Langford_3_18 was the CSP for which the number of propagations increased more when creating more sub-search spaces. The objective for all the instances of the Langford Numbers problem that were used for benchmarking in this thesis was to find a single solution.

After analysing the sub-search spaces that were created when using 4 and 8 threads to find one solution for the Langford_3_18, it was found that the solution was the same. It was located in the sub-search space number 1,632 when using 8 threads (and 40,000 sub-search spaces) and in the sub-search space number 816 when using 4 threads (and 20,000 sub-search spaces). However, some of the sub-search spaces prior to the ones where the solution was found required a lot of propagations to fully explore. For example, the thread that picked the first sub-search space did not finish to explore all of it before another thread found the solution, when using both 4 threads or 8.

It was also noted that, when using 4 threads, PHACT explored 1,225 sub-search spaces before finding the solution, but when using 8 threads it explored 5,714, which explains the large increase in the number of propagations from when using 4 threads. When using 8 threads, the one that found the solution only explored 245 sub-search spaces, but when using 4 it explored 329, which can only be explained by the

unbalanced sub-search spaces into which this CSP was divided. This means that the thread that found the solution when using 8 threads explored harder sub-search spaces than the one when using only 4 threads, before finding the solution. This resulted in the slowdown visible in Figure 5.6 when passing from 4 to 8 threads to solve the Langford_3_18 problem.

When solving the M_queens_12, PHACT only increased its speedup when using up to 2 threads. With more than 2 threads the time needed to solve the problem did not change much. The reason for this behaviour was that, for example, when using 8 threads, 7 of them finished their work before 22 s have passed, and the remaining thread kept working on the same sub-search space for about more 39 s. Once again, like with the Langford_3_18 problem, the cause were the unbalanced sub-search spaces that were created.

For the remaining CSPs, the speedup has always increased when increasing the number of threads to solve the problem, which was the expected result. The best speedup obtained was of about 4.6 when using 8 threads to solve the Project_planning_13_7 problem. Due to the memory requirements of PHACT to solve this problem, it was not solved on GPUs because it would be solved with only a few threads which would take too long to finish. However, the Queens_15, which was also solved on GPUs as presented in Section 5.1, where it allowed the GPUs to achieve the best speedup, was also the problem for which the best speedup was achieved on the I7 CPU, from the problems that were solved on both GPUs and CPUs.

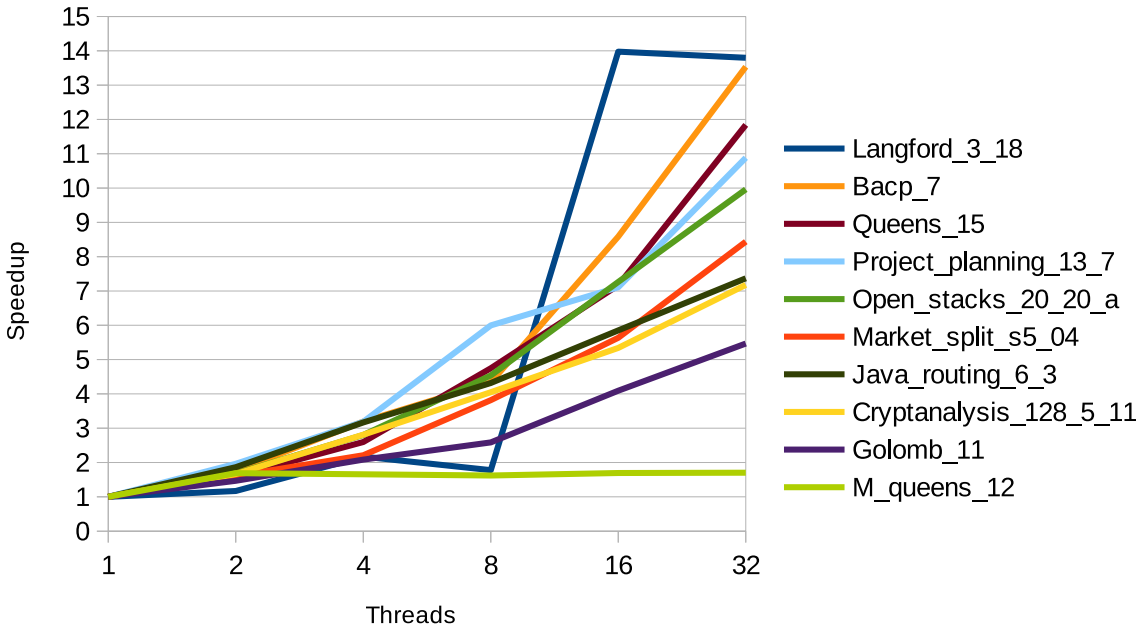

Figure 5.7: Speedups achieved with PHACT when using from 1 to 32 threads on Xeon 1

When solving the same CSPs on Xeon 1, with up to 8 threads, the results were not very different from when solved on I7. However, after the eight threads some changes occurred, as shown in Figure 5.7. The greater change was in the speedup achieved when solving the Langford_3_18 problem which went from about 1.2 when using 8 threads to 14 when using 16 threads. Once again, this was due to the very unbalanced sub-search spaces that were created, which when using 16 threads were explored by a different thread than the one that found the solution.

The behaviour of PHACT when solving the remaining problems on Xeon 1 was similar to when the I7 was

used, with all the problems achieving better speedups when increasing the number of threads, except for the Langford_3_18 and the M_Queens_12, for the same reasons as described before. But with this CPU the best speedup went up to about 14.

Figure 5.8 presents the speedups achieved when solving the same CSPs on Xeon 3, which is also capable of running 32 threads simultaneously. In this CPU, the maximum speedup obtained was also of about 14 when using 16 threads to solve the Langford_3_18 problem. However from 16 to 32 threads, the decrease in performance to solve this problem was more accentuated on Xeon 3 than on Xeon 2. This may be due to the differences in the architecture of both CPUs, from which Xeon 3 is more recent than Xeon 1 and possess an higher clock frequency.

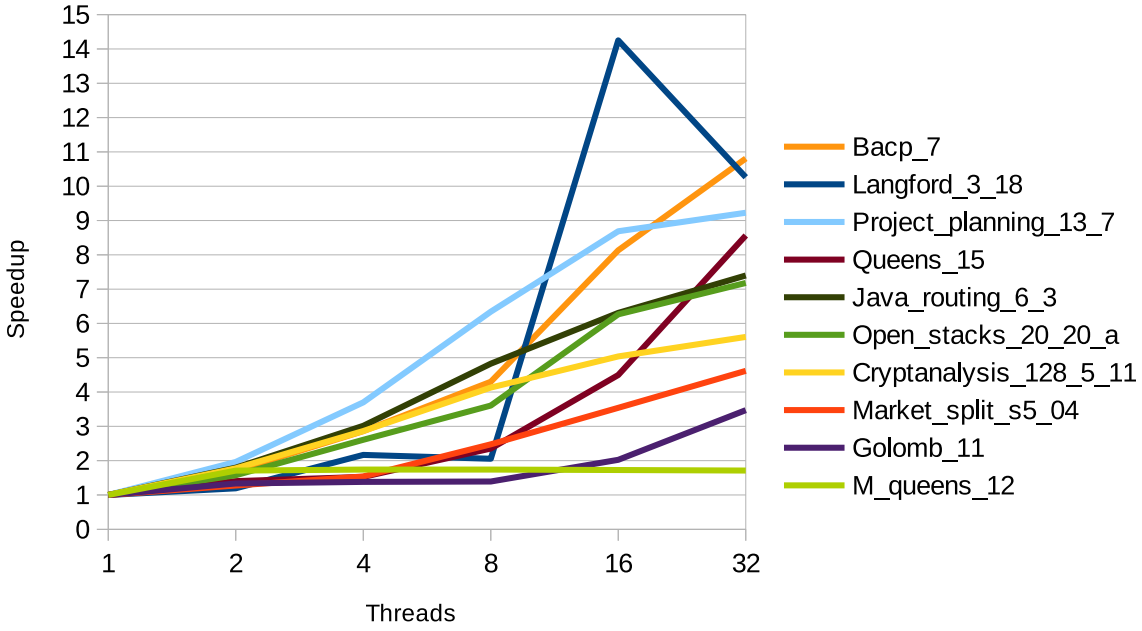

Figure 5.8: Speedups achieved with PHACT when using from 1 to 32 threads on Xeon 3

The faster architecture of Xeon 3 allowed it to solve all the problems faster when using one thread. For example, Xeon 1 took 457 s to solve the Project_planning_13_7 problem and Xeon 3 took 210 s, which leaves less space for speeding up the solving process when using more threads. This may explain why most of the speedups achieved by Xeon 3 where lesser than the ones obtained by Xeon 1.

Figure 5.9 shows the speedups achieved when using Xeon 2 to solve the same CSPs, when using from 1 to 40 threads.

In this chart it is observable that as the number of threads increases so does the spread of speedups between the problems and that the speedup tended to increase less as the number of threads was augmented. This may be due to the fact that by adding more threads to help solving the problems, it also adds more sub-search spaces to explore and more synchronization between the threads. This increase in synchronization and in repeated work due to the added number of sub-search spaces may end up by deteriorating the speedups.

However, once again, the Langford_3_18 problem was the exception, as it achieved a speedup boost when increasing the number of threads from 32 to 40, which made this problem the one for which the best speedup was achieved on the Xeon 2, namely of about 15.

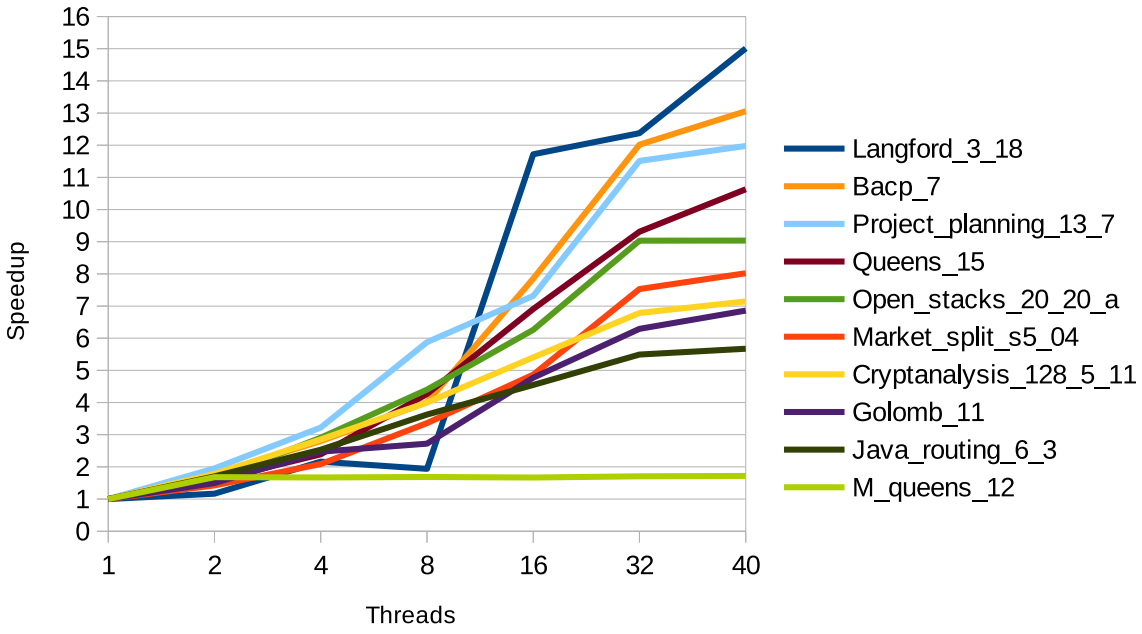

Figure 5.9: Speedups achieved with PHACT when using from 1 to 40 threads on Xeon 2

The speedups when solving the same CSPs, but using a CPU capable of running 64 threads simultaneously, are represented in Figure 5.10, and the respective elapsed times are shown in Table 5.6.

Table 5.6: Time in seconds that PHACT took to solve each CSP when using from 1 to 64 threads on Opteron

CSP	Number of threads						
	1	2	4	8	16	32	64
Bacp_7	472.10	356.18	203.37	125.98	54.62	26.82	14.65
Cryptanalysis_128_5_11	237.84	177.41	101.84	63.54	41.78	27.78	20.91
Golomb_11	207.34	132.23	81.84	50.60	27.60	16.50	10.21
Java_routing_6_3	89.72	64.29	36.65	22.93	14.03	9.51	7.67
Langford_3_18	371.31	495.57	260.38	295.54	25.81	26.27	14.38
M_queens_12	311.62	239.08	203.73	192.88	187.66	185.87	167.22
Market_split_s5_04	142.54	97.09	58.00	35.65	24.14	22.70	22.69
Open_stacks_20_20_a	334.99	240.71	131.46	76.10	40.70	22.85	14.15
Project_planning_13_7	1,284.39	1,016.73	535.70	301.46	177.80	97.22	58.71
Queens_15	124.04	78.99	44.04	26.58	15.04	8.83	5.75

The clock speed of the Opteron CPU is the smallest of the five CPUs used for experimentation, which resulted in it being the slowest CPU to solve each problem with 1 thread. However, when using all the 64 threads it was the fastest CPU to solve most of the problems. The exception was the Market_split_s5_04 problem, for which the Xeon 1 and Xeon 2 achieved greater speedups.

The time that PHACT took to solve the Market_split_s5_04 in the I7 and in the Opteron was almost the

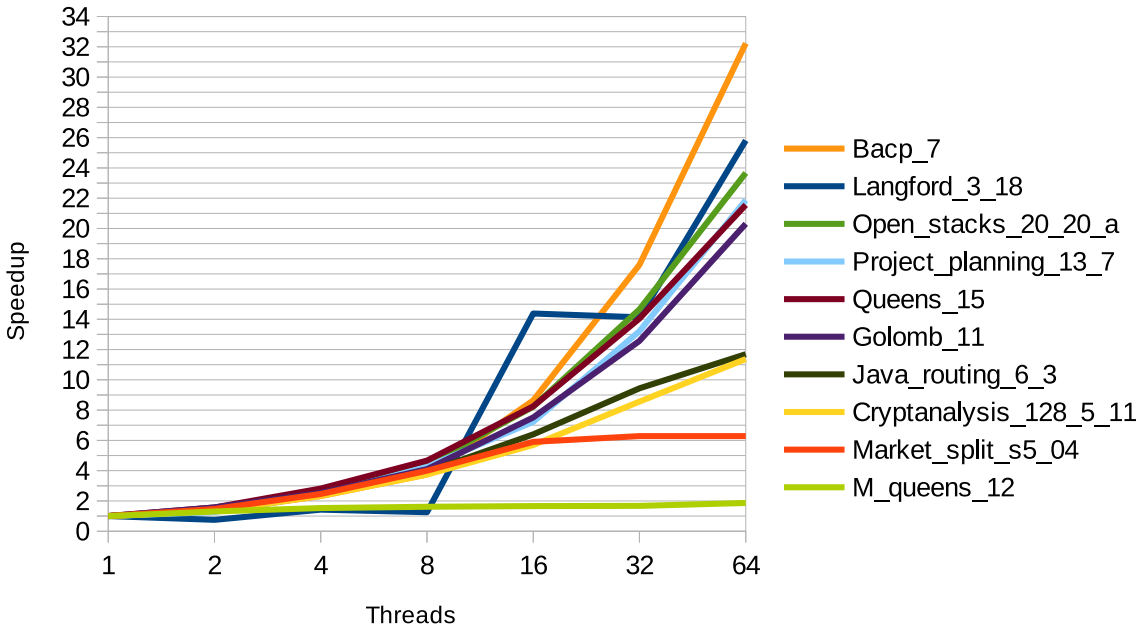

Figure 5.10: Speedups achieved with PHACT when using from 1 to 64 threads on Opteron

same, 18.9 s against 22.69 s, respectively. That may be explained by the fact that for this problem only one solution was required and this problem has only one solution. Which means that the time needed to solve the problem will depend on the time required to explore the sub-search space that contains the solution together with the time that the same thread that found the solution has spent exploring other sub-search spaces. For this problem, that time was similar in the I7 and in the Opteron CPUs.

When using the Opteron CPU, the differences between speedups achieved for each CSP are more evident, ranging from about 2 up to 32. In this CPU, the best speedup was obtained when solving the Bacp_7 problem, which was in the top 2 on four of the five CPUs used.

5.3.1 Conclusion

In this section, ten CSPs were solved using three CPUs with different parallel capabilities. It was noted that PHACT was capable of achieving a good speedup on all the CPUs and for most of the problems. Also, as expected, the speedups increased as the number of threads also increased for all the problems, except for the M_Queens_12 and the Langford_3_18 problems due to unbalanced sub-search spaces.

5.4 Results on multiple devices

PHACT allows using more than one device at the same time to solve a CSP. In this section, various combinations of devices were used to solve some of the CSPs also used in the previous sections.

Table 5.7 shows the elapsed times obtained with PHACT when using 1 thread on the I7 CPU, the Geforce GPU, and both simultaneously to solve 10 problems. Figure 5.11 presents the speedups achieved with PHACT when comparing the elapsed times while using the Geforce GPU against 1 thread on the I7 CPU,

or both against 1 thread on the I7 CPU.

Table 5.7: Seconds that PHACT took to solve each CSP when using 1 thread on the I7 CPU, the Geforce GPU or both, on M1

CSP	1 thread on I7	Geforce	1 thread on I7 and Geforce
Bacp_7	133.13	83.68	197.18
Cryptanalysis_128_5_11	95.73	280.37	90.45
Golomb_11	115.69	424.71	136.62
Java_routing_6_3	43.42	65.74	45.36
Langford_3_18	79.24	184.48	145.78
M_queens_12	102.48	9,665.12	2,858.59
Market_split_s5_04	65.28	18.92	22.77
Open_stacks_20_20_a	92.90	201.86	93.21
Project_planning_13_7	286.72	1,565.27	357.28
Queens_15	68.87	9.39	13.65

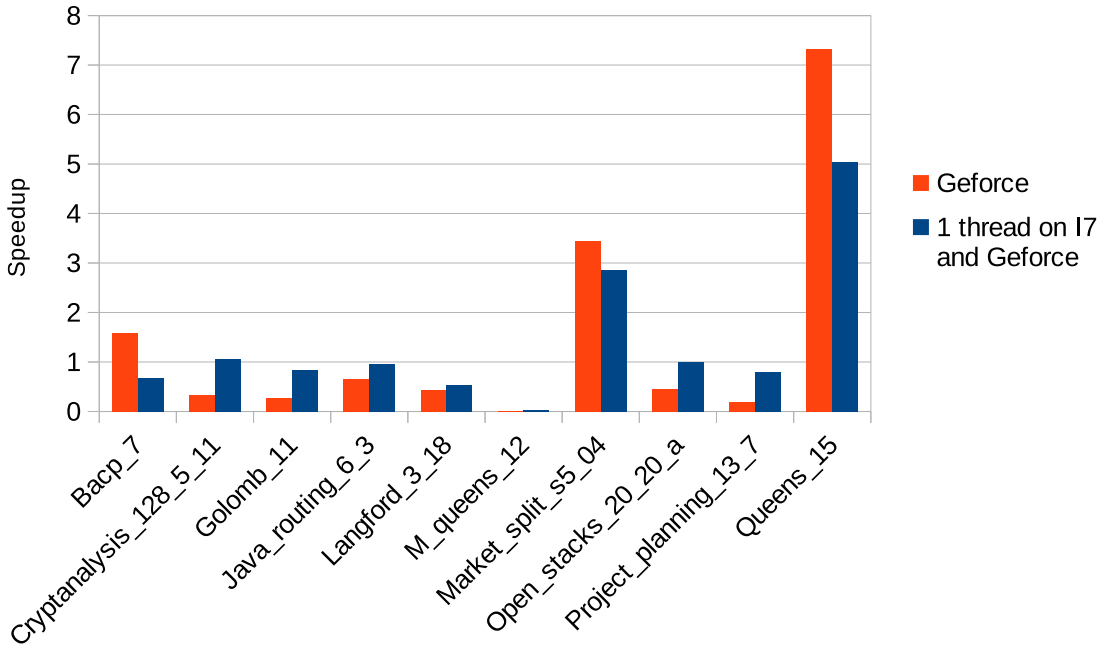

Figure 5.11: Speedups achieved with PHACT when comparing the elapsed times achieved with the Geforce GPU against 1 thread of the I7 CPU, and both against 1 thread of the I7 CPU, on M1

Using the Geforce GPU alone allowed to achieve speedups for only 3 of the 10 CSPs, achieving the best speedup (7.3) for the Queens_15 problem, and the worst speedup for the M_queens_12 problem (0.01).

GPUs are more inefficient when dealing with synchronization between threads, as for example, when using atomic operations to change the value of the cost to optimize when solving optimization problems. This can help to explain the motive of the bad performance of the Geforce GPU when solving these type of problems.

However, the greater bottleneck when using PHACT on this GPU to solve these problems was the size of the Geforce RAM. From these 10 CSPs, only the Golomb_11, the Market_split_s5_04 and the Queens_15 were executed with the default number of threads (65,536). The least number of threads used was 128 for the Cryptanalysis_128_5_11.

The number of divergent paths of the code used for solving the CSPs is also much related with the performance of PHACT when using GPUs to solve these CSPs. Although GPUs possess a much greater number of cores than CPUs, the CPU hardware is much better prepared for dealing with divergent paths in the source code than GPUs.

From the 10 CSPs, the Queens_15 is the simpler problem, with 15 variables, each one with only 15 values on its domain and only a small number of "not equal" constraints, as presented in Table 5.1. That simplicity makes the Queens_15 the problem with less divergent paths in the source code. The simplicity, the non existent atomic operations, as the objective of this problem was to count all the solutions, and the usage of 65,536 threads allowed the GPU to achieve a speedup of 7.3 when compared with a single thread on the I7 CPU.

On the contrary, the M_queens_12 is much more complex and only allowed to use 1,638 threads. Together with the unbalanced sub-search spaces which when explored in a GPU are much more problematic due to the slower cores which are also much less prepared for divergent paths, made the M_queens_12 take much longer to be solved on the Geforce GPU than on a single thread on the I7 CPU.

If the CSPs are solved by 1 thread on the I7 CPU and the Geforce GPU at the same time, new problems arise. To use more than one device simultaneously, PHACT needs to manage the synchronization between all the devices on the host side, as described in Chapter 4. For that, each device will receive many blocks of sub-search spaces to explore, one at a time, and only then it synchronizes with the host. When optimizing a problem, only then the best current cost is shared among devices, which may lead to devices searching for solutions that are already worse than others already found by other devices. When searching for one solution, the problem is similar, as a device may have already found a solution and finished, but the other device will only terminate when it depletes its current block of sub-search spaces.

These are the reasons why only for the Cryptanalysis_128_5_11, the Market_split_s5_04 and the Queens_15 the Geforce along with 1 thread on the I7 achieved a better performance than 1 thread on the I7 alone. However, for most of the CSPs, the speedup when using both devices was better than when using only the Geforce GPU. The exceptions were the Bacp_7, the Market_split_s5_04 and the Queens_15, for which the Geforce alone was faster than 1 thread of the I7 CPU.

When comparing the speedups of the Tesla GPU against a single thread of the Xeon 2 CPU and of both devices against a single thread of the Xeon 2 CPU in Figure 5.12, the results were similar with the ones of Figure 5.11, except that the speedups were smaller as the Tesla GPU is slower than the Geforce GPU. That difference in speed explain why in this case, 1 thread on Xeon 2 along with the Tesla achieved a greater speedup than the Tesla alone to solve the Queens_15 problem.

Using the Tesla GPU to help 1 thread of the Xeon 2 allowed to obtain a speedup only when solving the Market_split_s5_04 and the Queens_15 problems, when comparing with solving the problems with 1 thread on Xeon 2 alone.

Table 5.8 presents the times needed by PHACT to solve six CSPs when using 1 thread on the Opteron CPU, 1 Tahiti or 2 Tahitis, and combinations of the 3 devices of the M4 machine. The respective speedups are presented in Figure 5.13. For most of the problems, the Tahiti GPUs were slower than the other three GPUs used for testing PHACT. This resulted in not obtaining results with the Tahiti to four of the ten CSPs solved with the other GPUs, due to them taking more than twelve hours to solve.

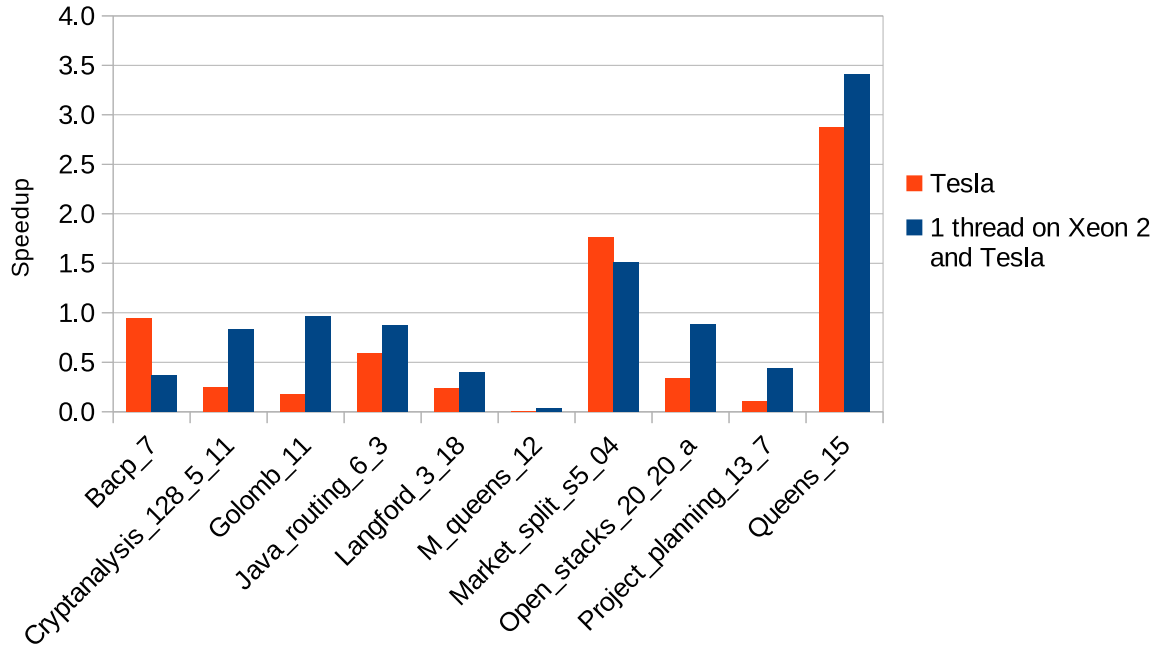

Figure 5.12: Speedups achieved with PHACT when comparing the elapsed times achieved with the Tesla GPU against 1 thread of the Xeon 2 CPU and both against 1 thread of the Xeon 2 CPU, on M3

Table 5.8: Seconds that PHACT took to solve each CSP when using combinations of the devices on M4

CSP	1 Opteron thread	1 Tahiti	2 Tahitis	1 Opteron thread and 1 Tahiti	1 Opteron thread and 2 Tahitis
Cryptanalysis_128_5_11	237.84	694.09	411.29	290.69	282.35
Golomb_11	207.34	829.38	229.69	171.81	157.93
Java_routing_6_3	89.72	417.98	533.29	251.72	493.78
Market_split_s5_04	142.54	2,052.51	34.95	45.90	33.81
Open_stacks_20_20_a	334.99	431.40	484.25	306.49	299.58
Queens_15	124.04	21.85	17.44	22.98	17.14

When compared with 1 thread on Opteron, 1 Tahiti GPU allowed to obtain a speedup only when solving the Queens_15 problem, which as stated before, is the problem for which the GPUs achieved the greater speedups. For the remaining problems, most of the results were different among them, and speedups were only achieved when using 1 or 2 Tahitis along with 1 thread on the Opteron CPU. The exception was when solving the Market_split_s5_04, for which a speedup was also obtained when using 2 Tahitis. This means that the usage of more devices allowed to start exploring a sub-search space containing a solution, faster than a single device alone.

Table 5.9 presents the times needed by PHACT to solve ten CSPs when using 1 thread on the Xeon 3, 1 Titan or 2 Titans, and combinations of the 3 devices of the M5 machine. The respective speedups are presented in Figure 5.14.

The M5 machine possess the most recent and fastest GPUs of the five machines, and as such better

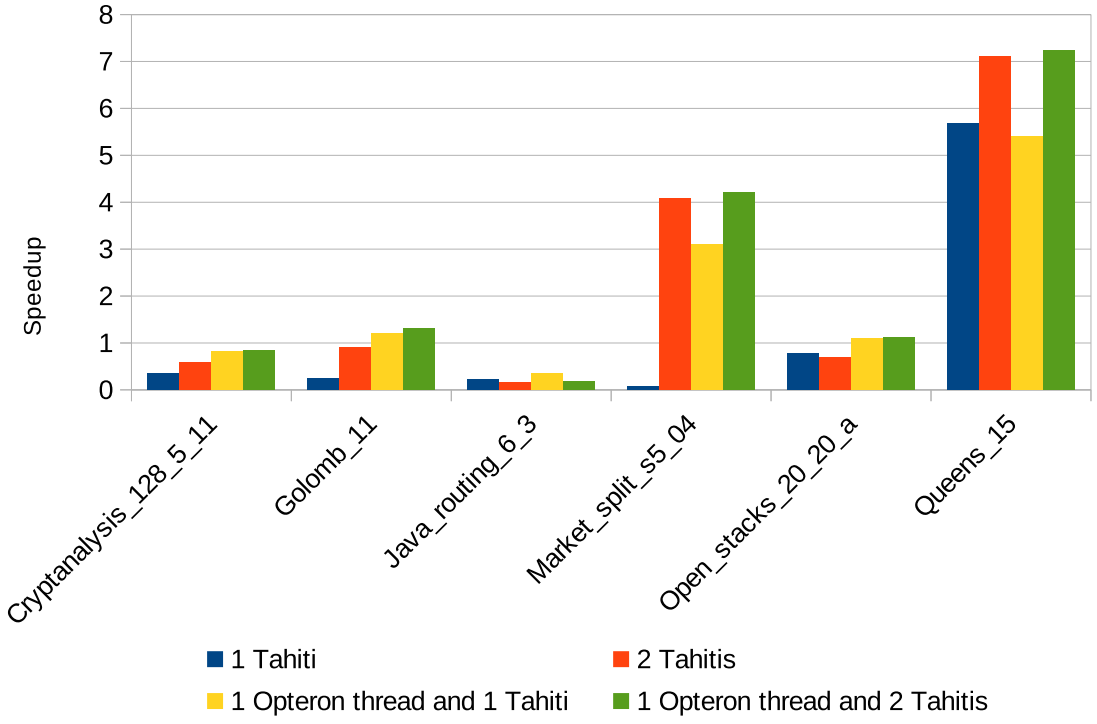

Figure 5.13: Speedups achieved with PHACT when comparing the elapsed times achieved with combinations of the devices on M4

Table 5.9: Seconds that PHACT took to solve each CSP when using combinations of the devices on M5

CSP	1 Xeon 3 thread	1 Titan	2 Titans	1 Xeon 3 thread and 1 Titan	1 Xeon 3 thread and 2 Titans
Bacp_7	109.64	29,52	694,37	397,74	551,12
Cryptanalysis_128_5_11	91.46	109,51	80,11	84,58	76,57
Golomb_11	127.91	85,75	34,81	42,80	31,09
Java_routing_6_3	42.26	13,08	26,46	25,71	22,33
Langford_3_18	91.93	59,44	757,73	266,19	280,04
M_queens_12	102.25	3474,46	3400,31	104,99	105,80
Market_split_s5_04	66.06	48,25	14,58	20,30	15,27
Open_stacks_20_20_a	85.97	67,00	117,07	94,13	86,77
Project_planning_13_7	210.27	923,78	1288,15	540,78	1563,70
Queens_15	84.20	7,10	5,62	6,47	5,75

speedups were expected. This was the case, as speedups were achieved with all the combinations of devices for four of the problems, and for seven problems when using 1 Titan. The best speedup was obtained when using the 2 Titans to solve the Queens_15 problem, for which they were about 15 times faster than 1 thread on the Xeon 3 CPU.

Figure 5.15 presents the speedups achieved by PHACT when comparing the elapsed times on solving

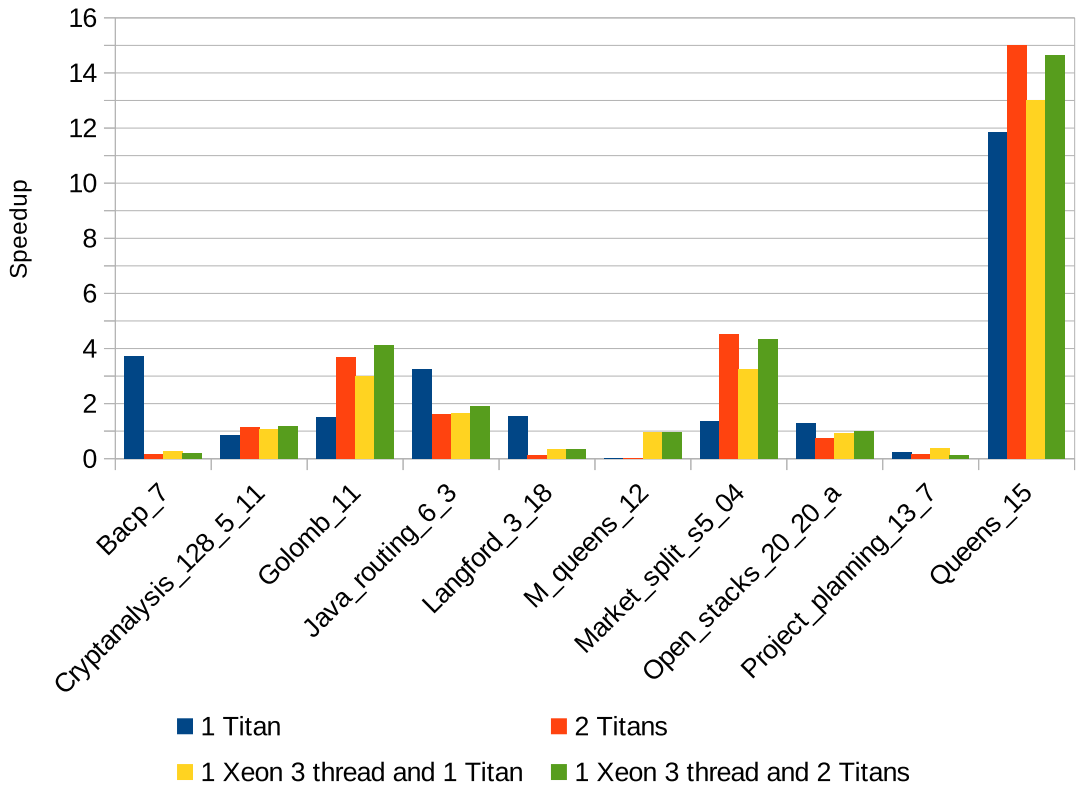

Figure 5.14: Speedups achieved with PHACT when comparing the elapsed times achieved with combinations of the devices on M5

10 CSPs with different combinations of the devices on M2. The respective elapsed times are shown in Table 5.10.

Table 5.10: Seconds that PHACT took to solve each CSP when using combinations of the devices on M2

CSP	1 Xeon 1 thread	1 MIC	2 MICs	1 Xeon 1 thread and 1 MIC	1 Xeon 1 thread and 2 MICs
Bacp_7	203.65	14.32	89.57	96.77	69.88
Cryptanalysis_128_5_11	146.64	37.46	74.28	58.20	57.20
Golomb_11	182.52	40.67	34.55	39.03	36.52
Java_routing_6_3	71.18	16.22	44.36	41.27	37.06
Langford_3_18	127.01	34.48	15.01	135.82	137.11
M_queens_12	160.57	828.61	234.88	99.64	97.38
Market_split_s5_04	114.64	20.81	17.30	23.24	17.98
Open_stacks_20_20_a	144.03	23.91	88.82	68.76	61.01
Project_planning_13_7	456.94	217.85	384.40	597.51	1,166.54
Queens_15	110.81	13.88	12.95	13.81	13.29

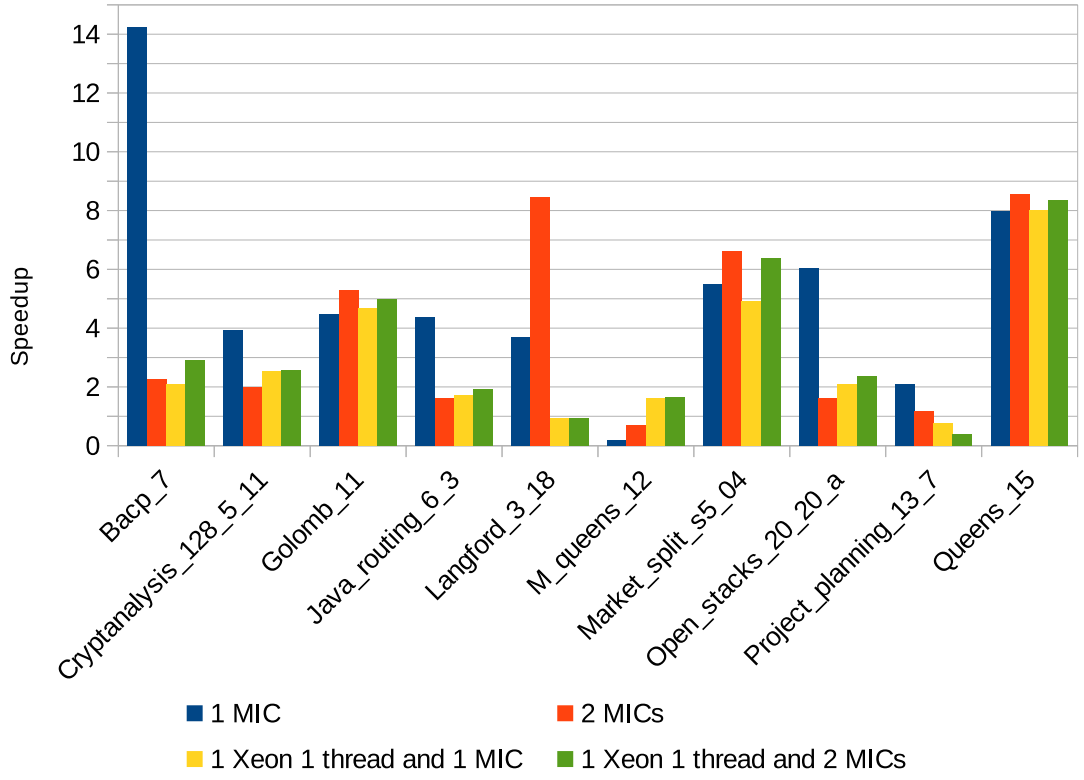

Figure 5.15: Speedups achieved with PHACT when comparing the elapsed times achieved with combinations of the devices on M2

The elapsed times when using one or two MICs against a single thread on the Xeon 1 CPU generated many different speedups. The speedups ranged from 0.2 when comparing the time needed by PHACT to solve the M_queens_12 problem with 1 MIC against 1 thread on the Xeon 1 CPU, up to 14.2 when making the same comparison, but for solving the Bacp_7 problem.

The Intel MICs hardware is much more similar to the one of the CPUs than to the one of the GPUs, which made PHACT achieve better results than the ones achieved with GPUs. One major difference is the number of threads that these devices need to execute simultaneously to achieve their best performance, namely about 240 threads. Unlike to the three GPUs used for testing PHACT, the MICs RAM (16 GB) is more than enough to fulfill all PHACT memory requirements to solve these 10 CSPs.

When looking at the chart, the greater difference in speedups achieved for a single problem occurred when solving the Bacp_7 problem. The only possible explanation for this difference in speedups is that the combination of the number of sub-search spaces generated to use 1 MIC with the number of threads used by the device, allowed a thread to pick the sub-search spaces that contained the best solution very early in the solving process. The same may have happened when solving the Open_Stacks_20_20_a. When dealing with the much unbalanced sub-search spaces of the Langford_3_18 problem, the best combination of numbers of threads and generated sub-search spaces favoured the usage of 2 MICs.

In the previous tables and charts of this section, the elapsed times were compared against a single thread on each machine CPU. The following tables and charts present the elapsed times and speedups achieved by PHACT when solving bigger instances of the same CSPs, but using all the processing power of all the

devices on each machine. Only the instance of the Langford problem is the same, as bigger instances would take too long to solve. Solving some problems in the five machines would take longer than twelve hours, so the respective results are not presented.

Even when using all the cores of the I7, of the Xeon 2, of the Opteron or of the Xeon 3 CPU, for some CSPs, the GPUs on the same machine, the Geforce, the Tesla, the Tahitis or the Titans, respectively allow to gain some speedup, as presented in Figures 5.16, 5.17, 5.18 and 5.19. The elapsed times used for calculating the speedups presented in Figure 5.16 are shown in Table 5.11.

Table 5.11: Seconds that PHACT took to solve each CSP on M1, when using the I7, or the I7 and the Geforce

CSP	I7	I7 and Geforce
Bacp_6	273.65	1,600.40
Cryptanalysis_128_5_14	212.53	270.91
Golomb_12	390.20	1,777.32
Langford_3_18	56.64	125.92
Market_split_s5_01	234.41	124.02
Open_stacks_20_20_b	193.29	224.39
Project_planning_13_8	51.74	5,353.03
Queens_17	732.13	319.22

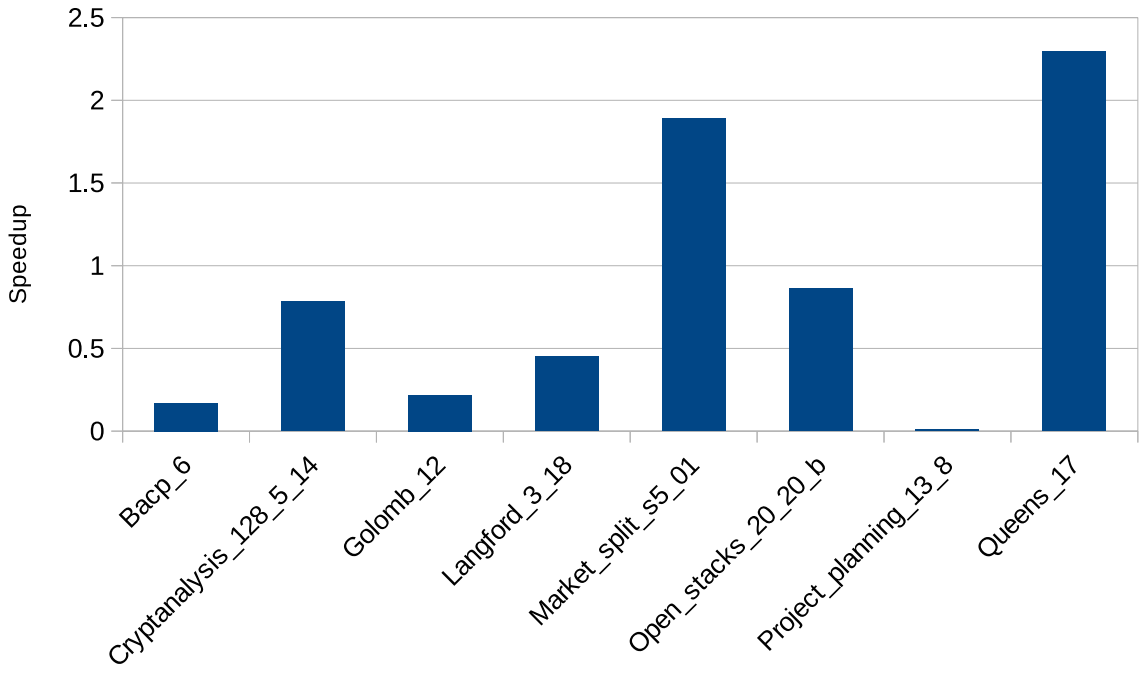

Figure 5.16: Speedups achieved with PHACT when using the Geforce GPU along with the I7 CPU on M1

However, when using such different devices as CPUs and GPUs simultaneously to solve the same CSP, multiple challenges arise, as described in Section 4.3. Mainly, more sub-search spaces are created, leading to more repeated propagations needed to explore all of them and some CPU resources are occupied while

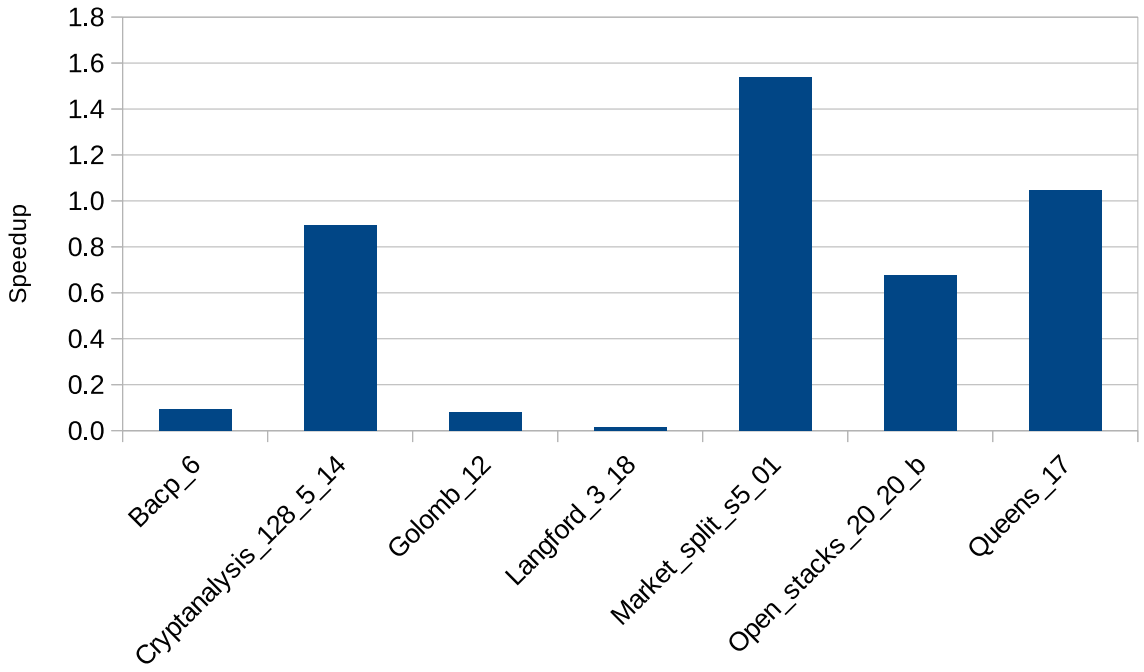

Figure 5.17: Speedups achieved with PHACT when using the Tesla GPU along with the Xeon 2 CPU on M3

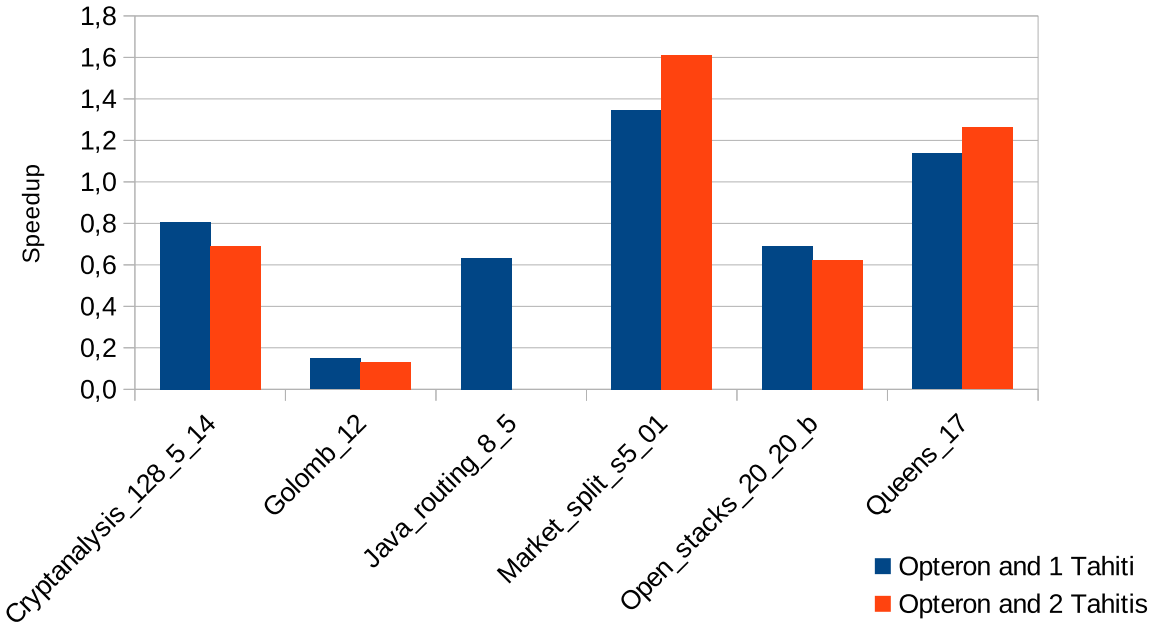

Figure 5.18: Speedups achieved with PHACT when using one and two Tahiti GPUs along with the Opteron CPU on M4

configuring, initializing and communicating with the other devices, instead of being used to solve the CSP.

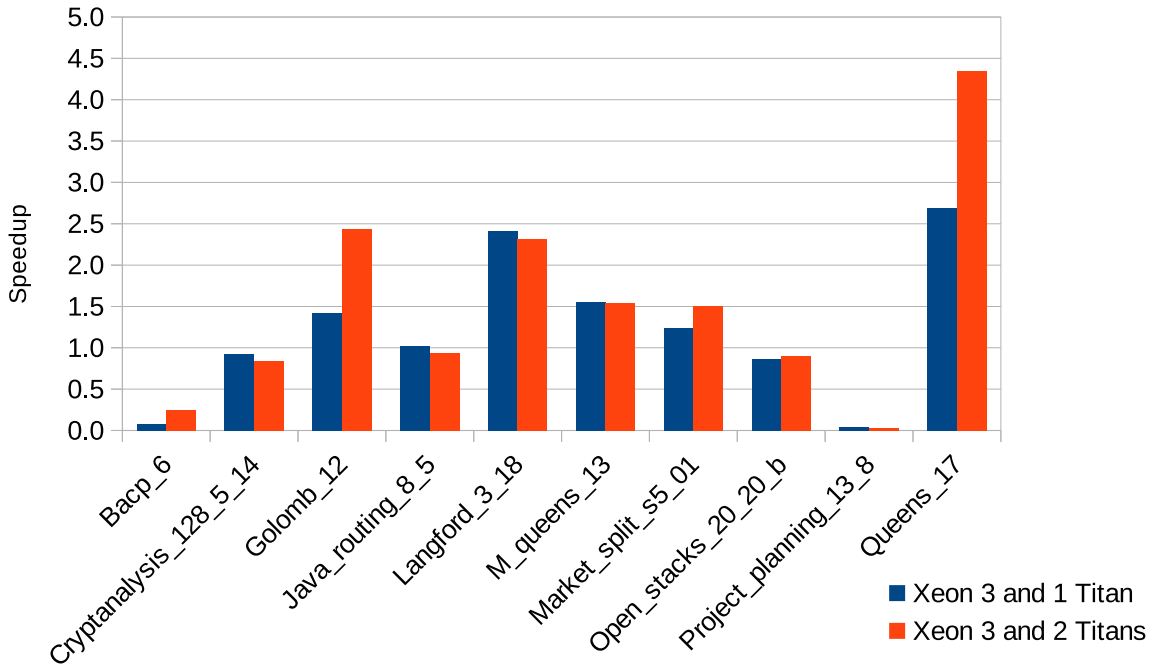

Figure 5.19: Speedups achieved with PHACT when using one and two Titan GPUs along with the Xeon 3 CPU on M5

This lead to the GPUs being unable to speed up the solving process of most of the problems when comparing with the time taken to solve them with all the CPU cores alone. Nevertheless, the Geforce allowed to speedup the solving process for two of the ten problems, achieving a top speedup of 2.3 with the Queens_17 problem. The Tesla and the Tahitis GPUs were also capable of speeding up the solving process for the same two problems. This means that, the work done by this three GPUs did not compensate the time spent by the respective CPU in controlling them, for most of the CSPs.

As expected, the Titan GPUs allowed the best speedups, achieving a speedup for five of the ten problems. This GPUs were the only ones capable of helping the respective CPU to solve all the 10 problems in less than twelve hours. The best speedup was achieved when solving the Queens_17 problem, being about 4.3 times faster than the Xeon 3 alone when using both GPUs to help.

When using one or both MICs to speedup the solving process of the Xeon 1 with all its threads, PHACT achieved speedups for seven of the ten problems, as presented in Figure 5.20.

PHACT creates blocks with less sub-search spaces when using more devices, to make them synchronize more frequently, and even fewer when solving optimization problems or looking for a single solution. That allows PHACT to minimize unnecessary work done while looking for a solution worse than another solution already found by another device, or when instead of optimizing, only one solution is to be found and another device has already found one.

However, that also increases the number of blocks that will be communicated to the devices and these communications consume time, which will negatively impact of the performance of PHACT. For example, when solving the Java_routing_8_5 problem, more than 200 blocks were communicated from the host to the devices.

For the Project_planning_13_8 problem another trouble emerged, namely, its unbalanced sub-search spaces

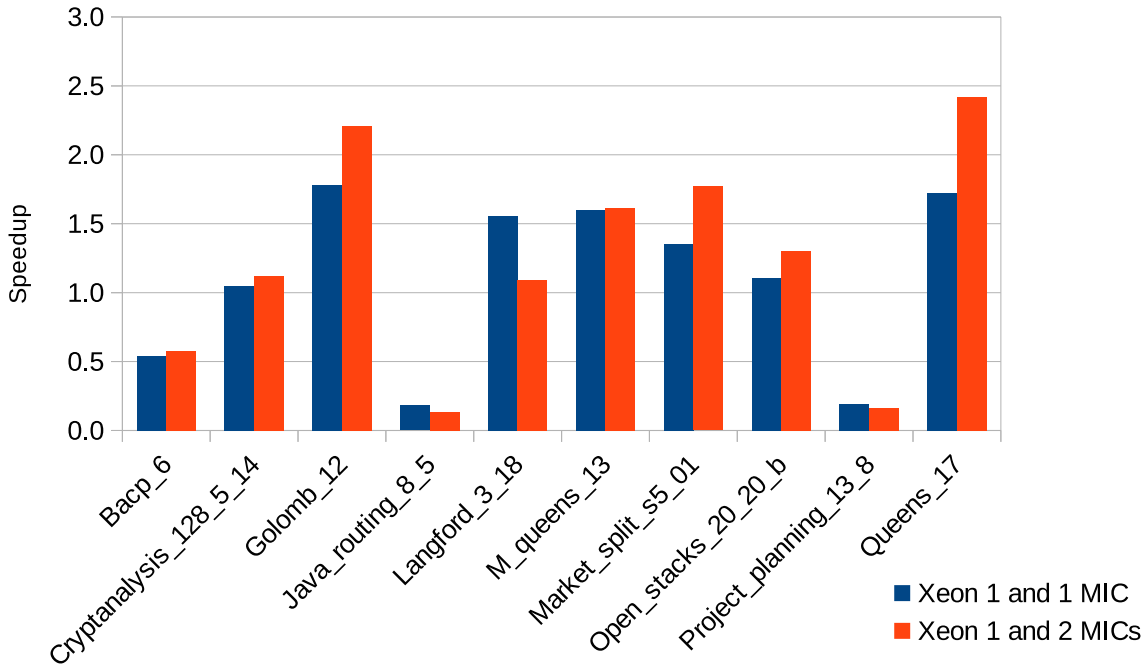

Figure 5.20: Speedups achieved with PHACT when using MIC 1 and MIC 2 accelerators along with the Xeon 1 CPU on M2

caused the MIC 2 to finish its work much after the other MIC and the Xeon 1. Before receiving the last block of sub-search spaces, the MIC 2 needed an *average time* to solve each sub-search space of about 3.6 ms, which lead the host to deliver to it a block with 321 sub-search spaces, a size similar to the one before, which took about 1.1 s to solve. However, this last block took 206 s to solve which made this device finish about 155 s after the other devices.

5.4.1 Conclusion

When using more than one device to solve a problem, the greater difficulty is to achieve a good load balancing between the devices while limiting the number and size of communications between them. That difficulty is even greater when dealing with devices with such different architectures as CPUs and GPUs. GPUs are also less prepared to deal with the divergent paths of the source code needed to solve the problems with a greater number and more complex constraints. That along with the limitations on the maximum number of threads that can be used to solve a problem, due to their RAM size, makes GPUs much less prepared to solve bigger and more complex CSPs than the CPUs.

However, PHACT was capable of achieving top speedups of about 13 and 14 when using one and two GPUs to help a single CPU core, respectively. The usage of one and two MICs to help a single CPU thread allowed to obtain top speedups of about 8. When using all the processing power of the CPUs, one GPU allowed to achieve a top speedup of about 2.7 and two GPUs of about 4.3, both against a 32 threads CPU. One MIC and two MICs allowed a top speedup of about 1.8 and 2.4, respectively, against a CPU with 32 cores.

5.5 Comparison with the state of the art

For comparing the speedups achieved by PHACT when using multithreaded CPUs, with the state of the art, Gecode [66], Choco [55] and OR-Tools [29] were selected. Gecode is one of the most used solvers for performance comparisons [63, 53, 15]. As such, it may also serve as a bridge to compare relative speedups with other solvers. The Choco and the OR-Tools use portfolio search strategies to allow a multithreaded resolution of a CSP and Gecode uses work-stealing techniques.

The tests that were presented in Section 5.3, were repeated with Gecode, Choco and the OR-Tools solvers and are presented in this section, along with a comparison of elapsed times and speedups between them.

Table 5.12 shows the time that each solver needed to solve each one of the CSPs with a different number of threads when using the Xeon 2 of the machine M3. This elapsed times were used to build Figure 5.24 and the problems were the ones also solved on each machine to produce Figures 5.21, 5.22, 5.23, 5.24 and 5.25.

Table 5.12: Seconds needed for each solver to solve the problems with different number of threads on Xeon 2

Threads	Solver	Bacp_7	Crypt-analysis 128_5_11	Golomb 11	Java routing 6_3	Langford 3_18	M queens 12	Market split s5_04	Open stacks 20_20_a	Project planning 13_7	Queens 15
1	PHACT	137.72	101.76	121.49	45.77	86.49	110.89	78.17	95.60	305.30	76.61
	Gecode	158.31	850.38	353.77	60.79	59.46	208.80	68.58	576.41	490.74	164.66
	Choco	346.03	789.78	375.43	113.99	104.00	326.85	104.31	598.96	869.76	100.32
	OR-Tools	0.48	36.94	1,304.47	8.37	38.13	30.25	150.71	13.55	2.48	5,212.78
2	PHACT	77.16	57.14	81.13	26.86	74.16	65.98	54.85	58.97	156.36	48.33
	Gecode	82.54	509.32	186.45	42.06	36.57	121.26	93.49	437.15	296.14	152.45
	Choco	0.93		150.81	1,486.59	2.03	5.29	572.54	4,216.97	100.02	
	OR-Tools	0.26		841.50	6.72	2.10	30.00	150.82	13.66	2.14	
4	PHACT	49.17	35.67	49.14	18.06	40.02	66.32	37.45	32.89	94.74	32.07
	Gecode	27.12	362.08	102.34	29.98	21.74	78.50	74.04	419.22	179.54	127.41
	Choco	1.26		145.23	1,503.61	2.52	5.34	89.49	909.93	117.45	
	OR-Tools	0.23		730.07	4.82	0.82	30.68	155.27	13.98	1.97	
8	PHACT	34.36	25.43	44.63	12.62	44.58	65.78	23.26	21.72	51.95	17.97
	Gecode	28.29	206.68	61.22	30.20	8.33	38.78	148.64	542.56	191.88	127.16
	Choco	1.57		141.38	1,518.42	2.87	4.03	211.87	1,031.34	109.30	
	OR-Tools	0.22		778.13	4.79	0.80	28.77	164.18	14.69	1.97	
16	PHACT	17.53	18.83	25.46	10.07	7.38	66.40	16.07	15.27	41.73	11.09
	Gecode	13.22	176.44	37.46	32.92	3.72	20.33	211.62	11.62	138.12	133.46
	Choco	2.02		142.15	1,832.61	3.70	4.69	447.85	1,039.62	107.62	
	OR-Tools	0.23		867.00	4.80	0.86	31.10	165.52	16.02	1.98	
32	PHACT	11.46	15.00	19.31	8.34	6.99	65.02	10.38	10.58	26.52	8.23
	Gecode	18.21	172.42	27.83	38.46	3.40	13.69	283.50	557.87	109.45	140.50
	Choco	3.40		224.64	2,246.25	5.94	9.67	489.58	1,447.13	184.60	
	OR-Tools	0.23		1,608.44	6.01	0.82	36.43	233.47	24.71	2.01	
40	PHACT	10.55	14.24	17.71	8.07	5.76	64.56	9.75	10.58	25.49	7.21
	Gecode	17.46	178.01	27.09	39.43	4.08	12.94	235.64	90.61	126.91	143.70
	Choco	4.17		229.40	2,689.61	7.25	15.44	645.44	1,721.67	209.69	
	OR-Tools	0.21		1,997.60	6.28	0.88	48.01	273.27	30.14	2.17	

Those figures present the speedups achieved by PHACT when comparing its elapsed times against the execution of the other three solvers on the M1, M2, M3, M4 and M5 CPUs, respectively. Each bar in the charts represents the geometric mean of the speedups achieved for the ten CSPs for a given number of threads.

The OR-Tools and the Choco solvers do not allow the search for all the solutions of a CSP while us-

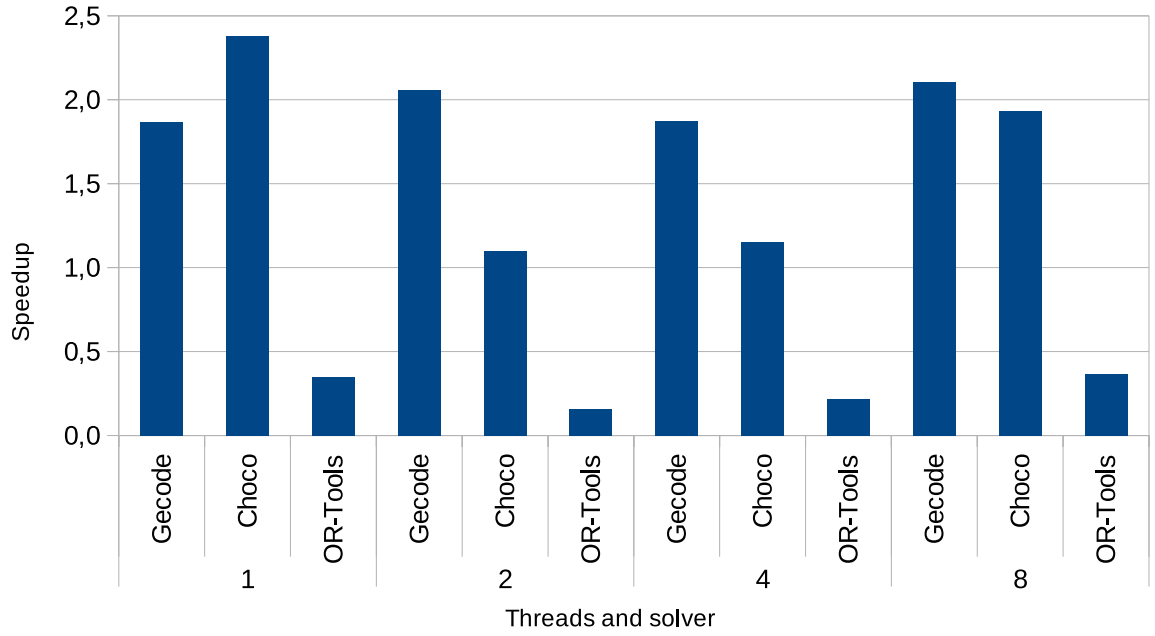

Figure 5.21: Average speedups achieved with PHACT against Gecode, Choco and OR-Tools when using from 1 to 8 threads on I7

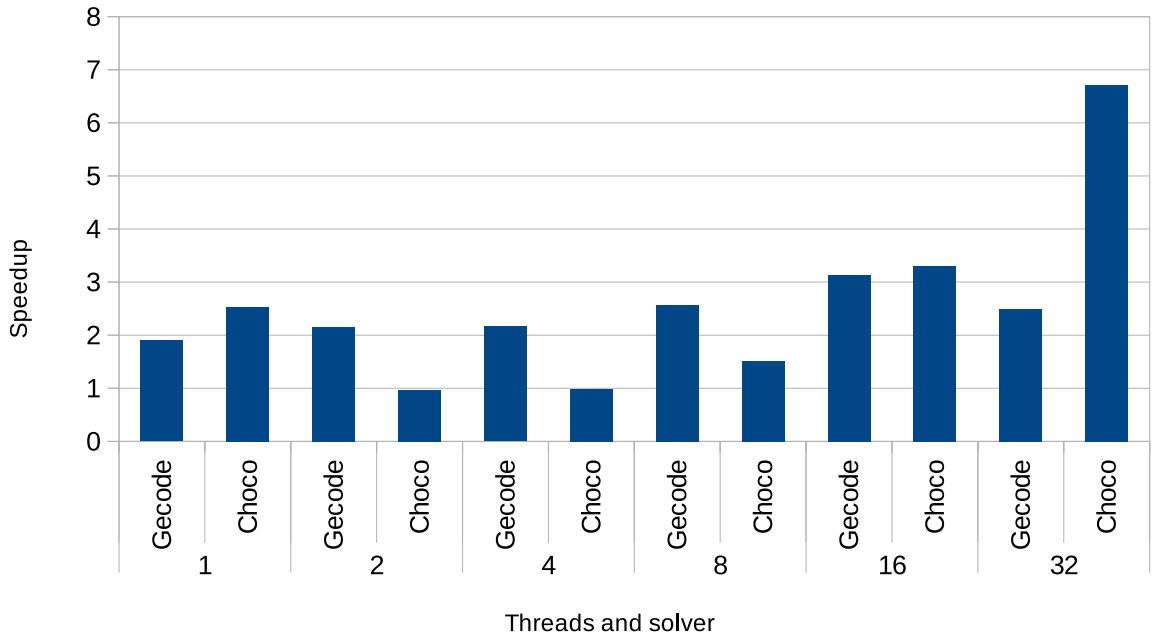

Figure 5.22: Average speedups achieved with PHACT against Gecode, and Choco when using from 1 to 32 threads on Xeon 1

ing more than one thread. As such, there are no values presented for counting all the solution for the Cryptanalysis_128_5_11 and the Queens_15 problems for these two solvers when using more than one

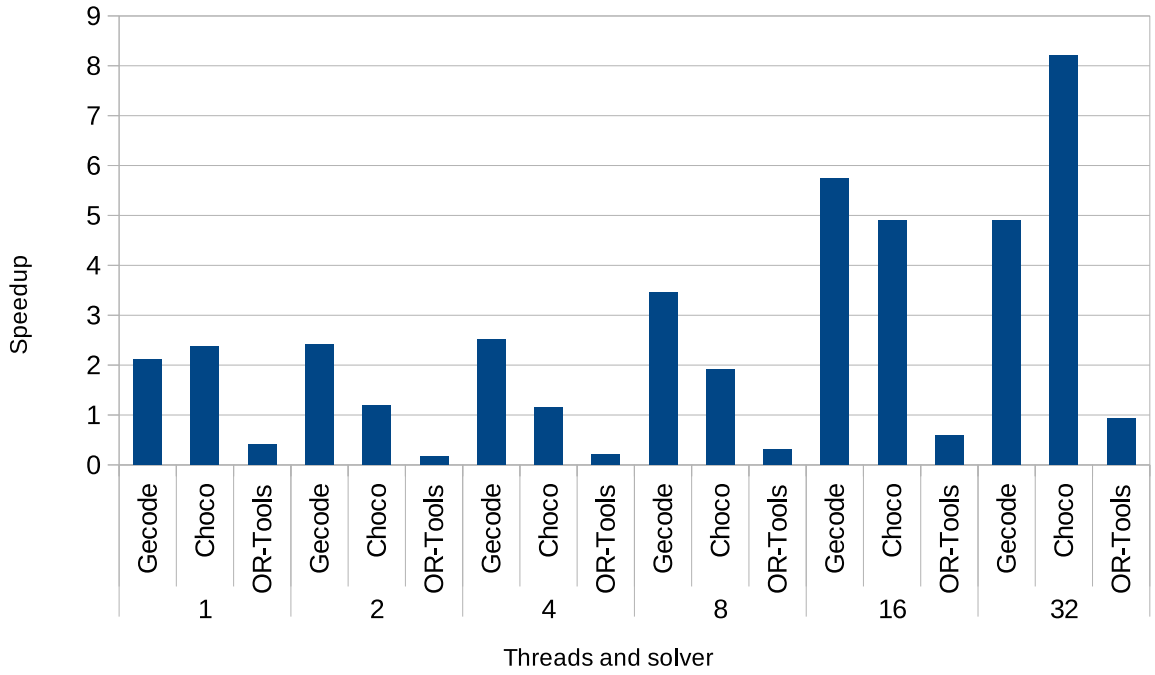

Figure 5.23: Average speedups achieved with PHACT against Gecode, Choco and OR-Tools when using from 1 to 32 threads on Xeon 3

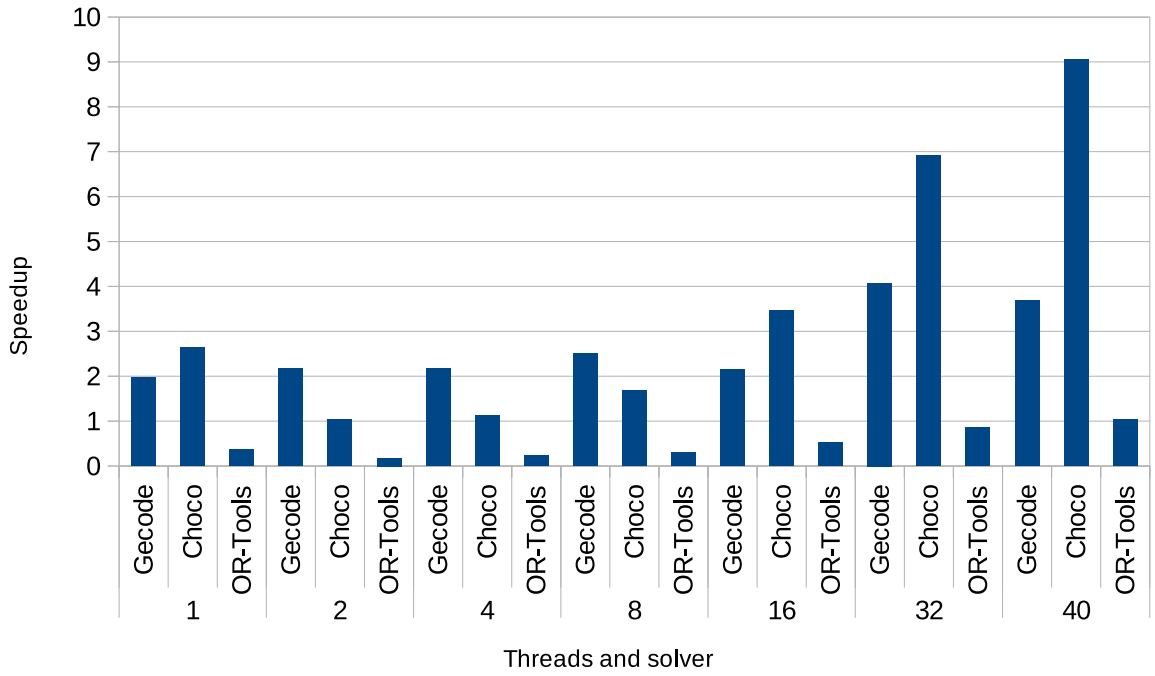

Figure 5.24: Average speedups achieved with PHACT against Gecode, Choco and OR-Tools when using from 1 to 40 threads on Xeon 2

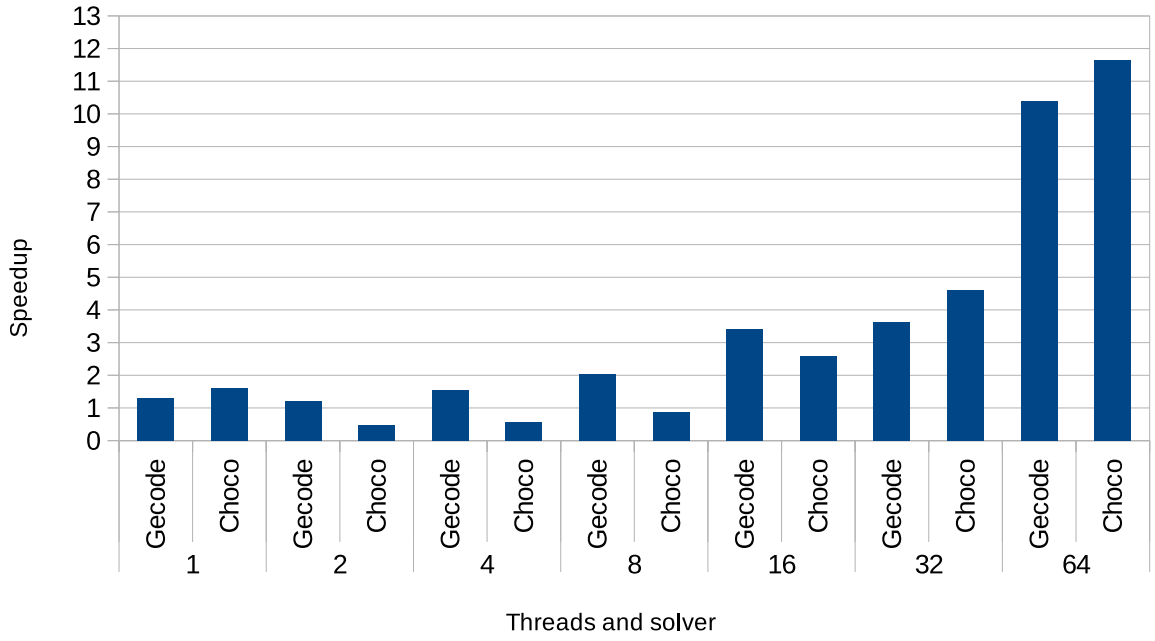

Figure 5.25: Average speedups achieved with PHACT against Gecode and Choco when using from 1 to 64 threads on Opteron

thread.

From the five charts in Figures 5.21, 5.22, 5.23, 5.24 and 5.25 it is visible that, in average, PHACT achieve greater speedups than Gecode with all the number of threads on the five machines. However, looking at the elapsed times presented in Table 5.12, we can see that Gecode was faster in some cases, namely when solving the Bacp_7, the Langford_3_18 and the Market_split_s5_04 with fewer threads. When using only one thread, this can only be due to Gecode using more efficient propagators to solve these problems, or by a more efficient simplification of the model prior to exploring it, which may avoid some work during the exploration process.

However, when using more threads, PHACT was faster in all the problems, except when solving the Langford_3_18 and the M_queens_12 problems. When solving the Langford_3_18, the difference in time between the solvers corresponds to the time that PHACT needs to find, configure and initialize the OpenCL devices, even when running only on the CPU. For the M_queens_12, Gecode was faster than PHACT when using more than 4 threads, which shows that, for this problem the work stealing technique was more efficient.

When comparing the times for solving the problems with a single thread against the CHOCO solver, PHACT was faster than it for all the problems. When using more threads, the elapsed times diverge greatly between problems and number of threads used. This is due to the techniques used by Choco to use multithreading CPUs to solve a problem. It uses a portfolio of workers where each worker (thread) will use a different strategy to try to find a near optimal solution, when optimizing, or a solution when just looking for one. However, this strategy does not guarantee that the best solution will be found when optimizing a problem, and although for some problems it seems to be efficient, for other problems it takes too long to find a near optimal solution.

The OR-Tools uses a similar strategy to the one used by Choco, but its results are in most cases, better

than the ones achieved by Choco. Even when using a single thread to solve some problems, the OR-Tools is much faster than the other three solvers, as for example, when solving the Bacp_7 problem. However, for some other CSPs, it is much slower, as for example, when solving the Queens_15 problem.

Figures 5.21, 5.22, 5.23, 5.24 and 5.25 allow to check the average speedup of PHACT against the other three solvers when solving the ten CSPs on the five machines. From there we can also see that Choco achieved a greater performance than PHACT in some cases. However, just like when using the OR-Tools solver, when optimizing with more than one thread, the solution found is only near optimal. In average, OR-Tools was faster with all the different number of threads used. OR-Tools was not used in M2 and M4 machines as it does not support the older versions of the software libraries installed in them.

The Table 5.13 presents the time that each solver took to solve bigger instances of the same problems as before, but when using all the threads of the Xeon 2 CPU. The respective speedups are represented in Figure 5.26.

Table 5.13: Seconds needed for different solvers to solve problems on the Xeon 2

CSP	PHACT	Gecode	Choco	OR-Tools
Bacp_6	75.63	544.61	4.10	0.25
Cryptanalysis_128_5_14	53.55	1,951.26		
Golomb_12	172.00	299.08	2,289.25	27,479.99
Java_routing_8_5	231.30	2,914.79		21.58
Langford_3_18	5.57	3.08	7.02	0.89
M_queens_13	360.15	72.41	8.72	51.04
Market_split_s5_01	131.53	3,139.55	6,034.04	2,066.21
Open_stacks_20_20_b	54.40	637.39	14,652.00	22.86
Project_planning_13_8	21.09	123.20	218.67	1.85
Queens_17	190.68	6,234.38		

From Table 5.13 we can see that, although each problem took more time to solve, in most cases the comparison between solvers remains similar to the one in Table 5.12, where smaller instances of the same problems were solved.

Figure 5.26 shows that PHACT was faster than Gecode for eight of the ten problems. For the Langford_3_18 and the M-queens_13, PHACT generates very unbalanced sub-search spaces which results in some of the threads finishing their work much before the others, resulting in a decrease of teh performance of PHACT. For this situations, the work-stealing techniques like the ones used by Gecode become more efficient, as they may share the remaining work among all the threads.

As for Choco and the OR-Tools solvers, they are unable to find all the solutions, and as such no result is presented for solving the Cryptanalysis_128_5_14 and the Queens_17 problems. For the Java_routing_8_5 problem no result is presented for Choco, as after 12 hours it was yet trying to solve the problem, and it was terminated.

PHACT was faster than Choco and the OR-Tools solvers when finding one solution for Market_split_s5_01 problem, as this problem contains only a single solution, which is problematic for portfolio search, when none of the used heuristics allow to find the solution faster.

For some of the optimization problems, Choco and OR-Tools were faster than PHACT, however, the solutions found by these two solvers may not be the best one and PHACT guarantees that the solution it

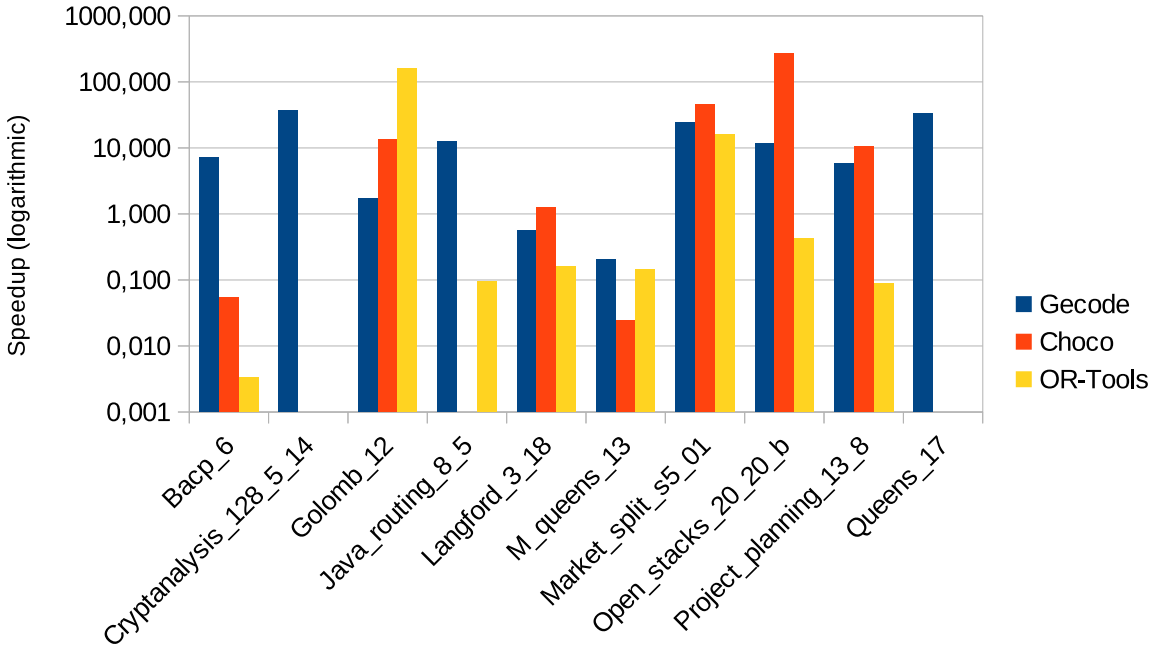

Figure 5.26: Speedups of PHACT against Gecode, Choco and OR-Tools when solving ten problems on the Xeon 2 with 40 threads

finds is one of the best, as many solutions can exist with the same cost.

Figure 5.27 presents the geometric mean speedups achieved by PHACT against the other three solvers, when solving the same problems shown in Table 5.13, but using only the CPU, or all the devices on each machine. Note that, when trying to solve some of the problems in some machines or devices, PHACT and Choco took longer than twelve hours, and as such, these results were not considered when building the chart of this figure. Namely, the results for the Java_routing_8_5 and the M_Queens_13 problems when running PHACT on all the devices of M1 and the results for the Bacp_6, the Java_routing_8_5, the Langford_3_18, the M_Queens_13 and the Project_planning_13_8 problems when running PHACT on all the devices of M4. For the same motive, the Java_routing_8_5 problem was not solved by Choco in any of the five machines.

PHACT achieved an average top speedup of about 11.9 when using 64 threads on the Opteron, comparing against Gecode. Comparing with Choco, it achieved better speedups in more than half of the situations. In average, the OR-Tools was faster than the other three solvers on the machines where it was executed.

When using all the devices on each machine (M1, M2, M3, M4 and M5), PHACT achieved speedups in the same executions as when using only the CPU on the machine, but in a lesser degree. This was mainly due to the problems used for this benchmark being the most complex used in all the tests presented in this thesis, which makes them very hard to solve in GPUs due to their hardware limitations. The exception was when using all the devices on the M4 machine. In this case, the OpenCL compiler presented a strange behavior, as although the kernel for each device is compiled by different threads, they seemed to have been compiled sequentially, and to take much longer to compile it for the Tahitis than for the other devices.

For example, when solving the Cryptanalysis_128_5_14 problem, compiling the kernel for the Opteron took about 1 s, for the first Tahiti took about 23 s and for the other Tahiti took about 46 s. This means that only after 46 s all the devices were actually solving the problem that Gecode took 27 s to solve. For

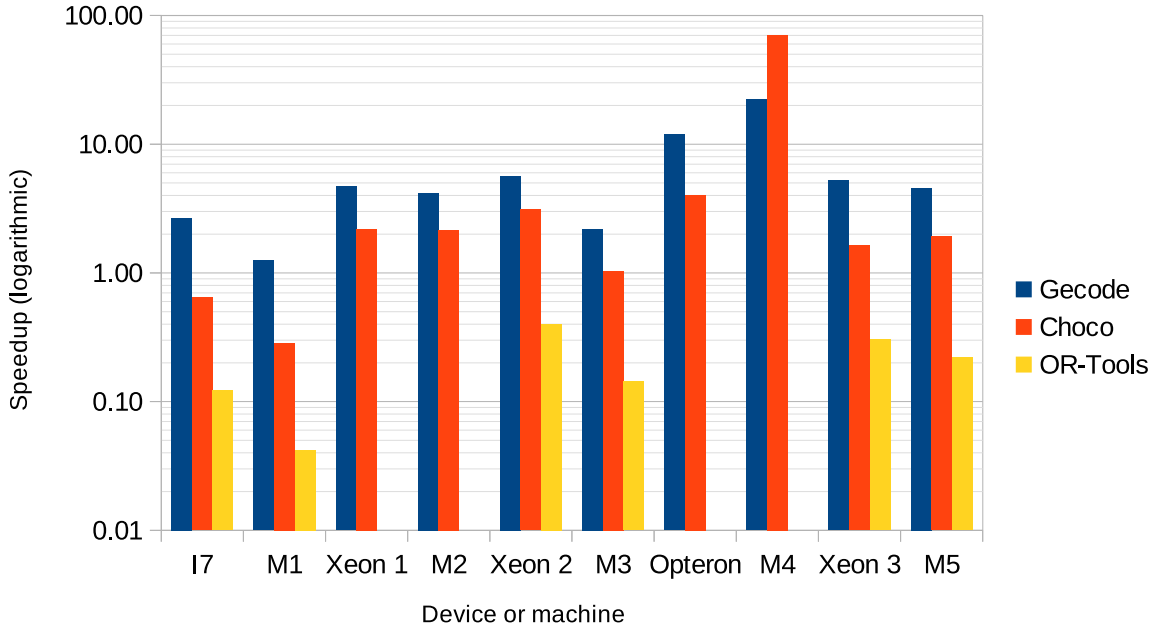

Figure 5.27: Speedups of PHACT against Gecode, Choco and OR-Tools on different devices

the same problem, the compilation of the kernel for the three devices of M5 took less than 0.2 s.

When using all the devices of M4, only five of the ten problems were solved in less than 12 hours. However, these five problems allowed PHACT to achieve a greater speedup when using all the devices, than when using only the Opteron CPU.

5.6 Conclusion

In this chapter one or two GPUs and one or two MICs were used to help a CPU solving some problems. When comparing the time taken for one or two devices to solve the problems against a single CPU thread it was possible to achieve speedups for most of the problems, achieving speedups of up to 15 with two GPUs and up to 14 with a single MIC. When comparing the elapsed times of each CPU with all its cores, against the same CPU aided by one or two devices, the speedups were much reduced or even inexistent. However, PHACT achieved speedups of up to 4.3 with two GPUs and up to 2.4 with two MICs.

To the present date, it appears that it does not exist any constraint solver capable of using MICs or GPUs, so it is not possible to compare the performance of PHACT against other constraint solvers when using these devices. In Chapter 3 the NVIDIOSO solver was presented. According to Campeotto *et al.* [15], this solver is capable of using an Nvidia GPU together with a CPU to solve CSPs, however that solver was discontinued, and no source code is available which would allow to compare it against PHACT.

Nevertheless, a different branch of PHACT was implemented with load balancing techniques between devices similar to the ones described in Campeotto *et al.* [15]. However, due to the number and size of data transfers needed between the devices and the host, which are very time consuming operations, that branch was rapidly abandoned.

The current major week points of PHACT are related with OpenCL limitations and although some of them

have already been removed in newer versions of this framework, most of the devices are not yet compatible with them. The biggest problem is the lack of synchronization between devices, which would be much helpful to improve the load balancing between them. Although the load balancing techniques implemented in PHACT try to achieve a good load balancing between the devices, when running on devices with very different speeds to solve the same problem, and the sub-search spaces generated for that problem are very unbalanced, these techniques are not effective to achieve a good load balancing.

To mitigate the cases where a very slow device finishes its work much later than the other devices, or even to control the time that each device would be without communicating with the host, a timer could be used on each device. That timer would allow the device to decide to stop working and to communicate with the host. However, OpenCL does not posses such a timer in the devices.

Nevertheless, newer GPUs are improving their performance to handle more complex problems and some of them are already compatible with OpenCL 2.0, which may help to improve the load balancing techniques implemented in PHACT.

6

Conclusions and future work

At the beginning, the objective of this project was to develop a new constraint solver capable of using GPUs to speedups the solving process of constraint problems. However, rapidly it was found that it would not be an easy task due to the dynamic nature of constraint solving going against the architectural characteristics of GPUs.

At the same time, it was found that the only programming language capable of working with both AMD and Nvidia GPUs, OpenCL, was also capable of working with most GPUs, CPUs and MICs, which lead to the development of PHACT, which, at the current date seems to be the only existent constraint solver capable of using all these devices at the same time to solve a constraint problem.

Several tests were made on GPUs to check out their capabilities when solving constraint problems using the backtracking paradigm. In the end, it was found that although their shared memory is very fast, for most of the CSPs, its size is not enough to be used for practical speedups gains. When considering their dual parallelism capabilities (work-groups and work-items), it was found that for most of the CSPs that were used, the best combination is using 512 work-groups with 128 work-items, each. However, for bigger problems, those numbers needed to be reduced for them to fit in the GPU memory.

One of the biggest challenges when using several devices to solve the same constraint problem, is the need for their synchronization, which when using OpenCL can only be done on the host side. This implies that the only method to check on the other devices progress is for each device to stop the solving process and to communicate with the host. These communications are very time consuming, and a balance was made between their frequency and the amount of work needed to be done.

For that purpose, the search space is split in multiple sub-search spaces that are distributed to the devices in blocks with different amounts ou sub-search spaces, depending on the performance of each device when solving the current CSPs. That process is dynamic and the size of each block will vary along the solving process of the CSP, and is also different when optimizing, finding one solutions, or counting all the solutions.

To reduce the number and size of data transfers between the host and the devices, the full CSP is only passed once to each device, at the beginning of the solving process, and after that, each block of sub-search spaces is represented only by a numeric identifier of the first and last sub-search space to solve.

PHACT was tested when solving ten different CSPs, for two or three different sizes of each. When comparing with the state of the art solver, Gecode, which uses work stealing techniques, PHACT was faster in most of the sequential tests, and in most of the parallel executions, being capable of better harnessing the parallel processing power of multi-threaded CPUs. When comparing parallel executions in a CPU with 64 cores, PHACT achieved an average speedup of more than 11 when compared to Gecode. The obtained results seems to indicate that the search space splitting techniques implemented in PHACT achieve better results than the work stealing techniques used by Gecode for load balancing when using many threads.

The performance of PHACT was also compared with the ones of Choco and the OR-Tools solvers. However, these two solvers do not guarantee that a best solution will be found when optimizing a problem, neither can count all the solutions when using multiple threads. Nevertheless, in average, PHACT was faster than Choco in more than half of the problems, but the OR-Tools was faster than PHACT in most of the problems.

To the current date, it seems that it does not exist any constraint solver capable of using GPUs or MICs, so PHACT capabilities to use these devices were not compared with any state of the art solver. However, using a GPU instead of a single CPU thread allowed to achieved a top speedup of 12, although for some more complex problems and when optimizing, no speedups were achieved. Nevertheless, even when using all the threads of a CPU, if aided by two GPUs, speedups of up to 4.3 were obtained, and of up to 2.4 when helped by two MICs.

In overall, PHACT is capable of using any device compatible with OpenCL to solve a CSP, and most of its current limitations are due to the small size of the RAM of the GPUs, and to their inefficiency to deal with divergent paths, which is inherent to constraint propagation and backtracking. However, every new generation of GPUs makes them more comparable to CPUs and brings much more resources, such as RAM. PHACT is ready to use these new devices, and its load balancing techniques are easy to adjust, if needed.

6.1 Future work

PHACT can load CSP models using its own C interface or the MiniZinc/FlatZinc language and, at the current date it has 38 constraints implemented, each one including a reified version. The FlatZinc interpreter is yet under development, and it will be improved and extended to allow the input of more CSP models, and if needed, new constraints may also be added to PHACT.

All the load balancing techniques implemented in PHACT are achieving relatively good results, except when using devices with major differences in performance when solving the most complex CSPs. This point needs

to be further analysed and improved in the future.

Although PHACT uses the OpenCL programming language version 1.2 to allow the usage of most of the existent devices, a new version 2.x already exists and is already compatible with some devices. This new version of OpenCL introduces some key features that would be much useful in PHACT, namely new atomic operations that allow a device to communicate with the host without stopping its execution, and pointer sharing between the host and the devices. Those two features may greatly improve PHACT performance by reducing the costs for synchronization between the host and the device while making it much more frequently.

Another point of interest, consists in adding distributed capabilities for PHACT, whose current load balancing techniques between devices could now be applied between machines.

A

Implementing a CSP in PHACT

Besides loading a MiniZinc/FlatZinc model of a CSP, PHACT can solve a problem modeled with its own C programming language interface. Several examples of CSPs modeled in C for PHACT are distributed with the PHACT source code in the "csps" folder, and a new one can be implemented in the "csps/CSP.c" file.

To add a new CSP variable to the C model, three functions can be used :

```
// val - value to assign to the variable
unsigned int v_new_val(unsigned int val);

// vals - vector with all the domain values
// n_vals - number of values in the domain of the variable
// to_label - true if this variable should be labeled
unsigned int v_new_vals(unsigned int* vals, unsigned int n_vals, bool
    to_label);

// min - minimum value of the domain
// max - maximum value of the domain
```

```
// to_label - true if this variable should be labeled
unsigned int v_new_range(unsigned int min, unsigned int max, bool
    to_label);
```

Each function to create the CSP variables returns a unique identifier that will be used to associate the variables with the constraints and to output its value after solving the CSP.

At the time when this thesis was written, PHACT had 38 constraints implemented, each one with two functions, one for the reified version, and another one without reification. The name of the reified function of each constraint is the same as the one without reification, but with "_reif" added to its end. A new argument indicating the identifier of the reification variable is also added to the end of the list of arguments. For example, the "not equal" has the next two functions:

```
c_ne(unsigned int x_id, unsigned int y_id);
c_ne_reif(unsigned int x_id, unsigned int y_id, int reif_v_id);
```

The calls to the 38 constraints are listed bellow, where:

- x_id, y_id, z_id or var_to_opt_id - identifier of the respective variable;
- X_ids or Y_ids - array with the identifiers of the variables returned on their creation;
- n_vs - number of identifiers of variables in the X_ids or Y_ids array;
- n, k, c - integer value;
- K - array with integer values;
- n_consts - number of integer values in the K array;
- S_ids - array with the identifier of the variables of all the sets, ordered by set;
- n_sets - number of sets in the S_ids array;
- N_vs - number of variables in each set, ordered by set.

```
// All variables in X_ids[n_vs] must be assigned a different value
c_all_different(unsigned int* X_ids, unsigned int n_vs);

// At least m of the variables in X_ids[n_vs] must be assigned value k
c_at_least(unsigned int m, unsigned int* X_ids, unsigned int n_vs,
    unsigned int k);

// Any two sets S_ids[N_vs] from n_sets sets can only have 1 variable
    assigned with the same value
c_at_most_one(unsigned int* S_ids, int* N_vs, unsigned int n_sets);

// At most m of the variables in X_ids[n_vs] must be assigned value k
```

```
c_at_most(unsigned int m, unsigned int* X_ids, unsigned int n_vs,
    unsigned int k);

// The logical AND between the variables in X_ids[n_vs] must be assigned
    the same value as y_id
c_bool_and(unsigned int* X_ids, unsigned int n_vs, unsigned int y_id);

// The variable x_id must be assigned 1 or the variable y_id must be
    assigned 0
c_bool_clause(unsigned int x_id, unsigned int y_id);

// The logical OR between the variables in X_ids[n_vs] must be assigned
    the same value as y_id
c_bool_or(unsigned int* X_ids, unsigned int n_vs, unsigned int y_id);

// The boolean variable x_id must be assigned the same value as the
    variable y_id
c_bool2int(unsigned int x_id, unsigned int y_id);

// The variable z_id must be assigned value K[y_id] from K[n_consts]
c_element_int_var(int* K, unsigned int n_consts, unsigned int y_id,
    unsigned int z_id);

// The variable z_id must be assigned the same value as variable
    X_ids[y_id] from X_ids[n_vs]
c_element_var(unsigned int* X_ids, unsigned int n_vs, unsigned int y_id,
    unsigned int z_id);

// The variable X_ids[y_id] from X_ids[n_vs] must be assigned value k
c_element(unsigned int* X_ids, unsigned int n_vs, unsigned int y_id,
    unsigned int k);

// The variable x_id must be assigned the same value as variable y_id
c_eq_var(unsigned int x_id, unsigned int y_id);

// The variable x_id must be assigned value k
c_eq(unsigned int x_id, unsigned int k);

// The value assigned to variable y_id must be the number of variables
    in X_ids[n_vs] assigned with value c
c_exactly_var(unsigned int* X_ids, unsigned int n_vs, unsigned int c,
    unsigned int y_id);

// Exactly c variables from X_ids[n_vs] must be assigned value k
c_exactly(unsigned int* X_ids, unsigned int n_vs, unsigned int c,
    unsigned int k);

// Expands to multiple "c_ne" constraints
c_fake_all_different(unsigned int* X_ids, unsigned int n_vs);
```

```
// The value assigned to variable x_id must be greater than or equal to
   the value assigned to variable y_id
c_ge(unsigned int x_id, unsigned int y_id);

// The value assigned to variable x_id must be greater than the value
   assigned to variable y_id
c_gt(unsigned int x_id, unsigned int y_id);

// The value assigned to variable x_id must be less than or equal to the
   value assigned to variable y_id
c_le(unsigned int x_id, unsigned int y_id);

// The internal product of the values K[n] with the variables Y_ids[n]
   must be less than the value c
c_linear_lt(int* K, unsigned int* Y_ids, unsigned int n, int c);

// The internal product of the values K[n] with the variables Y_ids[n]
   must be different than the value c
c_linear_ne(int* K, unsigned int* Y_ids, unsigned int n, int c);

// The internal product of the values K[n] with the variables Y_ids[n]
   must be the value assigned to the variable z_id
c_linear_var(int* K, unsigned int* Y_ids, unsigned int n, unsigned int
   z_id);

// The internal product of the values K[n] with the variables Y_ids[n]
   must be the value c
c_linear(int* K, unsigned int* Y_ids, unsigned int n, int c);

// The value assigned to variable x_id must be less than the value
   assigned to variable y_id
c_lt(unsigned int x_id, unsigned int y_id);

// The variable z_id must be assigned with the biggest value between the
   ones assigned to the variables x_id and y_id
c_max(unsigned int x_id, unsigned int y_id, unsigned int z_id);

// Assign the biggest value to var_to_opt_id
c_maximize(unsigned int var_to_opt_id);

// The variable z_id must be assigned with the smallest value between
   the ones assigned to the variables x_id and y_id
c_min(unsigned int x_id, unsigned int y_id, unsigned int z_id);

// Assign the smallest value to var_to_opt_id
c_minimize(unsigned int var_to_opt_id);

// The value assigned to the variable x_id minus the value assigned to
```

```
    variable y_id must be value k
c_minus_eq(unsigned int x_id, unsigned int y_id, int k);

// The value assigned to the variable x_id minus the value assigned to
    variable y_id must be different than value k
c_minus_ne(unsigned int x_id, unsigned int y_id, unsigned int k);

// The values assigned to the variables x_id and y_id must be different
c_ne(unsigned int x_id, unsigned int y_id);

// The internal product of the variables X_ids[n_vs] with the variables
    Y_ids[n_vs] must be value k
c_sum_prod(unsigned int* X_ids, unsigned int* Y_ids, unsigned int n_vs,
    unsigned int k);

// The sum of the values assigned to the variables X_ids[n_vs] must be
    equal to the value assigned to variable y_id
c_sum_var(unsigned int* X_ids, unsigned int n_vs, unsigned int y_id);

// The sum of the values assigned to the variables X_ids[n_vs] must be
    equal to the value k
c_sum(unsigned int* X_ids, unsigned int n_vs, unsigned int k);

// The value assigned to variable x_id must be equal to the value
    assigned to variable y_id minus the value assigned to variable z_id
c_var_eq_minus(unsigned int x_id, unsigned int y_id, unsigned int z_id);

// The value assigned to variable x_id must be equal to the modulo of
    the value assigned to variable y_id minus the value assigned to
    variable z_id
c_var_eq_minus_abs(unsigned int x_id, unsigned int y_id, unsigned int
    z_id);

// The variable x_id must be assigned with the value assigned to the
    variable y_id added to the value assigned to variable z_id
c_var_eq_plus(unsigned int x_id, unsigned int y_id, unsigned int z_id);

// The variable x_id must be assigned with the value assigned to the
    variable y_id multiplied by the value assigned to variable z_id
c_var_eq_times(unsigned int x_id, unsigned int y_id, unsigned int z_id);
```

After using the functions above to create the CSP variables and constraints, the function "solve_CSP()" will begin the solving process. This function will return the number of solutions that were found, or 1 if optimizing and a best solution was found or if only one solution is wanted and one solution was found.

After compiling PHACT[1], which will compile the "csps/CSP.c" file, the CSP may be solved to count all the solutions, to find one solution or for optimization, according to the command line arguments introduced

[1] Compilation instructions for PHACT are described in Appendix B.

when executing PHACT. In the "csps/CSP.c" file, three flags may be used to detect with which purpose the problem is being solved:

- OPTIMIZING - true if optimizing;
- FINDING_ONE_SOLUTION - true if looking for a single solutions;
- COUNTING_SOLUTIONS - true if counting all the solutions.

These flags may be used to implemented different outputs for the solver when solving the "csps/CSP.c" problem, according to the solving objective. To output the value of the CSP variables, three function may be used:

```
// output the offset value added to the value assigned to the variable
   whose identifier is v_id
v_print_single_val(unsigned int v_id, unsigned int offset);

// output the offset value added to the value assigned to the variables
   whose identifiers are in the vs_id array
// n_vs_id is the number of variables that are in the vs_id array
vs_print_single_val(unsigned int* vs_id, unsigned int n_vs_id, unsigned
   int offset);

// when optimizing, this will output the offset value added to the value
   assigned to the variable representing the cost to optimize
v_print_cost(unsigned int offset);
```

After implementing the CSP in "csps/CSP.c" and recompiling PHACT, it may be executed using the command line arguments presented in Appendix B.

B

Compiling and executing PHACT

PHACT can use all the devices compatible with OpenCL on a machine to solve the CSP modeled in the FlatZinc file "CSP.fzn" just by executing the command "./PHACT CSP.fzn". With that command, PHACT will solve that CSP using the default number of sub-search spaces for load-balancing, presented in Section 4.2, and the default number of work-groups and work-items per device, as shown in Section 2.3.

As described in the "readme.txt" file distributed with PHACT source code, also in the Appendix B and shown when executing the command "./PHACT -H", PHACT allows to override some of its default parameters with the introduction of command line arguments. Besides the command line arguments, some compilation options are also available by changing some values of variables defined in the "src/config.h" file of PHACT source code.

PHACT is compiled with several default values that influence key components of its execution. The default values are the ones that yield the best average results for the set of benchmarks that were made during PHACT development. Nevertheless, those values may be changed through some variables defined in the "src/config.h" file of PHACT source code, and are identified and described in the following items:

- PRE_FILTER - Set to 1 to enable pre-filtering previous to starting exploration, or to 0 to disable it. Enabled by default;

- SS_MULTIPLIER - Set to 1 to enable a sub-search space multiplier that may be applied to increase the number of sub-search spaces inside each device when blocks are small, or to 0 to disable it. Enabled by default;
- USE_BOOLEAN_VS - Set to 1 to distinguish boolean variables in the CSP, or to 0 to make no distinction. Enabled by default;
- USE_CS_IGNORE - Set to 1 to ignore a constraint when it is fixed or its propagator is unable to propagate more, or to 0 to disable this feature; Available only when using bitmap domains (not for intervals representation). Enabled by default;
- SHARED_SS - Set to 1 to enable work-sharing inside each device, or to 0 to disable it. Disabled by default;
- USE_MORE_BUFFERS - Set to 0 to use only one buffer for backtracking, or to 1 to use more buffers. For some devices, more buffers will allow to use more threads. Enabled by default;
- define USE_MORE_BUFFERS_P - Value to multiply by the maximum buffer size allowed by OpenCL to use on each one of the backtracking buffers. The default is 0.8;
- If any of the following options is enabled, it will be used when no labeling heuristic is defined by the CSP model:
  - SORT_BY_LABEL - Set to 1 to sort variables by the ones that may be labeled. Selected by default;
  - SORT_BY_LABEL_LESS_VALS - Set to 1 to sort variables by the ones that may be labeled and that have less values on their domains;
  - SORT_BY_LABEL_MORE_VALS - Set to 1 to sort variables by the ones that may be labeled and that have more values on their domains;
  - SORT_BY_MOST_USED_CONSTR - Set to 1 to sort constraints on each variable by the constraint that is more common on the CSP.
- USE_LOCAL_MEM - Set to 0 if devices should use only global memory, to 1 if they should use also local memory, or to 2 to decide by default (use only in CPUs and accelerators). Set to 2 by default;
- PRE_LABELING (or Revision) - Set to 0, 1 or 2. 1 to propagate the last labeled variable with all the remaining values before backtracking it, 0 to not, and 2 to use if any propagator is capable of propagating variables with more than one value in its domain. Disabled by default;
- COMPILE_FZN - Set to 0 if the FlatZinc/MiniZinc interpreter is not needed or mzn2fzn, flex or bison are not available, or to 1 to enable the FlatZinc/MiniZinc interpreter. Enabled by default;
- load balancing parameters:
  - GPU_DEFAULT_N_WG - Default number of work-groups to create for a GPU. The default value is 512;
  - GPU_DEFAULT_N_WI - Default number of work-items per work-group to create for a GPU. The default value is 128;
  - GPU_CUTOFF - Approximated number of times that a GPU core (SM) is slower than a CPU core. The default value is 8;
  - ACC_CUTOFF - Approximated number of times that a ACC core is slower than a CPU core. The default value is 4;

- CNT_INIT_PERC - Value to multiply by the total number of Search Spaces (SS) created for calculation of the size of the first block to be sent to each device when counting all the solutions. The default value is 0.25;
- OPT_ONE_INIT_PERC - Value to multiply by the total number of SS created for calculation of the size of the first block to be sent to each device when optimizing or finding the first solution. The default value is 0.01;
- MAX_SS - Maximum number of SS to create. The default value is 1,000,000;
- SS_GPU - Number of SS to create if only one GPU is to be used. The default value is 500,000;
- SS_ACC - Number of SS to create if only one ACC is to be used. The default value is 250,000;
- SS_CPU - Number of SS to create per compute unit, if only one CPU is to be used. The default value is 5,000;
- N_FIRST_BLOCKS - Number of blocks explored before calculating rank. The default value is 3;
- MS_HALF_FIRST_BLOCKS - Minimum milliseconds taken to solve the previous block (of the N_FIRST_BLOCKS - 1) to reduce the size of the next block to half. The default value is 1,000;
- MS_TAKE_ALL - Milliseconds estimated for a device to solve all the remaining sub-search spaces, to make it take all of them in the next block. The default value is 2,000;
- PERCENT_REM_SS_DOUBLE - Value to multiply by the remaining SS to be considered for the calculation of the size of the next block for the device that finished the N_FIRST_BLOCKS first. The default value is 0.2;
- PERCENT_REM_SS_RANK_GPU - Value to multiply by the remaining SS to be considered for the calculation of the size of the next block with RANK for GPUs. The default value is 0.3;
- PERCENT_REM_SS_RANK_CPU - Value to multiply by the remaining SS to be considered for the calculation of the size of the next block with RANK for CPUs. The default value is 0.8;
- PERCENT_REM_SS_RANK_ACC - Value to multiply by the remaining SS to be considered for the calculation of the size of the next block with RANK for ACCs. The default value is 0.6;
- FAST_BLOCKS_MS_OPT - Time taken to solve a block during optimization to be considered exceptionally easy to explore. The default is 2,000;
- FAST_BLOCKS_MS_ONE - Time taken to solve a block when looking for one solution to be considered exceptionally easy to explore. The default is 2,000;
- N_FAST_BLOCKS_OPT - Number of blocks in a row that took less than EMPTY_BLOCKS_-MS_OPT s to solve (each), to double the size of the next block. The default is 3;
- PERCENT_BLOCKS_ADD - Value to multiply by the last block to add to the next block when the previous N_EMPTY_BLOCKS_OPT were EMPTY_BLOCKS_MS_OPT. The default is 0.2;
- N_EMPTY_BLOCKS - Number of blocks with zero SS that a device would receive to be terminated. The default is 1;
- TIMES_USED_TRESHOLD - Number of times that a device must be used in order to double the number of sub-search spaces on the next block, when optimizing or finding one solution until SS_REM_PERC_TRESHOLD. The default is 10;
- SS_REM_PERC_TRESHOLD - Lower limit of SSs that remains to be explored, to stop doubling the size of the next block when TIMES_USED_TRESHOLD, when optimizing or finding one solution. The default is 0.3 of the total SS that were created.

To compile PHACT, one of the following commands must be executed in folder "PHACT/Debug":

- make all - To solve CSPs with variables whose domains have values between 0 and 1023;
- make all CFLAGS="-D BITS=n" - To solve CSPs with variables whose domains have values between 0 and $n-1$. When recompiling the Solver to change CL_BITS value, please run "make clean" before;
- make all CFLAGS="-D COMPILE_FZN=0" - Required when the programs mzn2fzn, flex or bison are not available. MiniZinc and FlatZinc interpreter will not be available.

To execute PHACT, multiple command line arguments are available as described in the "readme.txt" file distributed with PHACT source code and shown when executing the command "./PHACT -H":

- -D [GPU|CPU|ACC][:n][/[wg]/[wi]] - Select the device/s to use. Examples:
  - -D CPU:1/64/1 -D GPU:2 -D ACC//1 - Use first CPU with 64 work-groups and 1 work-item per work-group, second GPU with the default number of work-groups and of work-items, and all accelerators with default number of work-groups and one work-item per work-group;
  - -D CPU -D GPU - Use all GPUs and all CPUs with default number of work-groups and work-items;
  - If no -D argument is introduced, all the devices compatible with OpenCL will be used.
- -E [QUEENS | COSTAS | GOLOMB | SUDOKU | ALL-DIFF | QAP | LANGFORD | STEINER | LATIN | ALL-INTERVAL | MARKET-SPLIT | SCHURS] - Select one of the sample CSPs implemented through the interface of PHACT;
- -FZN /home/user/csp.fzn - Solve the FlatZinc model in the file "/home/user/csp.fzn". If only the mame of the file is given, it will be searched for in src/csps/csp.fzn. flex and bison programs are required;
- -MZN /home/user/csp.mzn /home/user/csp.dzn - Solve the MiniZinc model in the files "/home-/user/csp.mzn" and "/home/user/csp.dzn". If only the name of the files is given, they will be searched in src/csps/csp.Xzn. Mzn2fzn, flex and bison programs are required;
- -MZN /home/user/csp.mzn /home/user/csp.dzn -MZN2FZN-ONLY - Only converts the MZN file in "/home/user/csp.mzn" and "/home/user/csp.dzn" to the FZN file "/home/user/csp.fzn". If only the name of the file is given, it will be searched for in src/csps/csp.Xzn. Mzn2fzn program is required;
- (int) - CSP dimension. "(int)" should be replaced by each dimension of the CSP to solve. Not used when solving a MiniZinc or FlatZinc model;
- [-COUNT | -ONE | -OPT] - Select what must be done with the CSP. When solving FlatZinc models, it overrides the model selection:
  - -COUNT - Count all the solutions;
  - -ONE - Find one solution. Default for CSPs modeled with PHACT C interface;
  - -OPT - Do optimization.
- -INTERVALS - Use interval representation for domains, instead of bitmaps.

- -N-SS n - Number of sub-search spaces to create. "n" should be replaced by the number of sub-search spaces to create. If not present, the default number of sub-search spaces will be created;
- [-FIRST-FAIL | -INPUT-ORDER | -OCCURRENCE | -MAX-REGRET | -SMALLEST] - Method to select the variable to label:
  - -FIRST-FAIL - Select the variable to label that has less values in its domain;
  - -INPUT-ORDER - Select the variable to label by the order in which they were created. Default;
  - -OCCURRENCE - Select the variable to label that is more constrained;
  - -MAX-REGRET - Select the variable to label that has the largest difference between the two smallest values;
  - -SMALLEST - Select the variable to label that has the smallest value in its domains.
- [-MIN-VALUE | -MAX-VALUE | -SPLIT-VALUES] - Method to select the value to assign to the variable for labeling:
  - -MIN-VALUE - Select the minimum value to assign. Default;
  - -MAX-VALUE - Select the maximum value to assign;
  - -SPLIT-VALUES - Splits the domain about half and tries the first half.
- -STATS - Print statistics about the solving process:
  - Total time - Time elapsed between the launch of PHACT and its termination;
  - Solve time - Time elapsed executing search engines;
  - Solutions - Number of solutions found;
  - Search spaces - Number of disjoint sub-search spaces obtained from splitting the main search-space;
  - Variables - Number of CSP variables;
  - Boolean variables - Number of CSP variables that are boolean;
  - Domains range - The minimum and maximum values of any CSP variable;
  - Constraints - Number of CSP constraints;
  - Boolean constraints - Number of CSP constraints whose constrained variables are all boolean;
  - Constraint types - Number and name of the different types of constraints used in the CSP;
  - Variables to label - Number of CSP variables that can be labeled;
  - Maximum depth - Maximum expansion depth of the search tree, for labeling;
  - Labels - Number of variable labelings;
  - Nodes explored - Number of nodes that were explored;
  - Nodes failed - Number of nodes that were detected as inconsistent;
  - Backtracks - Number of backtrackings done;
  - Prunings - Approximated number of prunings;
  - Propagations - Number of times a constraint propagator was executed.
- -PRINT-SOLUTIONS - Try to print all the solutions. Only available when using only one thread per device, and depending on the device and on the number of solutions of the CSP, only some solutions may be printed;

- -PRINT-CSP - Before starting the exploration, prints all the variables with their domains, the constraints and the relation between them;
- -V - Print more information and timings about what is being done by each device;
- -H - Displays all the command line arguments accepted by PHACT.

The OpenCL drivers implemented by some vendors for their devices will try to vectorize the kernel. When using PHACT in some devices, that may result in crashing the OpenCL compiler, or in a poor performance of the solver. For that motive, it is recommended to disable this OpenCL feature by running "CL_CONFIG_USE_VECTORIZER=false" before executing PHACT.

Some execution examples are:

- For counting the number of solutions of the Costas Array 10 problem using all the devices compatible with OpenCL on the running machine, execute the following command on folder "PHACT/Debug":

  ./PHACT -E COSTAS 10 -COUNT

- For finding one solution for the 30-queens problem using all the GPUs compatible with OpenCL on the running machine, execute the following command on folder "PHACT/Debug":

  ./PHACT -E QUEENS 30 -D GPU

- For finding one solution for a new CSP modeled in the file "/src/csps/CSP.c" and using the CPU on the running machine, after recompiling PHACT,execute the following command on folder "PHACT/Debug":

  ./PHACT -D CPU

- For solving the CSP modeled in the FlatZinc file "PHACT/Debug/src/csps/CSP.fzn" file using all the GPUs compatible with OpenCL on the running machine, execute the following command on folder "PHACT/Debug":

  ./PHACT CSP.fzn -D GPU

- For solving the CSP modeled in the MiniZinc files "PHACT/Debug/src/csps/CSP.mzn" and "PHACT/Debug/src/csps/CSP.dzn" files using all the devices compatible with OpenCL on the running machine, execute the following command on folder "PHACT/Debug":

  ./PHACT -MZN CSP.mzn CSP.dzn

C

Constraint satisfaction problems used for benchmarking

For benchmarking PHACT and comparing its results against other solvers, some CSPs were used. All the CSPs were retrieved from the MiniZinc Benchmarks [49]. Table C.1 shows the name of the "mzn" file and the name or content of the "dzn" file from where the FlatZinc model was created. The asterisk that follows the name of some "dzn" files indicate that the respective file was not retrieved from the MiniZinc Benchmarks [49], and that its contents are the text located in the same table cell. For example, the content of the "n8.dzn" file is "n = 8;".

The following definitions describe each one of the CSPs that were used in Chapter 5:

Definition 2. *Balanced academic curriculum problem (BACP) - Identified in this thesis as "bacp". "The BACP is to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible. The curriculum must obey the following administrative and academic regulations:*

Table C.1: Source of the CSPs models and data files

CSP	"mzn" file	data file
bacp_2	bacp-2.mzn	curriculum.mzn.model
bacp_6	bacp-6.mzn	curriculum.mzn.model
bacp_7	bacp-7.mzn	curriculum.mzn.model
cryptanalysis_128_3_4	step1_aes.mzn	kb128_n3_obj4.dzn
cryptanalysis_128_5_11		kb128_n5_obj11.dzn
cryptanalysis_128_5_14		kb128_n5_obj14.dzn
golomb_9	golomb.mzn	09.dzn
golomb_11		11.dzn
golomb_12		12.dzn
java_routing_6_3	trip_6_3.mzn	
java_routing_8_5	trip_8_5.mzn	
langford_3_9	langford.mzn	l_3_9.dzn
langford_3_18		l_3_18.dzn
m_queens_8	mqueens2.mzn	n = 8; (n8.dzn*)
m_queens_12		n12.dzn
m_queens_13		n13.dzn
market_split_s4_07	market_split.mzn	s4_07.dzn
market_split_s5_01		s5_01.dzn
market_split_s5_04		s5_04.dzn
open_stacks_10_10	open_stacks_01.mzn	problem_10_10_1.dzn
open_stacks_20_20_a		problem_20_20_1.dzn
open_stacks_20_20_b		ShawInstances_1.dzn
project_planning_13_7	ProjectPlannertest_13_7.mzn	
project_planning_13_8	ProjectPlannertest_13_8.mzn	
queens_15	queens.mzn	n = 15; (queens_15.dzn*)
queens_17		n = 17; (queens_17.dzn*)

- *Academic curriculum: an academic curriculum is defined by a set of courses and a set of prerequisite relationships among them.*
- *Number of periods: courses must be assigned within a maximum number of academic periods.*
- *Academic load: each course has associated a number of credits or units that represent the academic effort required to successfully follow it.*
- *Prerequisites: some courses can have other courses as prerequisites.*
- *Minimum academic load: a minimum number of academic credits per period is required to consider a student as full time.*
- *Maximum academic load: a maximum number of academic credits per period is allowed in order to avoid overload.*
- *Minimum number of courses: a minimum number of courses per period is required to consider a student as full time.*

- *Maximum number of courses: a maximum number of courses per period is allowed in order to avoid overload.*

The goal is to assign a period to every course in a way that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. An optimal balanced curriculum minimises the maximum academic load for all periods." [30].

Definition 3. *Chosen Key Differential Cryptanalysis - Identified in this thesis as "cryptanalysis". This problem consists in the first step of a cryptanalysis attack against block ciphers to test their level of confidentiality, integrity and signature. In this step the bytes are abstracted by binary values and the CSP model consists in the implementation of the AES rules [28].*

Definition 4. *Golomb ruler - "These problems are said to have many practical applications including sensor placements for x-ray crystallography and radio astronomy. A Golomb ruler may be defined as a set of m integers $0 = a_1 < a_2 < ... < a_m$ such that the $m(m–1)/2$ differences $a_j - a_i, 1 <= i < j <= m$ are distinct. Such a ruler is said to contain m marks and is of length a_m. The objective is to find optimal (minimum length) or near optimal rulers. Note that a symmetry can be removed by adding the constraint that $a_2 - a_1 < a_m - a_m - 1$, the first difference is less than the last. There is no requirement that a Golomb ruler measures all distances up to its length - the only requirement is that each distance is only measured in one way. However, if a ruler does measure all distances, it is classified as a perfect Golomb ruler." [8].*

Definition 5. *Java routing - The problem is categorized by the MiniZinc authors [50] as a real life case, but in its description is only described as "automatically generated from a problem description in Java" [24]. From the model, it may be understood as a routing problem in which multiple map locations must be visited, while visiting some locations prior to others. Each map location must also be visited in a predefined time and the time needed to visit all the locations must be minimized.*

Definition 6. *Langford - "Arrange k sets of numbers 1 to n so that each appearance of the number m is m numbers on from the last. For example, the $L(3, 9)$ problem is to arrange 3 sets of the numbers 1 to 9 so that the first two 1's and the second two 1's appear one number apart, the first two 2's and the second two 2's appear two numbers apart, etc." [11].*

Definition 7. *M-queens - It consists in an optimization problem for which a minimum number of queens must be placed on a chessboard, such that all the paths are covered, each one by a single queen [25].*

Definition 8. *Market Split - "A company with two divisions supplies retailers with several products. The goal is to allocate each retailer to one of the divisions such that division 1 controls $100_{ci}\%$, $0 \leq ci \leq 1$, of the market for product i, and division 2 controls $(100 - 100_{ci})\%$. There are n retailers and $m \leq n$ products. Let a_{ij} be the demand of retailer j for product i, and let d_i be determined as $\lfloor c_i d'_i \rfloor$, where d'_i is the total amount of product i that is supplied to the retailers. The decision variable x_j takes value 1 if retailer j is allocated to division 1 and 0 otherwise. The question is: 'there exist an allocation of the retailers to the divisions such that the desired market split is obtained?'" [1].*

Definition 9. *Open Stacks - "This scheduling problem involves a set of products and a set of customer's orders. Each order requires a specific subset of the products to be completed and sent to the customer. Once an order is started (i.e. its first product is being made) a stack is created for that order. At that time, the order is said to be open. When all products that an order requires have been produced, the stack/order is closed. Because of limited space in the production area, the maximum number of stacks that are used simultaneously, i.e. the number of customer orders that are in simultaneous production, should be minimized. Therefore, a solution for the MOSP is a total ordering of the products describing the production sequence that minimizes the set of simultaneously opened stacks." [13].*

Definition 10. *Project planning - Categorized by the MiniZinc authors [50] as a real life case, in its description is only described as "automatically generated from a problem description in Java" [68]. It may be understood as a scheduling problem for which a set of tasks with predecessors must be appointed while minimizing the amount of time required to execute all the tasks.*

Definition 11. *N-queens - The problem consists in placing n queens in a $n \times n$ chessboard, such that no queen attacks another one [53].*

Bibliography

[1] Aardal, K., Bixby, R.E., Hurkens, C.A.J., Lenstra, A.K., Smeltink, J.W.: Market split and basis reduction: Towards a solution of the cornuéjols-dawande instances. In: Cornuéjols, G., Burkard, R.E., Woeginger, G.J. (eds.) Integer Programming and Combinatorial Optimization. pp. 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg (1999), DOI = 10.1007/3-540-48777-8_1

[2] Advanced Micro Devices Inc.: AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE. White paper, Advanced Micro Devices Inc. (2012)

[3] Alsinet, T., Béjar, R., Cabiscol, A., Fernàndez, C., Manyà, F.: Minimal and redundant SAT encodings for the all-interval-series problem. In: Escrig, M.T., Toledo, F., Golobardes, E. (eds.) CCIA. vol. 2504, pp. 139–144. Springer (2002), DOI = 10.1007/3-540-36079-4_12

[4] 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), DOI = 10.1109/PDP.2014.28

[5] Audemard, G., Lagniez, J.M., Simon, L.: Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In: Järvisalo, M., Van Gelder, A. (eds.) Theory and Applications of Satisfiability Testing – SAT 2013. pp. 309–317. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), DOI = 10.1007/978-3-642-39071-5_23

[6] Barták, R., Salido, M.A.: Constraint satisfaction for planning and scheduling problems. Constraints 16(3), 223–227 (July 2011), DOI = 10.1007/s10601-011-9109-4

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

[8] van Beek, P.: CSPLib problem 006: Golomb rulers. http://www.csplib.org/Problems/prob006, [Online; accessed 29-Jan-2019]

[9] Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. pp. 443–448. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)

[10] Brailsford, S., Potts, C., Smith, B.: Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research 119, 557–581 (1999), DOI = 10.1016/S0377-2217(98)00364-6

[11] Brand, S.: Langford's problem problem in MiniZinc. https://github.com/MiniZinc/minizinc-benchmarks/blob/master/langford/langford.mzn, [Online; accessed 28-August-2019]

[12] Brodtkorb, A.R., Hagen, T.R., Sætra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing 73(1), 4–13 (2013), DOI = 10.1016/j.jpdc.2012.04.003

[13] Cambazard, H., Jussien, N.: Solving the minimum number of open stacks problem with explanation-based techniques. In: Explanation-Aware Computing, Papers from the 2007 AAAI Workshop, Vancouver, British Columbia, Canada, July 22-23, 2007. pp. 14–19 (2007)

[14] Campeotto, F., Dovier, A., Fioretto, F., Pontelli, E.: A GPU implementation of large neighborhood search for solving constraint optimization problems. In: Proceedings of the Twenty-first European Conference on Artificial Intelligence. pp. 189–194. ECAI'14, IOS Press, Amsterdam, The Netherlands, The Netherlands (2014), DOI = 10.3233/978-1-61499-419-0-189

[15] 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), DOI = 10.1007/978-3-319-04132-2_11

[16] 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), DOI = 10.1007/978-3-642-20364-0_9

[17] Caseau, Y., Laburthe, F.: Solving small tsps with constraints. In: Proceedings of the 14th International Conference on Logic Programming. pp. 316–330. MIT Press (1997), DOI = 10.7551/mitpress/4299.003.0028

[18] 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), DOI = 10.1007/978-3-642-04244-7_20

[19] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (Jul 1962), DOI = 10.1145/368273.368557

[20] Diaz, D., Codognet, P., Abreu, S.: Adaptive search distribution. http://cri-dist.univ-paris1.fr/diaz/adaptive/, [Online; accessed 10-February-2018]

[21] Fang, J., Sips, H., Zhang, L., Xu, C., Che, Y., Varbanescu, A.L.: Test-driving intel xeon phi. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. pp. 137–148. ICPE '14, ACM, New York, NY, USA (2014), DOI = 10.1145/2568088.2576799

[22] Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: Gao, G.R., Tseng, Y.C. (eds.) International Conference on Parallel Processing - ICPP 2011. pp. 216–225. IEEE, Taipei, Taiwan (September 2011), DOI = 10.1109/ICPP.2011.45

[23] 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), DOI = 10.1016/j.eswa.2011.07.027

[24] Fischer, J.: Java routing problem in MiniZinc. https://github.com/MiniZinc/minizinc-benchmarks/tree/master/java-routing, [Online; accessed 28-August-2019]

[25] Francis, K.G.: Natural optimisation modelling for software developers. Ph.D. thesis, The University of Melbourne (2016)

[26] Gaster, B., Howes, L., Kaeli, D., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Morgan Kaufmann Publishers Inc., San Francisco, USA (2011), DOI = 10.1016/C2011-0-69669-3

[27] Gent, I.P., Jefferson, C., Miguel, I., Moore, N.C., Nightingale, P., Prosser, P.: A preliminary review of literature on parallel constraint solving. Workshop on parallel methods for constraint solving (2011)

[28] Gerault, D., Minier, M., Solnon, C.: Constraint Programming Models for Chosen Key Differential Cryptanalysis. In: 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016). Lecture Notes in Computer Science, vol. 9892, pp. 584–601. Springer, Toulouse, France (Sep 2016), DOI = 10.1007/978-3-319-44953-1_37

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

[30] Hnich, B., Kiziltan, Z., Walsh, T.: 030: Balanced academic curriculum problem (BACP). http://www.csplib.org/Problems/prob030/, [Online; accessed 20-August-2019]

[31] 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), DOI = 10.1007/978-3-642-23397-5_42

[32] Kheiri, A., Özcan, E.: Constructing constrained-version of magic squares using selection hyper-heuristics. The Computer Journal 57(3), 469–479 (2014), DOI = 10.1093/comjnl/bxt130

[33] Khronos Group: OpenCL Conformant Products. https://www.khronos.org/conformance/adopters/conformant-products/opencl, [Online; accessed 19-March-2018]

[34] Khronos OpenCL Working Group, Lee Howes and Aaftab Munshi: The OpenCL specification version: 2.0 document revision: 29. White paper

[35] Mazure, B., Saïs, L., Grégoire, É.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence 22(3), 319–331 (Jul 1998), DOI = 10.1023/A:1018999721141

[36] Mellanox and Oak Ridge National Laboratory: Reaching the summit with InfiniBand: Mellanox interconnect accelerates world's fastest HPC and artificial intelligence supercomputer at Oak Ridge National Laboratory (ORNL) (2018)

[37] Menouer, T., Baarir, S.: Parallel satisfiability solver based on hybrid partitioning method. In: PDP. pp. 54–60. IEEE (2017), DOI = 10.1109/PDP.2017.70

[38] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: Top500. https://www.top500.org/list/2019/06/" (2019), [Online; accessed 9-July-2019]

[39] Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. INFORMS Journal on Computing 21(3), 363–382 (May 2009), DOI = 10.1287/ijoc.1080.0313

[40] MIT: Sat competition 2015. https://baldur.iti.kit.edu/sat-race-2015/index.php?cat=results (2016), [Online; accessed 2-February-2018]

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

[42] Muth, J.F., Thompson, G.L.: Industrial scheduling. Englewood Cliffs, N.J. : Prentice-Hall (1963), bibliography: p. 380-387

[43] 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), DOI = 10.1007/978-3-540-74970-7_38

[44] NVIDIA Corporation: NVIDIA GF100, World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism. White paper, NVIDIA Corporation (2010)

[45] NVIDIA Corporation: OpenCL Best Practices Guide 1.0. White paper, NVIDIA Corporation (May 2010)

[46] NVIDIA Corporation: NVIDIA's next Generation^{TM}, CUDA Compute Architecture: Kepler^{TM} GK110, The Fastest, Most Efficient HPC Architecture Ever Built. White paper, NVIDIA Corporation (2012)

[47] NVIDIA Corporation: NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made. White paper, NVIDIA Corporation (2014)

[48] Open MPI Team: Open MPI: Open source high performance computing. http://www.open-mpi.org/, [Online; accessed 17-January-2018]

[49] Optimisation Research Group NICTA: MiniZinc and FlatZinc. http://www.minizinc.org/, [Online; accessed 15-May-2019]

[50] Optimisation Research Group NICTA: Problems. https://www.minizinc.org/mznc_list_of_problems_and_globals.html, [Online; accessed 15-May-2019]

[51] Palù, A.D., Dovier, A., Formisano, A., Pontelli, E.: Exploiting unexploited computing resources for computational logics. In: CILC (2012)

[52] Parberry, I.: An efficient algorithm for the knight's tour problem. Discrete Applied Mathematics 73(3), 251–260 (March 1997), DOI = 10.1016/S0166-218X(96)00010-8

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

[54] 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)

[55] 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

[56] Rezgui, M.: eps-gecode. https://github.com/aeolus-project/eps-gecode, [Online; accessed 15-February-2019]

[57] Roque, P.: PHACT_src. https://git.xdi.uevora.pt/proque/phact_src, [Online; accessed 29-November-2019]

[58] Roque, P., Pedro, V., Abreu, S.: Load balancing for constraint solving with GPUs. INForum 2016, Oitavo Simpósio de Informática (2016)

[59] Roque, P., Pedro, V., Abreu, S.: Constraint solving on hybrid systems. Declare 2017 –Conference on Declarative Programming (2017), DOI = 10.1007/978-3-030-00801-7_1

[60] Roque, P., Pedro, V., Diaz, D., Abreu, S.: Improving constraint solving on parallel hybrid systems. 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI) (2018)

[61] Rossi, F., Beek, P.v., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence, Elsevier (2006)

[62] Roy, P.V.: Logic programming in oz with mozart. In: In International Conference on Logic Programming (ICLP 99. pp. 38–51. The MIT Press (1999)

[63] 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), DOI = 10.1007/978-3-642-40627-0_45

[64] 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)

[65] Schulte, C.: Programming Constraint Services: High-level Programming of Standard and New Constraint Services. Springer-Verlag, Berlin, Heidelberg (2002), DOI = 10.1017/S1471068403211935

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

[67] Smith, B.M., Petrie, K.E., Gent, I.P.: Models and symmetry breaking for 'peaceable armies of queens'. In: Régin, J.C., Rueher, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 271–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2004), DOI = 10.1007/978-3-540-24664-0_19

[68] Tack, G.: Project planning problem in MiniZinc. https://github.com/MiniZinc/minizinc-benchmarks/tree/master/project-planning, [Online; accessed 28-August-2019]

[69] Tendler, J.M., Dodson, J.S., Fields, J.S., Le, H., Sinharoy, B.: Power4 system microarchitecture. IBM J. Res. Dev. 46(1), 5–25 (Jan 2002), DOI = 10.1147/rd.461.0005

[70] Tompson, J., Schlachter, K.: An introduction to the OpenCL programming model (2012)

[71] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: Proceedings of the 3rd Conference on Computing Frontiers. pp. 9–20. CF '06, ACM, New York, NY, USA (2006), DOI = 10.1145/1128022.1128027

[72] Xie, F., Davenport, A.: Solving scheduling problems using parallel message-passing based constraint programming. In: Salido, M.A., Barták, R. (eds.) Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS 2009. pp. 53–58. Thessaloniki, Greece (September 2009)

Index

All Interval Series problem, 27
average time per propagation, 39

backtracking search, 3, 20, 28, 46, 52
Balanced academic curriculum problem, 48, 97
bison tool, 92, 94
bitmaps, 44, 45
blocks of sub-search spaces, 2, 36, 37, 42, 71
boolean satisfaction problems (SAT), 27, 28

Choco, 24, 50, 73, 76, 77
Chosen Key Differential Cryptanalysis problem, 48, 99
complete search, 2, 19, 26, 28
Compute Unified Device Architecture (CUDA), 2, 6, 7, 9, 32
compute unit, 32
constant memory, 33
constraint propagation, 20, 25, 27, 44, 45
Constraint Satisfaction Problem (CSP), 1, 47, 57, 97
Costas Array problem, 12, 20

Embarrassingly Parallel Search (EPS), 23

filtering, 31
first-fail, 20, 45
FlatZinc, 46, 85, 94
FlatZinc interpreter, 32, 92, 94
flex tool, 92, 94

Gecode, 23, 50, 73, 76
global memory, 7, 11, 12, 33
Golomb Ruler problem, 23
Golomb ruler problem, 48, 99

host, 33, 45

incomplete search, 2, 26, 28
Intel Many Integrated Cores (Intel MICs), 5, 33, 50, 55
intervals, 45

Java Routing problem, 49, 99

kernel, 33
Knights problem, 23

labeling, 20, 36, 45, 46, 85, 95
labeling heuristics, 20, 26, 45, 92, 95
Langford Numbers problem, 49, 58, 68, 99
leftmost, 20, 45
load balancing, 22, 24, 25, 28, 37, 43, 46
local memory, 33
local search, 26

M-queens problem, 49, 64, 99
Market Split problem, 49, 99
MiniZinc, 46, 85, 94
multiplier factor, 42
mzn2fzn tool, 32, 48, 92, 94

N-queens problem, 12, 20, 49, 51, 64, 100

occurrence, 45
Open Computing Language (OpenCL), 2, 3, 16, 32–35, 37, 40, 43, 44, 46, 50, 81–83, 96
Open Stacks problem, 49, 99
OpenCL device, 33

OpenMPI, 26
optimization, 40
OR-Tools, 24, 50, 73, 76, 77

Perfect Square Placement problem, 27
portfolio search, 24
private memory, 33
Project planning problem, 49, 71, 100

Queen-Armies problem, 23

rank, 39
reification, 44, 86
revision, 44

search engine, 35, 42, 43
search space, 21, 28, 35–37, 42, 43, 57, 82, 95
search-engine, 46
shared memory, 7, 10, 12, 13, 81
Single-Instruction Multiple-Data (SIMD), 9
Single-Instruction Multiple-Threads (SIMT), 7, 14
Streaming Multiprocessor (SM), 6, 10, 14, 15, 33

value assignment heuristics, 20, 26, 45, 95
variable selection heuristics, 20

warp, 7, 9, 14
work distribution, 2
work stealing, 21–23, 37, 43, 82
work-group, 14, 15, 17, 33, 53
work-item, 17, 33, 53

UNIVERSIDADE DE ÉVORA
INSTITUTO DE INVESTIGAÇÃO
E FORMAÇÃO AVANÇADA

Contactos:
Universidade de Évora
Instituto de Investigação e Formação Avançada — IIFA
Palácio do Vimioso | Largo Marquês de Marialva, Apart. 94
7002 - 554 Évora | Portugal
Tel: (+351) 266 706 581
Fax: (+351) 266 744 677
email: iifa@uevora.pt